

ReactiveML

The temporal expressiveness of synchronous languages
with the power of functional programming.

Louis Mandel

Marc Pouzet

Cédric Pasteur

Guillaume Baudart, Mehdi Dogguy, Louis Jachiet

What is ReactiveML?

Why?

Programming Reactive Systems with :

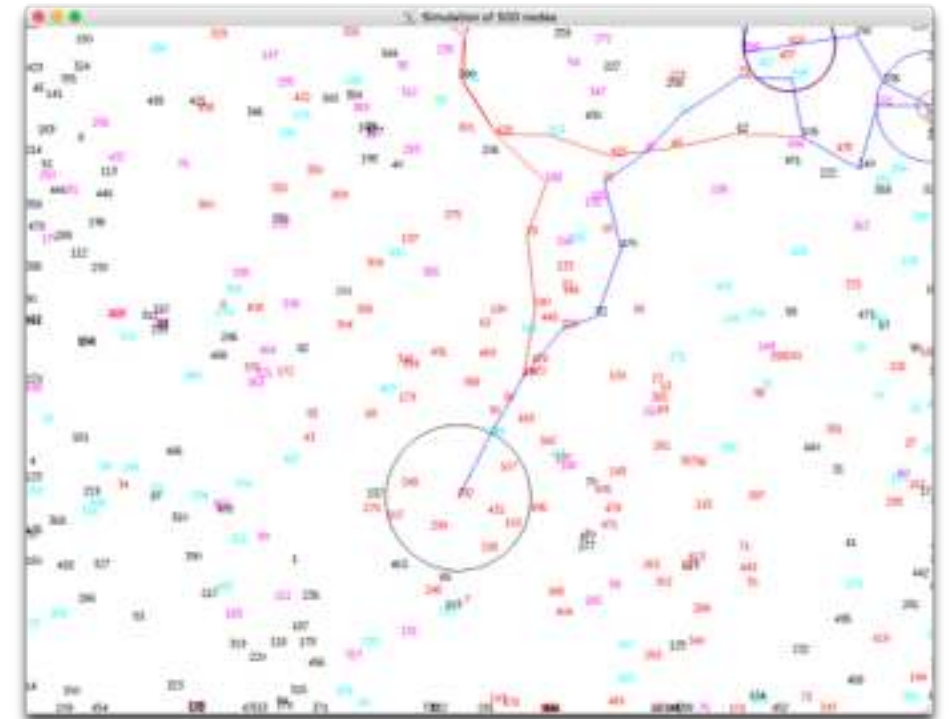
- complex control and data structures
- lots of | communication
synchronization
concurrency

What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of communication
synchronization
concurrency

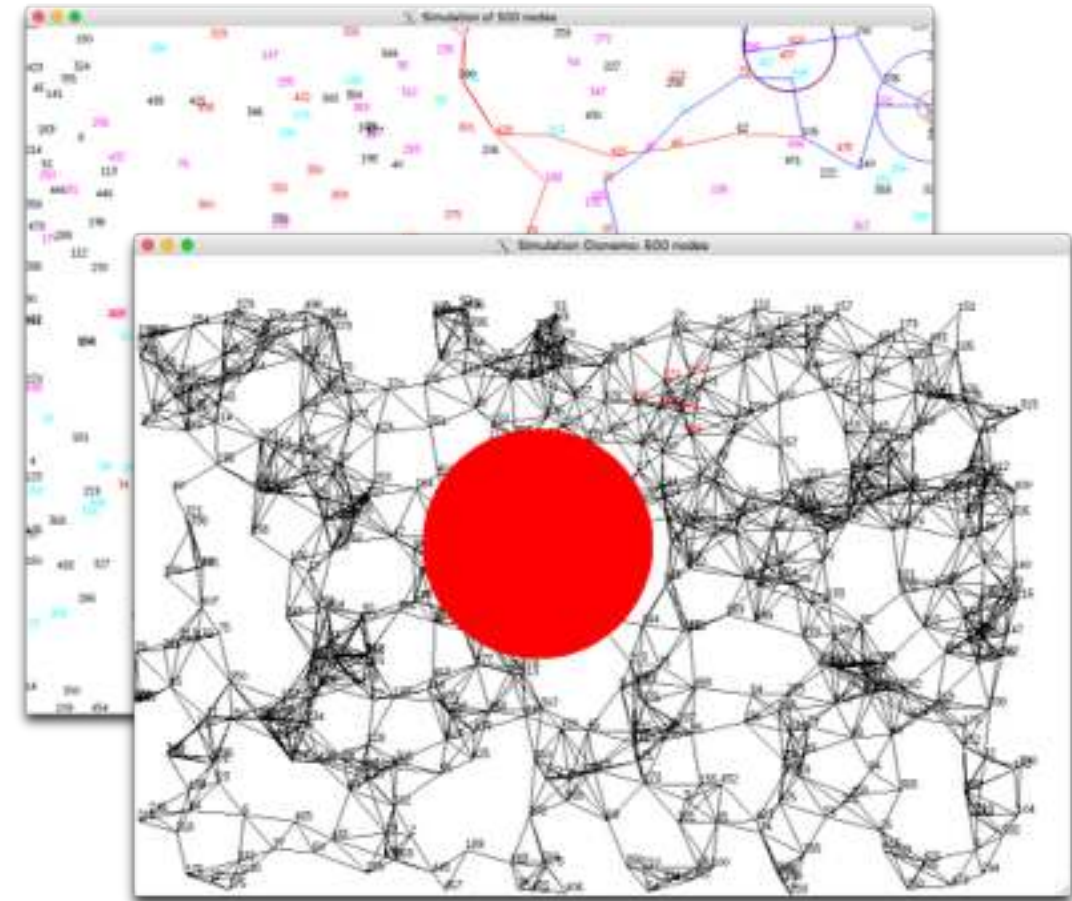


What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of communication
synchronization
concurrency

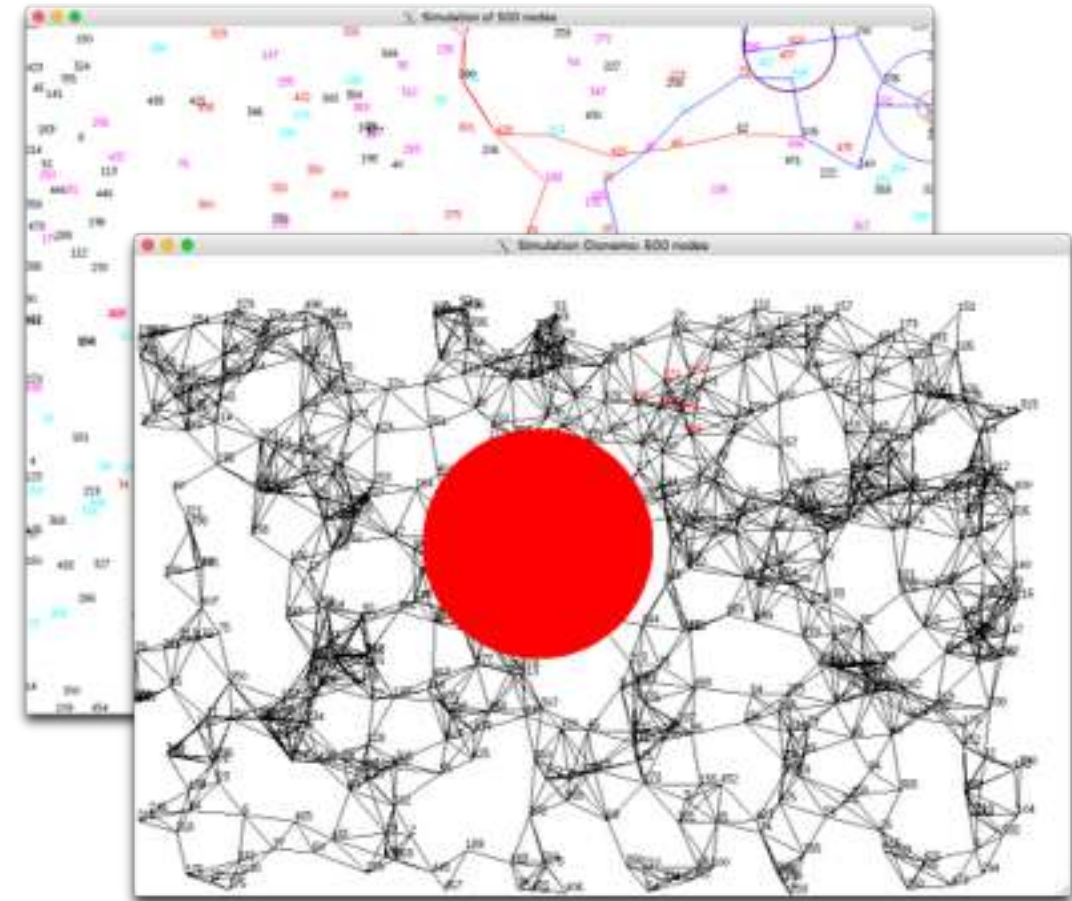


What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of communication
synchronization
concurrency



What?

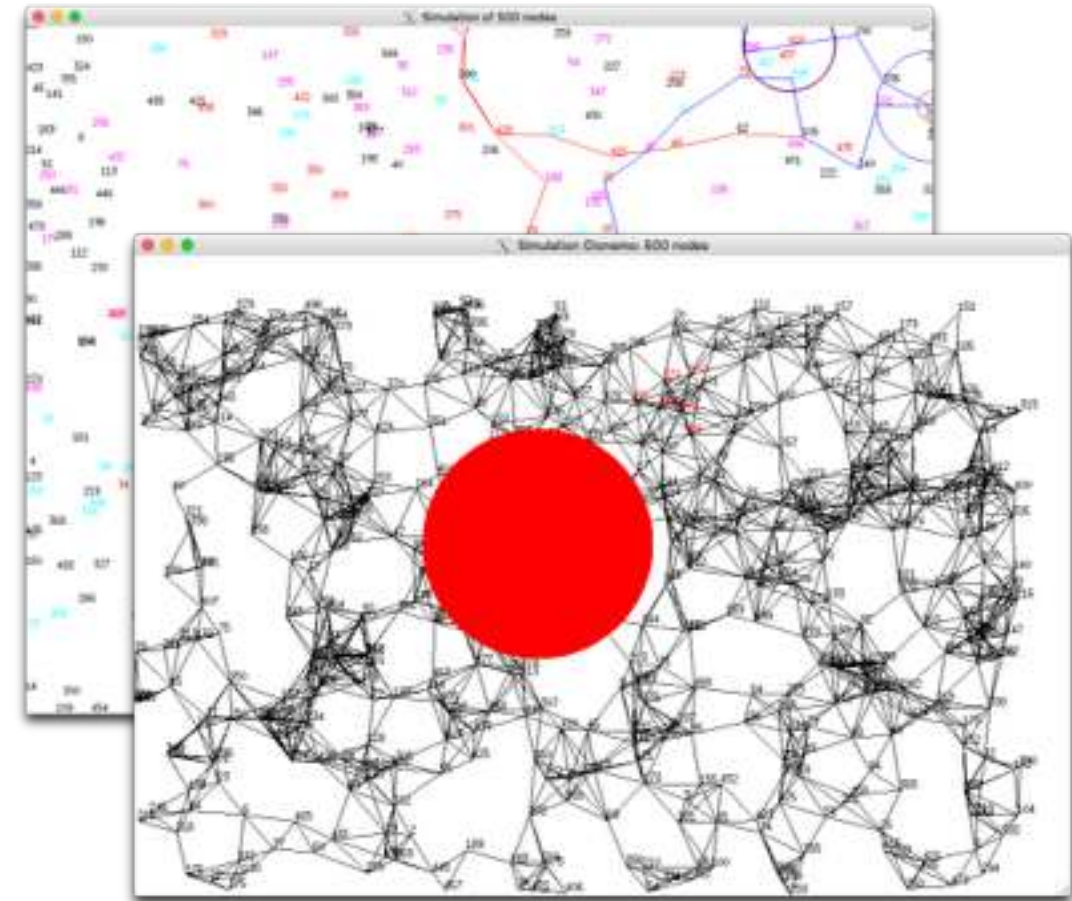
General purpose programming language

What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of communication
synchronization
concurrency



What?

General purpose programming language

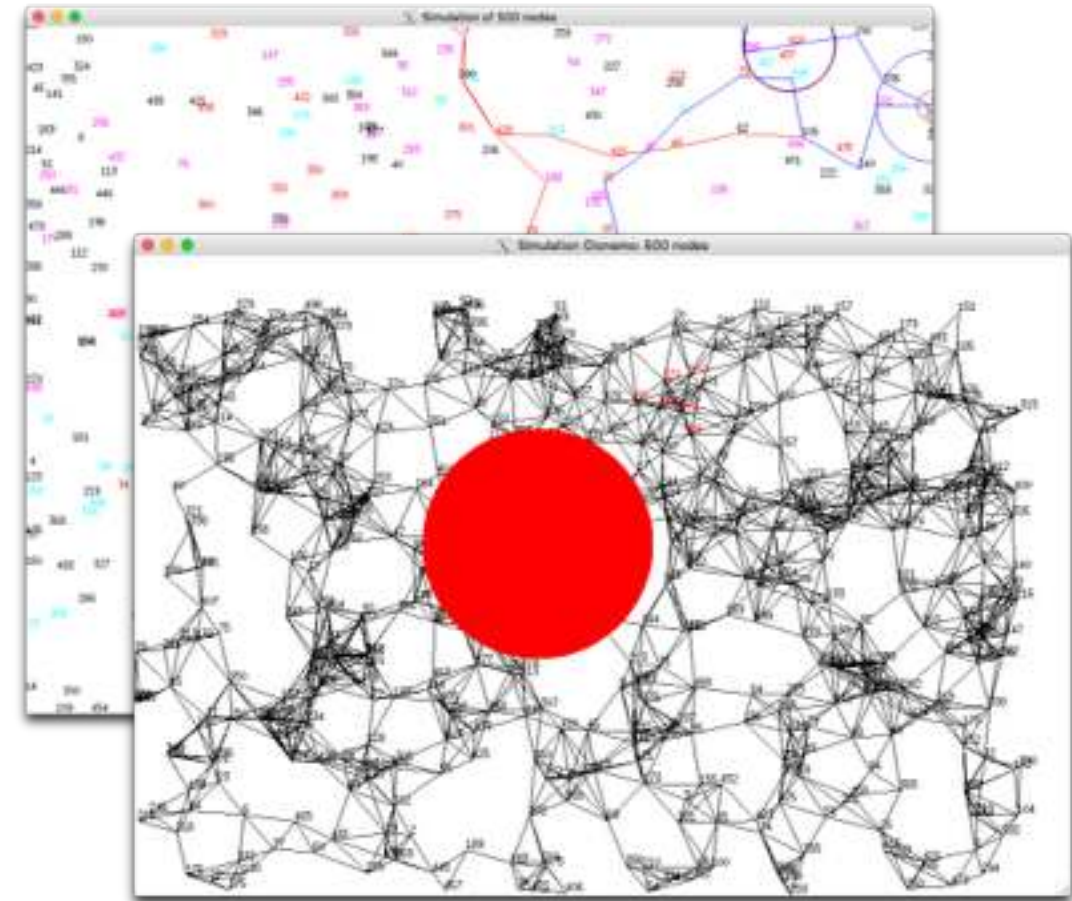
- Functional à la ML (OCaml)
- Synchronous lock-step concurrency (Esterel)
- Dynamic creation of processes

What is ReactiveML?

Why?

Programming Reactive Systems with :

- complex control and data structures
- lots of communication
synchronization
concurrency



What?

General purpose programming language

- Functional à la ML (OCaml)
- Synchronous lock-step concurrency (Esterel)
- Dynamic creation of processes

*No real-time nor bounded
memory constraints*

ppp'05

ReactiveML, a Reactive Extension to ML*

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6[†]

ABSTRACT

We present REACTIVEML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Bousinot. This model combines the so-called *synchronous* model found in ESTEREL which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics *à la* ESTEREL and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type inference system and programs are compiled into regular ML programs. The language has been used for programming several complex simulation problems (e.g., routing protocols in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement safety critical real-time systems. It is founded on the ideal zero delay model where communications and computations are supposed to be instantaneous. In this model, time is defined logically as the sequence of reactions of the system to input events. The main consequence of this model is to conciliate parallelism — allowing for a modular description of the system — and determinism. Moreover, techniques were proposed for this parallelism to be statically compiled, i.e, parallel programs are translated into purely sequential imperative code

[†]Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie, 8 rue du Capitaine Scott, 75015 Paris, France. email: {Louis.Mandel, Marc.Pouzet}@lip6.fr

*This work is supported by the French ACT Sécurité Alidecs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

in terms of transition systems [5, 14].

Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's, Bousinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (or simply *reactive*) and identified inside SL [11], a synchronous reactive calculus derived from ESTEREL. Later on, the JUNIOR [15] calculus was introduced as a way to give a semantics to the SUGARCUBES [13], this last one being an embedding of the reactive model inside JAVA. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the reactive model have been developed [7, 13, 25, 27]. These implementations have been proposed in the form of libraries inside general purpose programming languages. The “library” approach was indeed very attractive because it gives access to all the features of the host language and it is relatively light to implement. Nonetheless, this approach can lead to confusions between values from the host languages used for programming the instant and reactive constructs. This can lead to re-entrance phenomena which are usually detected by run-time tests. Moreover, signals in the reactive model are subject to dynamic scoping rules, making the reasoning on programs hard. Most importantly, implementations of the reactive model have to compete with traditional (mostly sequential) implementation techniques of complex simulation problems. This calls for specific compilation, optimization and program analysis techniques which can be hardly done with the library approach.

The approach we choose is to provide concurrency at language level. We enrich a strict ML language with new primitives for reactive programming. We separate regular ML expressions from reactive ones through the notion of a *process*. An ML expression is considered to be an atomic (timeless) computation whereas a process is a state machine whose be-

ppp'05

ReactiveML, a Reactive Extension to ML*

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6[†]

ABSTRACT

We present REACTIVEML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Bousinot. This model combines the so-called *synchronous* model found in ESTEREL which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics à la Esterel and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type theory. Systems and programs are compiled into regular ML programs. The language has been used for programming event complex simulation problems (e.g., routing or models in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement safety critical real-time systems. It is based on the ideal zero delay model where communications and computations are supposed to be instantaneous. In this model, time is defined logically as the sequence of reactions of the system to input events. The main consequence of this model is to conciliate parallelism allowing for a modular description of the system — and determinism. Moreover, techniques were proposed for this parallelism to be statically compiled, i.e., parallel processes are translated into purely sequential imperative code.

[†]Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie, 8 rue du Capitaine Scott, 75015 Paris, France.
email: {Louis.Mandel, Marc.Pouzet}@lip6.fr

*This work is supported by the French ACT Sécurité Alidecs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

in terms of transition systems [5, 14].

Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's Bousinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (or simply *reactive*) and identified inside SL [11], a new reactive calculus derived from ESTEREL. Later on, the JUNIOR [15] calculus was introduced as a way to give semantics to the SUGARCUBES [13], this last one being an embedding of the reactive model inside JAVA. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the reactive model have been developed [7, 13, 25, 27]. These implementations have been proposed in the form of libraries inside general purpose programming languages. The “library” approach was indeed very attractive because it gives access to all the features of the host language and it is relatively light to implement. Nonetheless, this approach can lead to confusions between values from the host languages used for programming the instant and reactive constructs. This can lead to re-entrance phenomena which are usually detected by run-time tests. Moreover, signals in the reactive model are subject to dynamic scoping rules, making the reasoning on programs hard. Most importantly, implementations of the reactive model have to compete with traditional (mostly sequential) implementation techniques of complex simulation problems. This calls for specific compilation, optimization and program analysis techniques which can be hardly done with the library approach.

The approach we choose is to provide concurrency at language level. We enrich a strict ML language with new primitives for reactive programming. We separate regular ML expressions from reactive ones through the notion of a *process*. An ML expression is considered to be an atomic (timeless) computation whereas a process is a state machine whose be-

ReactiveML, a Reactive Extension to ML*

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6†

ABSTRACT

We present REACTIVEML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Boussinot. This model combines the so-called *synchronous* model found in ESTEREL which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics à la Esterel and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type theory. Systems and programs are compiled into regular ML programs. The language has been used for programming several complex simulation problems (e.g., routing problems in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement real-time systems. It is based on the idea where communications and computation be instantaneous. In this model, time is the sequence of reactions of the system. The main consequence of this model is the possibility of allowing for a modular description and formal verification. Moreover, techniques of parallelism can be statically compiled into sequential code.

Laboratoire d'Informatique de Paris 6,
Marie Curie, 8 rue du Capitaine Scott,
email: {Louis.Mandel, Marc.Pouzet}@lip6.fr
*This work is supported by the French ANR.

in terms of transition systems [5, 14].

Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's, Boussinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (or simply *reactive*) and identified inside SL [11], a synchronous reactive calculus derived from ESTEREL. Later on, the JUNIOR [15] calculus was introduced as a way to give semantics to the SUGARCUBES [13], this last one being an embedding of the reactive model inside JAVA. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the re-

A Synchronous Embedding of Antescofo, a Domain-Specific Language for Interactive Mixed Music

Guillaume Baudart* Florent Jacquemard†
Louis Mandel‡ Marc Pouzet§

Programming Mixed Music in ReactiveML

Guillaume Baudart
École normale supérieure de Cachan
Antenne de Bretagne
DI, École normale supérieure
Guillaume.Baudart@ens-cachan.org

Louis Mandel
Univ. Paris-Sud 11
DI, École normale supérieure
INRIA Paris-Rocquencourt
Louis.Mandel@inria.fr

Marc Pouzet
Univ. Pierre et Marie Curie
DI, École normale supérieure
INRIA Paris-Rocquencourt
Marc.Pouzet@ens.fr

Abstract

Mixed music is about live musicians interacting with electronic parts which are controlled by a computer during the performance. It allows composers to use and combine traditional instruments with complex synthesized sounds and other electronic devices. There are several languages dedicated to the writing of mixed music scores. Among them, the Antescofo language coupled with an advanced score follower allows a composer to manage the reactive aspects of musical performances: how electronic parts interact with a musician. However these domain specific languages do not offer the expressiveness of functional programming.

We embed the Antescofo language in a reactive functional programming language, ReactiveML. This approach offers to the composer recursion, higher order, inductive types, as well as a simple way to program complex reactive behaviors thanks to the synchronous model of concurrency on which ReactiveML is built. This article presents how to program mixed music in ReactiveML through several examples.

Categories and Subject Descriptors: D.3.2 [Language Classification]: Concurrent, distributed, parallel languages; H.5.5 [Information Interfaces and Presentation]: Sound and Music Computing.

Keywords: Synchronous Programming; Language Embedding; Mixed Music; Live Coding.

1. Introduction

Technical progresses since the 1950's have led composers to write music mixing live performers with electronic parts. At first, performers simply followed a pre-registered magnetic band. But the advances in computing rapidly allowed real interaction between musicians and electronic parts.

Developed at IRCAM, Antescofo [5] is a state-of-the-art score following system dedicated to mixed music. Since 2000, Antescofo has been used in the creation of more than 40 original mixed electronic pieces by world renowned artists and ensembles, including

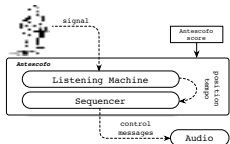
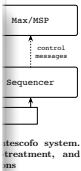


Figure 1. Architecture of the Antescofo system. Continuous arrow represent pre-treatment, and dotted ones real-time communications

Pierre Boulez, Philippe Manoury, Marco Stroppa, New York Philharmonics, Berlin Philharmonics, and the Radio France Orchestra. Figure 1 describes the organization of the Antescofo system. It is composed of two distinct subsystems: a listening machine and a sequencer. During a performance, the *listening machine* estimates the tempo (i.e., execution speed) and the position of the live performers in the score. The role of the *sequencer* is to use these information to trigger electronic actions by sending control messages to a music programming environment, e.g., Max/MSP². Then, the environment uses these messages to handle complex sound synthesis, manage lights, etc.

The language of Antescofo [9] is a descriptive language inspired by classical western music notation: a score is an interleaving of instrumental notes and electronic actions. An original aspect of this language is that it specifies synchronization and error-handling strategies for the electronic actions. It means that during the performance, depending on the tempo variations and the errors made by the musician or the listening machine, the sequencer will react in a way which is defined by the composer.

The sequencer of Antescofo is a *synchronous* reactive system: it continuously reads inputs from the listening machine, produces outputs to the musical environment and is subject to the strong timing requirement related to ear tolerance (typically 30ms). Synchronous languages [2] have been invented and are used widely³ for programming a wide range of critical control software: on board control of trains and track interlocking, fly-by-wire and cockpit displays for planes, etc. Therefore, they would have been well suited for programming the sequencer. But, to ensure statically that programs execute in bounded time and memory, synchronous lan-



Embedding: Error
Mixed Music.

a growing interest parts and live musicians and industry have musicians and composers the development musical interaction, lifted from the composer Philippe userCollider [12], tem and dedicated se, synchronization i computer music, or interacting with using this system, mental and elec-

ed in the creation e pieces by world ing Pierre Boulez, York Philharmon-France Orchestra, e of the system. It listening machine the listening ma-

†<http://jepous.srca.fr/antescofo>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PARM '13, September 28, 2013, Boston, MA, USA.
Copyright © 2013 ACM 978-1-4503-2386-4/13/09...\$15.00.
<http://dx.doi.org/10.1145/2385341.2385344>

²<http://cycling74.com/>

³<http://www.estavel-technologies.com>

ppnp'05

ReactiveML, a Reactive Extension to ML^{*}

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6[†]

ABSTRACT

We present REACTIVEML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Boussinot. This model combines the so-called *synchronous* model found in ESTEREL which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics à la Esterel and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type theory. Systems and programs are compiled into regular ML programs. The language has been used for programming event complex simulation problems (e.g., routing problems in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement time systems. It is based on the idea where communications and computation be instantaneous. In this model, time as the sequence of reactions of the system. The main consequence of this model is to allow for a modular description and formal verification. Moreover, techniques of parallelism to be statically compiled are translated into purely sequential

^{*}Laboratoire d'Informatique de Paris 6, Marie Curie, 8 rue du Capitaine Scott, 75005 Paris, France.
[†]email: {Louis.Mandel, Marc.Pouzet}@lip6.fr
[‡]This work is supported by the French ANR-05-0001-01 project.

in terms of transition systems [5, 14].

Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's, Boussinot observed that it was possible to reconcile the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (or simply *reactive*) and identified inside SL [11], a new calculus of reactive calculus derived from ESTEREL. Later on, the JUNIOR [15] calculus was introduced as a way to give semantics to the SUGARCUBES [13], this last one being an embedding of the reactive model inside JAVA. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the re-

A Synchronous Embedding of Antescofo, a Domain-Specific Language for Interactive Mixed Music

Guillaume Baudart^{*}

Florent Jacquemard[†]

Louis Mandel[‡]

Marc Pouzet[§]

Programming Mixed Music in ReactiveML

Guillaume Baudart
École normale supérieure de Cachan
Antenne de Bretagne
DI, École normale supérieure
Guillaume.Baudart@ens-cachan.org

Louis Mandel
Univ. Paris-Sud 11
DI, École normale supérieure
INRIA Paris-Rocquencourt
Louis.Mandel@inria.fr

Marc Pouzet
Univ. Pierre et Marie Curie
DI, École normale supérieure
INRIA Paris-Rocquencourt
Marc.Pouzet@ens.fr

Abstract

Mixed music is about live musicians interacting with electronic parts which are controlled by a computer during the performance. It allows composers to use and combine traditional instruments with complex synthesized sounds and other electronic devices. There are several languages dedicated to the writing of mixed music scores. Among them, the Antescofo language coupled with an advanced score follower allows a composer to manage the reactive aspects of musical performances: how electronic parts interact with a musician. However these domain specific languages do not offer the expressiveness of functional programming.

We embed the Antescofo language in a reactive functional programming language, ReactiveML. This approach offers to the composer recursion, higher order, inductive types, as well as a simple way to program complex reactive behaviors thanks to the synchronous model of concurrency on which ReactiveML is built. This article presents how to program mixed music in ReactiveML through several examples.

Categories and Subject Descriptors: D.3.2 [Language Classification]: Concurrent, distributed languages; H.5.5 [Information Interfaces and Presentation]: Sound and Music Computing.

Keywords: Synchronous Programming; Language Embedding; Mixed Music; Live Coding.

1. Introduction

Technical progresses since the 1950's have led composers to write music mixing live performers with electronic parts. At first, performers simply followed a pre-registered magnetic band. But the advances in computing rapidly allowed real interaction between musicians and electronic parts.

Developed at IRCAM, Antescofo [5][†] is a state-of-the-art score following system dedicated to mixed music. Since 2000, Antescofo has been used in the creation of more than 40 original mixed electronic pieces by world renowned artists and ensembles, including

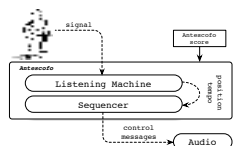


Figure 1. Architecture of the Antescofo system. Continuous arrow represent pre-treatment, and dotted ones real-time communications

Pierre Boulez, Philippe Manoury, Marco Stroppa, New York Philharmonics, Berlin Philharmonics, and the Radio France Orchestra. Figure 1 describes the organization of the Antescofo system. It is composed of two distinct subsystems: a listening machine and a sequencer. During a performance, the *listening machine* estimates the tempo (i.e., execution speed) and the position of the live performers in the score. The role of the *sequencer* is to use these information to trigger electronic actions by sending control messages to a music programming environment, e.g., Max/MSP[‡]. Then, the environment uses these messages to handle complex sound synthesis, manage lights, etc.

The language of Antescofo [9] is a descriptive language inspired by classical western music notation: a score is an interleaving of instrumental notes and electronic actions. An original aspect of this language is that it specifies synchronization and error-handling strategies for the electronic actions. It means that during the performance, depending on the tempo variations and the errors made by the musician or the listening machine, the sequencer will react in a way which is defined by the composer.

The sequencer of Antescofo is a *synchronous* reactive system: it continuously reads inputs from the listening machine, produces outputs to the musical environment and is subject to the strong timing requirement related to ear tolerance (typically 30ms). Synchronous languages [2] have been invented and are used widely[§] for programming a wide range of critical control software: on board control of trains and track interlocking, fly-by-wire and cockpit displays for planes, etc. Therefore, they would have been well suited for programming the sequencer. But, to ensure statically that programs execute in bounded time and memory, synchronous lan-

[†]<http://pegusa.srccn.fr/antescofo>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or fee. Request permissions from permissions@acm.org.

PARM '13, September 28, 2013, Boston, MA, USA.
Copyright © 2013 ACM 978-1-4503-2386-4/13/09...\$15.00.
<http://dx.doi.org/10.1145/2385341.2385344>

SAS'14

Reactivity of Cooperative Systems Application to ReactiveML

Louis Mandel^{1,3} and Cédric Pasteur^{2,3}

¹ Collège de France

² DI École normale supérieure (now at ANSYS-Esterel Technologies)

³ INRIA Paris-Rocquencourt

Abstract. Cooperative scheduling enables efficient sequential implementations of concurrency. It is widely used to provide lightweight threads facilities as libraries or programming constructs in many programming languages. However, it is up to programmers to actually cooperate to ensure the reactivity of their programs.

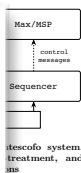
We present a static analysis that checks the reactivity of programs by abstracting them into so-called *behaviors* using a type-and-effect system. Our objective is to find a good compromise between the complexity of the analysis and its precision for typical reactive programs. The simplicity of the analysis is mandatory for the programmer to be able to understand error messages and how to fix reactivity problems. Our work is applied and implemented in the functional synchronous language ReactiveML. It handles recursion, higher-order processes and first-class signals. We prove the soundness of our analysis with respect to the big-step semantics of the language: a well-typed program with reactive effects is reactive. The analysis is easy to implement and generic enough to be applicable to other models of concurrency such as coroutines.

1 Introduction

Most programming languages offer lightweight thread facilities, either integrated in the language like the asynchronous computations [30] of F#, or available as a library like GNU Pth [12] for C, Concurrent Haskell [15] or Lwt [34] for OCaml. These libraries are based on cooperative scheduling: each thread of execution cooperates with the scheduler to let other threads execute. This enables an efficient and sequential implementation of concurrency, allowing to create up to millions of separate threads, which is impossible with operating system threads. Synchronization also comes almost for free, without requiring synchronization primitives like locks.

The downside of cooperative scheduling is that it is necessary to make sure that threads actually cooperate:

- Control must regularly be returned to the scheduler. This is particularly true for infinite loops, which are very often present in *reactive* and *interactive* systems.
- Blocking functions, like operating system primitives for I/O, cannot be called.



Embedding: Error
Mixed Music.

a growing interest parts and live musicians and industry have musicians and composers the development musical interaction, lifted from the composer Philippe userCollider [12], tem and dedicated se, synchronization i computer music, or interacting with using this system, mental and electrical in the creation e pieces by world ing Pierre Boulez, York Philharmon-France Orchestra, r of the system. It listening machine the listening ma-

Application: Mixed Music

ReactiveML, a Reactive Extension to ML^{*}

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6[†]

ABSTRACT

We present REACTIVEML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Bousinot. This model combines the so-called *synchronous* model found in ESTEREL which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics à la Esterel and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type theory. Systems and programs are compiled into regular ML programs. The language has been used for programming event complex simulation problems (e.g., routing problems in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement time systems. It is based on the idea where communications and computation be instantaneous. In this model, time as the sequence of reactions of the system. The consequence of this model is logical time allowing for a modular description of the system. Moreover, techniques of parallelism to be statically compiled are translated into purely sequential

Laboratoire d'Informatique de Paris 6,
Marie Curie, 8 rue du Capitaine Scott,
email: {Louis.Mandel¹, Marc.Pouzet²}
^{*}This work is supported by the French A

in terms of transition systems [5, 14].

Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's Bousinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (simply *reactive*) and identified inside SL [11], a new change reactive calculus derived from ESTEREL. Later on, the JUNIOR [15] calculus was introduced as a way to give semantics to the SUGARCUBES [13], this last one being an embedding of the reactive model inside JAVA. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the re-

A Synchronous Embedding of Antescofo, a Domain-Specific Language for Interactive Mixed Music

Guillaume Baudart^{*} Florent Jacquemard[†]
Louis Mandel[‡] Marc Pouzet[§]

Programming Mixed Music in ReactiveML

Guillaume Baudart
École normale supérieure de Cachan
Antenne de Bretagne
DI, École normale supérieure
Guillaume.Baudart@ens-cachan.org

Louis Mandel
Univ. Paris-Sud 11
DI, École normale supérieure
INRIA Paris-Rocquencourt
Louis.Mandel@inria.fr

Marc Pouzet
Univ. Pierre et Marie Curie
DI, École normale supérieure
INRIA Paris-Rocquencourt
Marc.Pouzet@ens.fr

Abstract

Mixed music is about live musicians interacting with electronic parts which are controlled by a computer during the performance. It allows composers to use and combine traditional instruments with complex synthesized sounds and other electronic devices. There are several languages dedicated to the writing of mixed music scores. Among them, the Antescofo language coupled with an advanced score follower allows a composer to manage the reactive aspects of musical performances: how electronic parts interact with a musician. However these domain specific languages do not offer the expressiveness of functional programming.

We embed the Antescofo language in a reactive functional programming language, ReactiveML. This approach offers to the composer recursion, higher order, inductive types, as well as a simple way to program complex reactive behaviors thanks to the synchronous model of concurrency on which ReactiveML is built. This article presents how to program mixed music in ReactiveML through several examples.

Categories and Subject Descriptors: D.3.2 [Language Classification]: Concurrent, distributed, parallel languages; H.5.5 [Information Interfaces and Presentation]: Sound and Music Computing.

Keywords: Synchronous Programming; Language Embedding; Mixed Music; Live Coding.

1. Introduction

Technical progress since the 1950's have led composers to write music mixing live performers with electronic parts. At first, performers simply followed a pre-registered magnetic band. But the advances in computing rapidly allowed real interaction between musicians and electronic parts.

Developed at IRCAM, Antescofo [5] is a state-of-the-art score following system dedicated to mixed music. Since 2008, Antescofo has been used in the creation of more than 40 original mixed electronic pieces by world renowned artists and ensembles, including

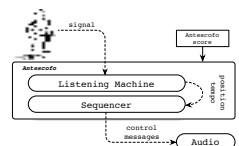


Figure 1. Architecture of the Antescofo system. Continuous arrow represent pre-treatment, and dotted ones real-time communications

Pierre Boulez, Philippe Manoury, Marco Stroppa, New York Philharmonics, Berlin Philharmonics, and the Radio France Orchestra. Figure 1 describes the organization of the Antescofo system. It is composed of two distinct subsystems: a listening machine and a sequencer. During a performance, the *listening machine* estimates the tempo (i.e., execution speed) and the position of the live performers in the score. The role of the *sequencer* is to use these information to trigger electronic actions by sending control messages to a music programming environment, e.g., Max/MSP[‡]. Then, the environment uses these messages to handle complex sound synthesis, manage lights, etc.

The language of Antescofo [9] is a descriptive language inspired by classical western music notation: a score is an interleaving of instrumental notes and electronic actions. An original aspect of this language is that it specifies synchronization and error-handling strategies for the electronic actions. It means that during the performance, depending on the tempo variations and the errors made by the musician or the listening machine, the sequencer will react in a way which is defined by the composer.

The sequencer of Antescofo is a *synchronous* reactive system: it continuously reads inputs from the listening machine, produces outputs to the musical environment and is subject to the strong timing requirement related to ear tolerance (typically 30ms). Synchronous languages [2] have been invented and are used widely[§] for programming a wide range of critical control software: on board control of trains and truck interlocking, fly-by-wire and cockpit displays for planes, etc. Therefore, they would have been well suited for programming the sequencer. But, to ensure statically that programs execute in bounded time and memory, synchronous lan-

^{*}<http://pegusa.srca.fr/antescofo>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or fee. Request permissions from permissions.acm.org.

PARM '13, September 28, 2013, Boston, MA, USA.
Copyright © 2013 ACM 978-1-4503-2386-4/13/09...\$15.00.
<http://dx.doi.org/10.1145/2505341.2505344>

[‡]<http://cycling74.com/>
[§]<http://www.estavel-technologies.com>

Reactivity of Cooperative Systems Application to ReactiveML

Louis Mandel^{1,3} and Cédric Pasteur^{2,3}

² DI École

Abstract.
mentations
facilities as
languages,
ensure the
We present
abstracting
Our objecti
analysis are

Time Refinement in a Functional Synchronous Language

Louis Mandel
LRI, Université Paris-Sud 11, Orsay, France
INRIA Paris-Rocquencourt, France
louis.mandel@lri.fr

Cédric Pasteur Marc Pouzet
DI, École normale supérieure, Paris, France
INRIA Paris-Rocquencourt, France
firstname.lastname@ens.fr

Science of Computer Programming 111 (2015) 190–211

Science of Computer Programming

www.elsevier.com/locate/scico

Time refinement in a functional synchronous language

Louis Mandel^{a,*}, Cédric Pasteur^{b,*}, Marc Pouzet^{d,b,c}

^a IBM Research, Yorktown Heights, NY, USA
^b DI, École normale supérieure, Paris, France
^c INRIA Paris-Rocquencourt, France
^d Université Pierre et Marie Curie, Paris, France

ARTICLE INFO

Article history:
Received 12 December 2013
Received in revised form 22 June 2015
Accepted 2 July 2015
Available online 10 July 2015

Keywords:
Synchronous languages
Functional languages
Semantics
Type systems

ABSTRACT

Concurrent and reactive systems often exhibit multiple time scales. This situation occurs, for instance, in the discrete simulation of a sensor network where the time scale at which agents communicate is very different from the time scale used to model the internals of an agent. The paper presents *reactive domains* to simplify the programming of such systems. Reactive domains allow for several time scales to be defined and they enable time refinement, that is, the replacement of a system with a more detailed version, without changing its observed behavior. Our work applies to the REACTIVEML language, which extends an ML language with synchronous programming constructs à la Esterel. We present an operational semantics for the extended language, a type system that ensures the soundness of programs, and a sequential implementation. We discuss how reactive domains can be used in a parallel implementation.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The concept of logical time greatly simplifies the programming of concurrent and reactive systems. It is the basis of synchronous languages [1] like ESTEREL [2]. Its principle is to see the execution of a reactive system as a sequence of reactions, called *instants*, where all communications and computations are considered to be instantaneous during one reaction. This interpretation of time is logical because it does not account for exact computation time and the precise way all the computations are done during a reaction. It has been originally introduced for programming real-time embedded controllers, but it is applicable for a wider range of applications, in particular large scale simulations.

Consider, for example, the simulation of the power consumption in a sensor network [3]. In order to precisely estimate the power consumption, we need to simulate the hardware of certain nodes, in particular the radio. There are now multiple time scales: for example, the time scale of the software (i.e., MAC protocol) is in milliseconds, while the time step of the hardware would be in microseconds. The communication between these time scales must be restricted. E.g., a slow process, whose time scale is in milliseconds, cannot observe all the changes of a faster process, whose scale is in microseconds. Said differently, a signal that is produced by a fast process cannot be used to communicate a value with a slower process. Furthermore, depending on the level of precision required for the simulation, it makes sense to be able to replace a precise

* Corresponding authors.

E-mail addresses: lmandel@us.ibm.com (L. Mandel), cedric.pasteur@inria.fr (C. Pasteur), marc.pouzet@ens.fr (M. Pouzet).

[†] Now at ANSYS-Estrel Technologies, Toulouse, France.

<http://dx.doi.org/10.1016/j.scico.2015.07.002>

0167-6423/© 2015 Elsevier B.V. All rights reserved.

Extensions

- Reactivity Analysis
- Time Refinement

Application: Mixed Music

ReactiveML, a Reactive Extension to ML^{*}

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6[†]

ABSTRACT

We present REACTIVEML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the *reactive* model introduced by Boussinot. This model combines the so-called *synchronous* model found in ESTEREL which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics à la Esterel and a transition semantics describing precisely the interaction between ML values and reactive processes. It is statically typed through a Milner type theory. Systems and programs are compiled into regular ML programs. The language has been used for programming event complex simulation problems (e.g., routing problems in mobile ad-hoc networks).

1. INTRODUCTION

Synchronous programming [4] has been introduced in the 80's as a way to design and implement time systems. It is based on the idea where communications and computation be instantaneous. In this model, time is the sequence of reactions of the system. The main consequence of this model is the absence of waiting for a modular description. Moreover, techniques of parallelism to be statically compiled are translated into purely sequential programs.

^{*}Laboratoire d'Informatique de Paris 6, Marie Curie, 8 rue du Capitaine Scott, 75013 Paris, France.
email: {Louis.Mandel, Marc.Pouzet}@lip6.fr
[†]This work is supported by the French ANR-08-0001-01.

in terms of transition systems [5, 14].

Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90's, Boussinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex *causality analysis* to statically reject inconsistent programs. This model was called the *synchronous reactive* model (or simply *reactive*) and identified inside SL [11], a new change reactive calculus derived from ESTEREL. Later on, the JUNIOR [15] calculus was introduced as a way to give semantics to the SUGARCUBES [13], this last one being an embedding of the reactive model inside JAVA. This model has been used successfully for the implementation of complex interactive systems as found in graphics user interfaces, video-games or simulation problems [12, 13, 2] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the re-

A Synchronous Embedding of Antescofo, a Domain-Specific Language for Interactive Mixed Music

Guillaume Baudart^{*} Florent Jacquemard[†]
Louis Mandel[‡] Marc Pouzet[§]

Programming Mixed Music in ReactiveML

Guillaume Baudart
École normale supérieure de Cachan
Antenne de Bretagne
DI, École normale supérieure
Guillaume.Baudart@ens-cachan.org

Louis Mandel
Univ. Paris-Sud 11
DI, École normale supérieure
INRIA Paris-Rocquencourt
Louis.Mandel@inria.fr

Marc Pouzet
Univ. Pierre et Marie Curie
DI, École normale supérieure
INRIA Paris-Rocquencourt
Marc.Pouzet@ens.fr

Abstract

Mixed music is about live musicians interacting with electronic parts which are controlled by a computer during the performance. It allows composers to use and combine traditional instruments with complex synthesized sounds and other electronic devices. There are several languages dedicated to the writing of mixed music scores. Among them, the Antescofo language coupled with an advanced score follower allows a composer to manage the reactive aspects of musical performances: how electronic parts interact with a musician. However these domain specific languages do not offer the expressiveness of functional programming.

We embed the Antescofo language in a reactive functional programming language, ReactiveML. This approach offers to the composer recursion, higher order, inductive types, as well as a simple way to program complex reactive behaviors thanks to the synchronous model of concurrency on which ReactiveML is built. This article presents how to program mixed music in ReactiveML through several examples.

Categories and Subject Descriptors: D.3.2 [Language Classification]: Concurrent, distributed, parallel languages; H.5.5 [Information Interfaces and Presentation]: Sound and Music Computing.

Keywords: Synchronous Programming; Language Embedding; Mixed Music; Live Coding.

1. Introduction

Technical progresses since the 1950's have led composers to write music mixing live performers with electronic parts. At first, performers simply followed a pre-registered magnetic band. But the advances in computing rapidly allowed real interaction between musicians and electronic parts.

Developed at IRCAM, Antescofo [5][†] is a state-of-the-art score following system dedicated to mixed music. Since 2008, Antescofo has been used in the creation of more than 40 original mixed electronic pieces by world renowned artists and ensembles, including

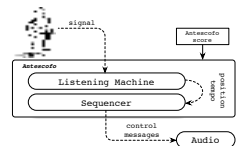
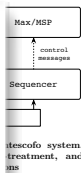


Figure 1. Architecture of the Antescofo system. Continuous arrow represents pre-treatment, and dotted ones real-time communications

Pierre Boulez, Philippe Manoury, Marco Stroppa, New York Philharmonics, Berlin Philharmonics, and the Radio France Orchestra. Figure 1 describes the organization of the Antescofo system. It is composed of two distinct subsystems: a listening machine and a sequencer. During a performance, the *listening machine* estimates the *tempo* (i.e., execution speed) and the position of the live performers in the score. The role of the *sequencer* is to use these information to trigger electronic actions by sending control messages to a music programming environment, e.g., Max/MSP[‡]. Then, the environment uses these messages to handle complex sound synthesis, manage lights, etc.

The language of Antescofo [9] is a descriptive language inspired by classical western music notation: a score is an interleaving of instrumental notes and electronic actions. An original aspect of this language is that it specifies synchronization and error-handling strategies for the electronic actions. It means that during the performance, depending on the tempo variations and the errors made by the musician or the listening machine, the sequencer will react in a way which is defined by the composer.

The sequencer of Antescofo is a *synchronous* reactive system: it continuously reads inputs from the listening machine, produces outputs to the musical environment and is subject to the strong timing requirement related to our tolerance (typically 30ms). Synchronous languages [2] have been invented and are used widely[§] for programming a wide range of critical control software: on board control of trains and track interlocking, fly-by-wire and cockpit displays for planes, etc. Therefore, they would have been well suited for programming the sequencer. But, to ensure statically that programs execute in bounded time and memory, synchronous lan-



Embedding: Error
Mixed Music.

a growing interest parts and live musicians and industry have musicians and composers the development musical interaction, lled from the composer Philippe user-Collier [12], tem and dedicated se, synchronization computer music, or interacting with 'ing this system, mental and electrical in the creation e given by world ing Pierre Boulez, York Philharmon-France Orchestra. of the system. The listening machine (the listening ma-

Reactivity of Cooperative Systems Application to ReactiveML

Louis Mandel^{1,3} and Cédric Pasteur^{2,3}

² DI École

Abstract.
mentations
facilities as
languages,
ensure the
We present
abstracting
Our objecti
analysis are

Time Refinement in a Functional Synchronous Language

Louis Mandel
LRI, Université Paris-Sud 11, Orsay, France
INRIA Paris-Rocquencourt, France
louis.mandel@lri.fr

Cédric Pasteur Marc Pouzet
DI, École normale supérieure, Paris, France
INRIA Paris-Rocquencourt, France
firstname.lastname@ens.fr

Science of Computer Programming 111 (2015) 190–211

Science of Computer Programming

www.elsevier.com/locate/scico

Time refinement in a functional synchronous language

Louis Mandel^{a,*}, Cédric Pasteur^{b,*}, Marc Pouzet^{d,b,c}

^a IBM Research, Yorktown Heights, NY, USA
^b DI, École normale supérieure, Paris, France
^c INRIA Paris-Rocquencourt, France
^d Université Pierre et Marie Curie, Paris, France

ARTICLE INFO

Article history:
Received 12 December 2013
Received in revised form 22 June 2015
Accepted 2 July 2015
Available online 10 July 2015

Keywords:
Synchronous languages
Functional languages
Semantics
Type systems

ABSTRACT

Concurrent and reactive systems often exhibit, for instance, in the discrete simulation of a system, agents communicate is very different from that of an agent. The paper presents reactive domains to simplify domains allow for several time scales to be defined. Our work applies to the REACTIVEML language, a synchronous programming constructs à la Esterel for the extended language, a type system for a sequential implementation. We discuss how implementation.

1. Introduction

The concept of logical time greatly simplifies the programming of concurrent systems. Its principle is to see the execution of a system as a sequence of logical time steps, where all communications and computations are considered to be instantaneous. This is logical because it does not account for exact computation times are done during a reaction. It has been originally introduced for program interpretation for a wider range of applications, in particular large scale simulation. Consider, for example, the simulation of the power consumption in a sensor network. The power consumption, we need to simulate the hardware of certain nodes, in particular time scales: for example, the time scale of the software (i.e., MAC protocol) is in hardware would be in microseconds. The communication between these time scale whose time scale is in milliseconds, cannot observe all the changes of a faster process. Said differently, a signal that is produced by a fast process cannot be used to control a slower process. Furthermore, depending on the level of precision required for the simulation, it may be necessary to use different time scales.

^{*} Corresponding authors.
E-mail addresses: lmandel@us.ibm.com (L. Mandel), cedric.pasteur@ansys.com (C. Pasteur), marc.pouzet@ansys.com (M. Pouzet).
[†] Now at ANSYS-Esterel Technologies, Toulouse, France.
http://dx.doi.org/10.1016/j.scico.2015.07.002
0167-6423/© 2015 Elsevier B.V. All rights reserved.

Louis Mandel
IBM Research
lmandel@us.ibm.com

Cédric Pasteur
École normale supérieure
(now at ANSYS-Esterel Technologies)
cedric.pasteur@ansys.com

Marc Pouzet
École normale supérieure
marc.pouzet@ens.fr

Abstract

Ten years ago we introduced ReactiveML, an extension of a strict ML language with synchronous parallelism à la Esterel to program reactive applications. Our purpose was to demonstrate that synchronous language principles, originally invented and used for critical real-time control software, would integrate well with ML and prove useful in a wider context: reactive applications with complex data structures and sequential algorithms, organized as a dynamically evolving set of tightly synchronized parallel tasks.

While all ReactiveML programs presented at PPDP'05 still compile, the language has evolved continuously to incorporate novel programming constructs, compilation techniques and dedicated static analyses. ReactiveML has been used for applications that we never anticipated: the simulation of large-scale ad-hoc and sensor networks, an interactive debugger, and interactive mixed music. These applications were only possible due to the efficient compilation of ReactiveML into sequential code, which we present here for the first time. We also present a parallel implementation that uses work-stealing techniques through shared memory. Finally, we give a retrospective view on ReactiveML over the past ten years.

1. Introduction

All general purpose programming languages propose a notion of parallel composition and synchronization. It is essential to program applications where several tasks evolve concurrently and communicate. If we consider OCaml, for example, there is the preemptive threads module Thread of the standard library,¹ cooperative threads libraries like Lwt,² Async,³ and Muthreads.⁴ There is also a library of Functional Reactive Programming (FRP),⁵ and a library for event driven programming.⁶

¹ <http://caml.inria.fr/pub/docs/manual-ocaml-4.02/11threads.html>
² <http://ocaml.org/lwt>
³ <https://ocaml.janestreet.com/ocaml-core/11.28.00/doc/async/>
⁴ <http://christophe.deleuze.free.fr/muthreads>
⁵ <http://erratique.ch/software/react>
⁶ <http://www.camcity.org/archive/programming/equeue.html>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPDP '15, July 14–16, 2015, Siena, Italy.
Copyright © 2015 ACM 978-1-4503-3516-4/15/07...\$15.00.
<http://dx.doi.org/10.1145/2790449.2790509>

Extensions

- Reactivity Analysis
- Time Refinement

ReactiveML, Ten Years Later

A distinctive feature of ReactiveML is to provide a deterministic model of concurrency with rich control structures. ReactiveML programs can await and react simultaneously to several events, compose processes in parallel and modularly suspend or preempted parts of a system.

The concurrency model of ReactiveML is based on that of Esterel [5, 8], a language designed for programming control in safety critical real-time systems (e.g., avionic software [7]). This model of concurrency is the *synchronous model* [3]. It relies on a notion of time defined as a succession of logical instants. In the context of real-time systems, logical time allows software to be programmed without having to worry about physical time and then later to check that the worst case execution time of an instant respects the real-time constraints.

Frédéric Boussinot was the first to see that the synchronous model can be used outside the scope of real-time systems [10, 13]. The notion of logical time is useful even if not bound to physical time. In particular, it allows for a precise and deterministic semantics of concurrency and some expressive control structures. ReactiveML is built on these ideas.

In our 2005 article [28], we introduced the ReactiveML language and defined its formal semantics. In this article, after an illustration of the language on an example (section 2), we explain its compilation (section 3) and the runtime (section 4). The implementation of suspension and preemption is explained in section 5. We present a parallel implementation of the runtime (section 6) and compare our implementation with other related systems (section 7). Finally, we will reflect on our experiences over the last ten years and discuss the choices made in ReactiveML (section 8).

The compiler and examples are available at the address

<http://reactiveml.org>.

2. A Complete Example

In this section we propose a solution to the 2013 ICFP Contest.⁷ This example illustrates the mixing of algorithmic and reactive parts in a ReactiveML program. It also serves to present the language syntax and intuitive semantics.

The purpose of the contest is to guess secret programs written in a small language called VB. Each program to guess is called a problem. It is an ID and some meta-data about the program to guess such as its size or the operators that it uses. The contest organizers control a *game server* that provides the following services: (1) evaluate a secret program on an array of 256 inputs; (2) check contestants solutions. If a contestant submits an incorrect solution, the server provides a counterexample. Contestants have 5 minutes to solve each problem. They are allowed to send at most 5 requests to the game server every 20 seconds. Communication with the game server occurs through a web API.

⁷ <http://icfp2013.cldapp.net>

Application: Mixed Music

The background of the image is a collage of musical notation. It features several staves with treble clefs, containing various musical notes, rests, and accidentals. Interspersed among the notes are several chord symbols: Am, C, D, F, and E. The notation is arranged in a way that suggests a musical score, with some staves appearing to be part of a larger piece. The overall aesthetic is clean and artistic, with a focus on the visual representation of music.

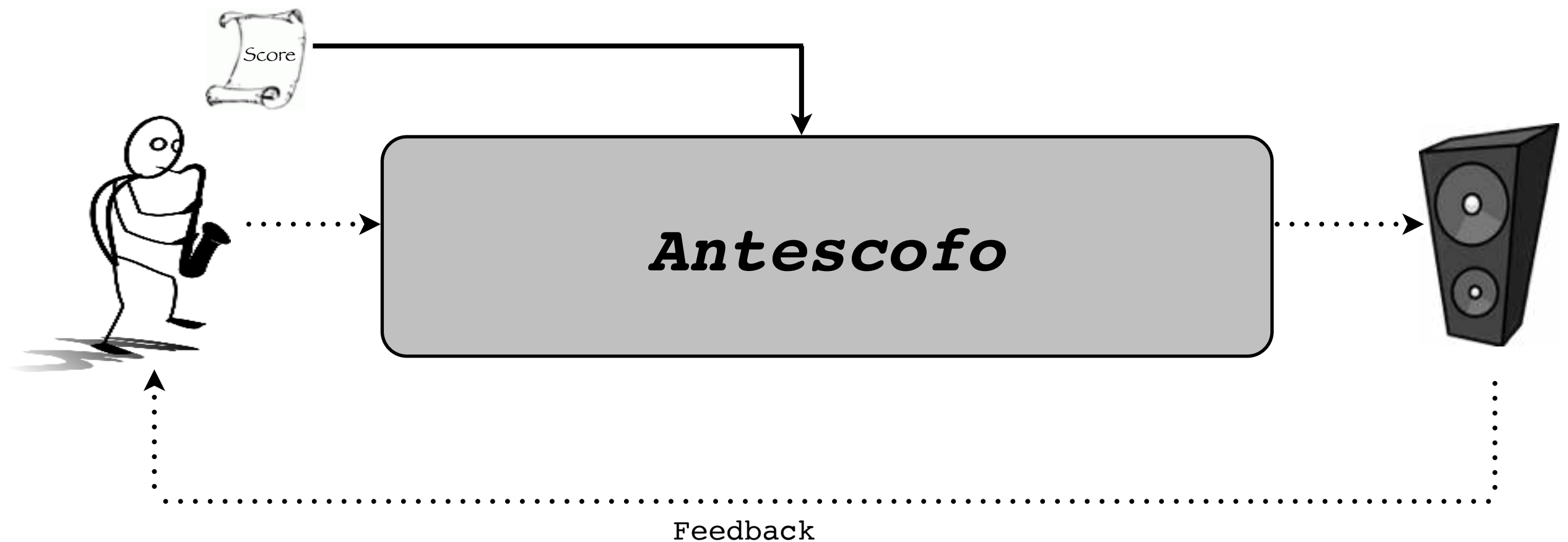
Live Coding with Antescofo

The House of the Rising Sun

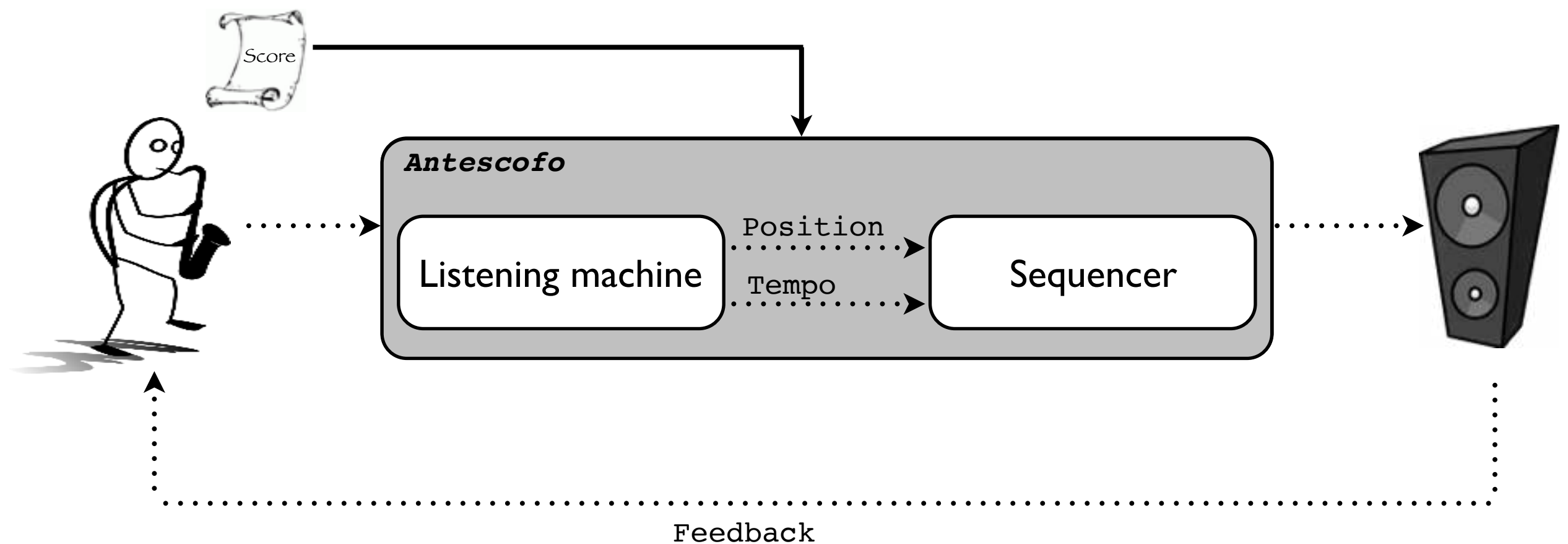
Mixed Music with Antescofo



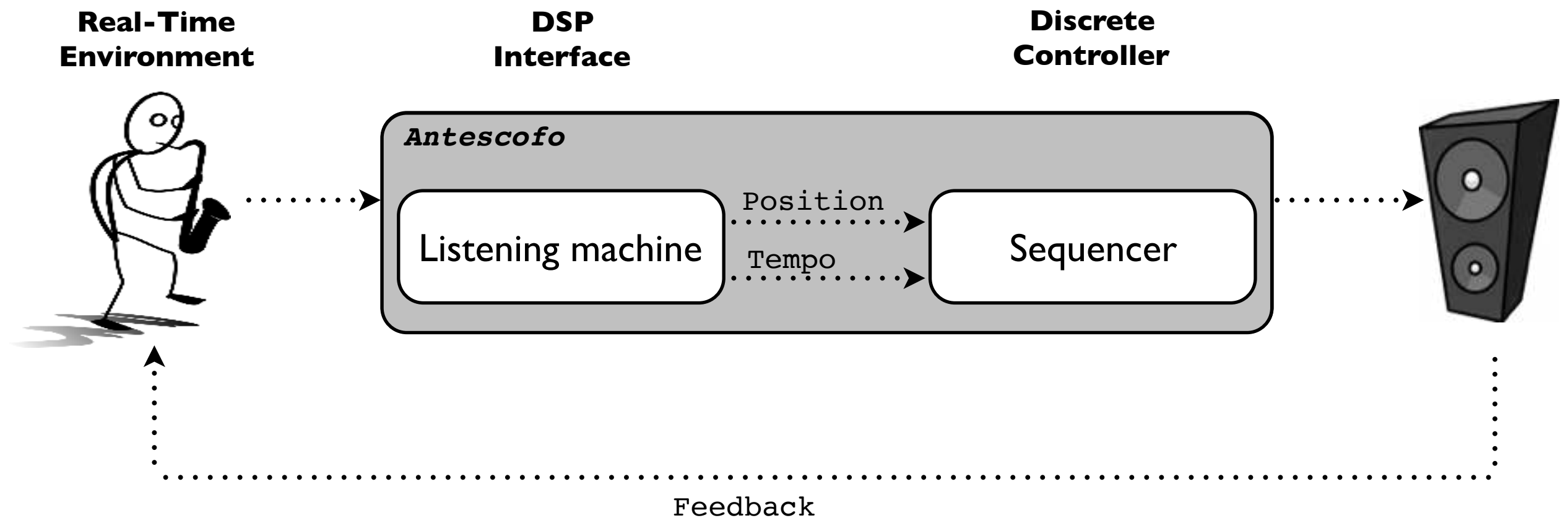
Mixed Music with Antescofo



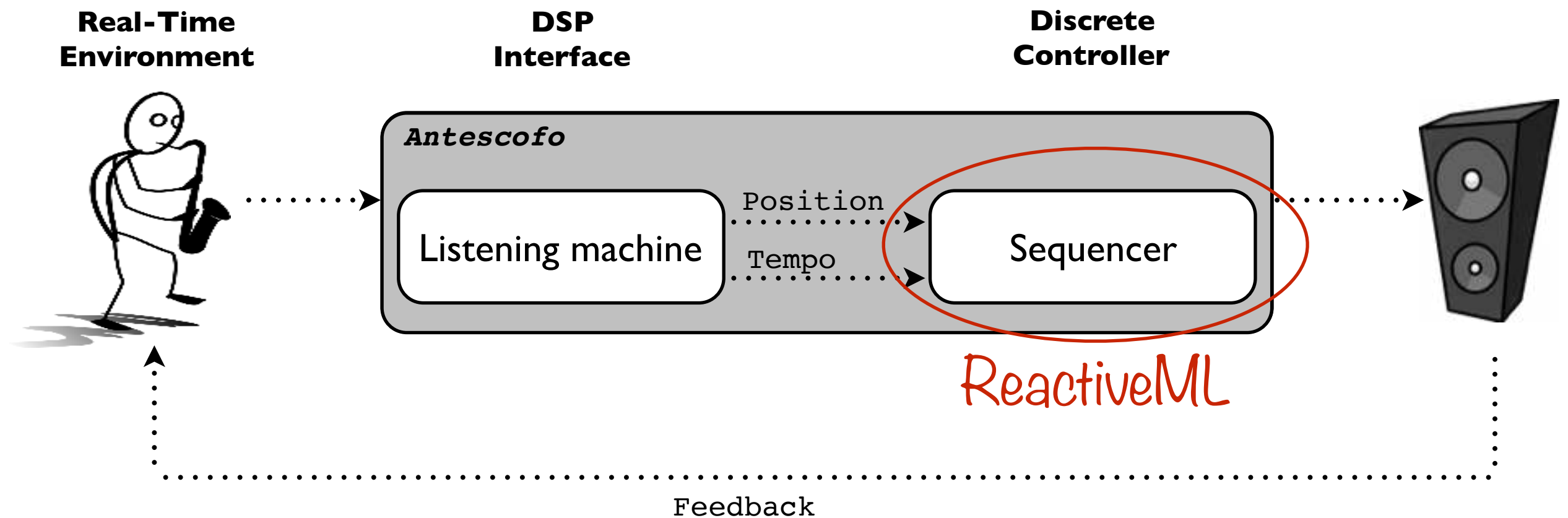
Mixed Music with Antescofo



Mixed Music with Antescofo



Mixed Music with Antescofo

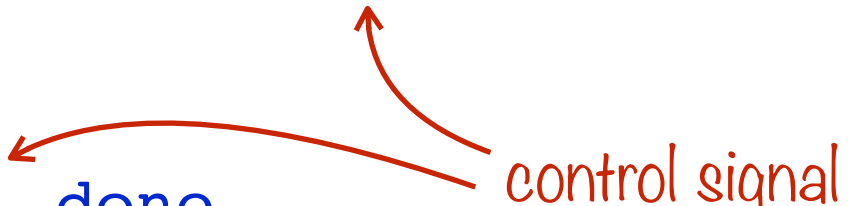


```
let process killable kill p =  
  do  
    run p  
  until kill done
```

val killable: (unit, unit) event -> unit process -> unit process

Higher order processes

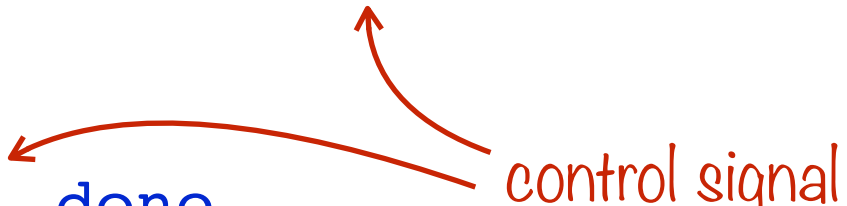
```
let process killable kill p =  
  do  
    run p  
  until kill done
```



A red curved arrow labeled "control signal" points from the `kill` variable in the `until` expression to the `done` argument.

val killable: (unit, unit) event -> unit process -> unit process

```
let process killable kill p =  
  do  
    run p  
  until kill done
```



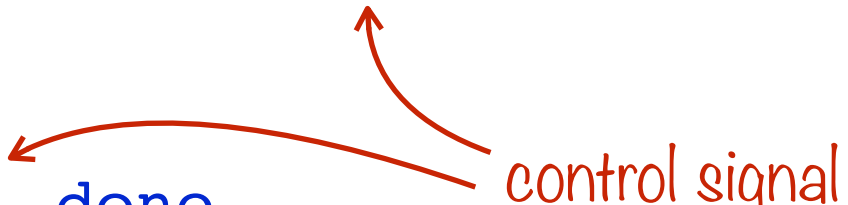
A red curved arrow labeled "control signal" points from the `kill` variable in the `until` clause to the `done` keyword.

val killable: (unit, unit) event -> unit process -> unit process

```
let rec process replaceable replace p =  
  do  
    run p  
  until replace(q) ->  
    run (replaceable replace q)  
  done
```

val replaceable: (unit process, unit process) event -> unit process -> unit process

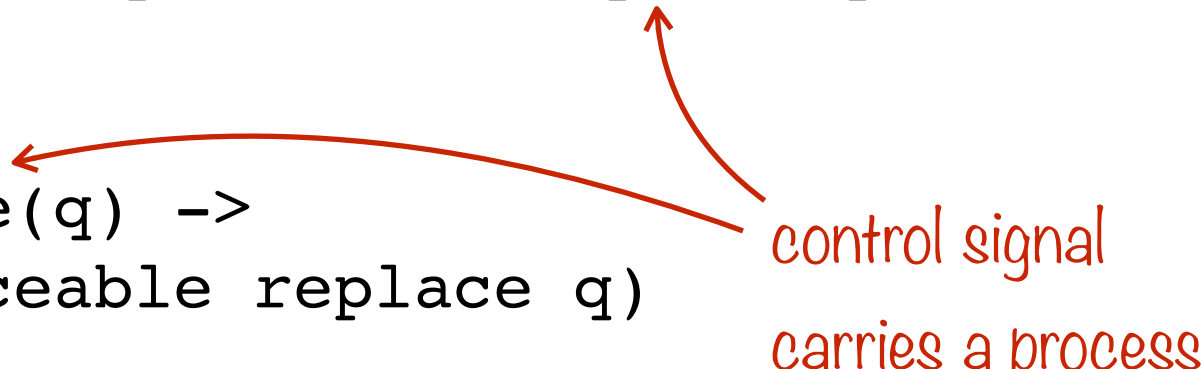
```
let process killable kill p =  
  do  
    run p  
  until kill done
```



A red curved arrow points from the `kill` parameter in the `killable` function signature to the `kill` argument in the `until` expression. The text "control signal" is written in red next to the arrow.

val killable: (unit, unit) event -> unit process -> unit process

```
let rec process replaceable replace p =  
  do  
    run p  
  until replace(q) ->  
    run (replaceable replace q)  
  done
```



A red curved arrow points from the `replace` parameter in the `replaceable` function signature to the `replace(q)` expression in the `until` expression. The text "control signal" and "carries a process" are written in red next to the arrow.

val replaceable: (unit process, unit process) event -> unit process -> unit process



Reactive Domains

n-body Simulation



```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```


Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Pause parametrised
by a domain



Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Pause parametrised
by a domain




Here 3 steps of ck for one of topck

Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Pause parametrised
by a domain



Here 3 steps of ck for one of topck



Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck



Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck



Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck

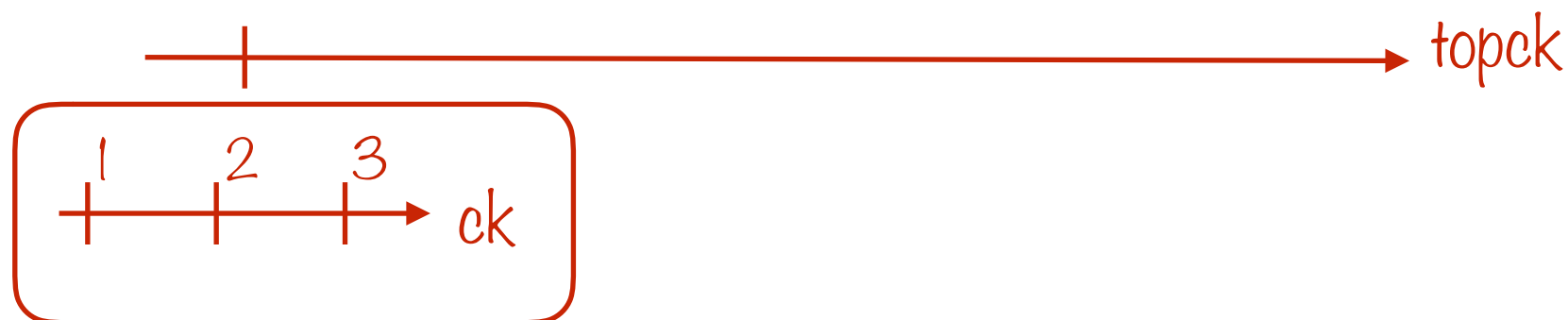


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck



Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Pause parametrised
by a domain



Here 3 steps of ck for one of topck

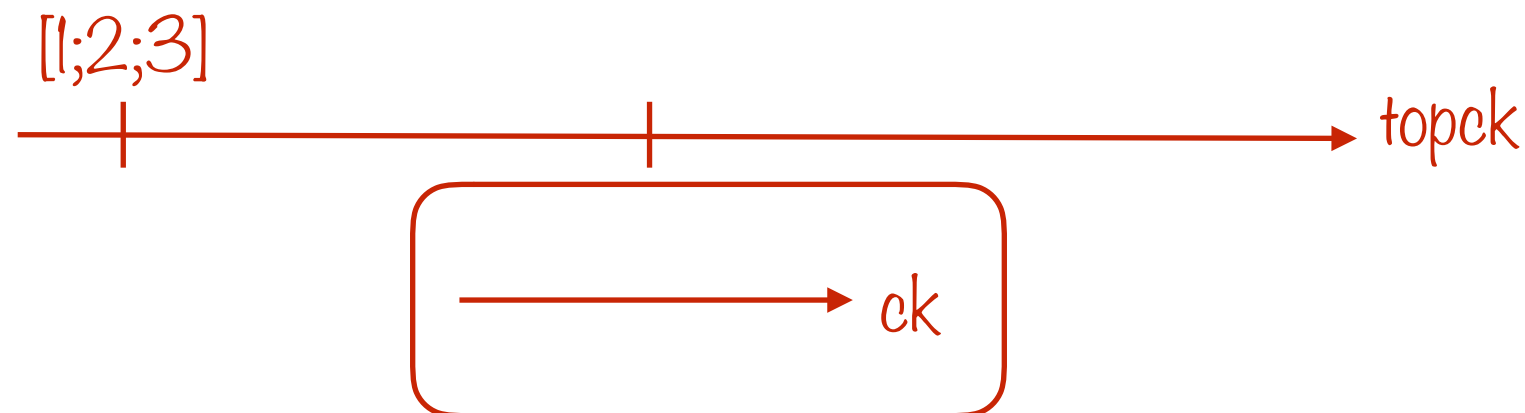


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck

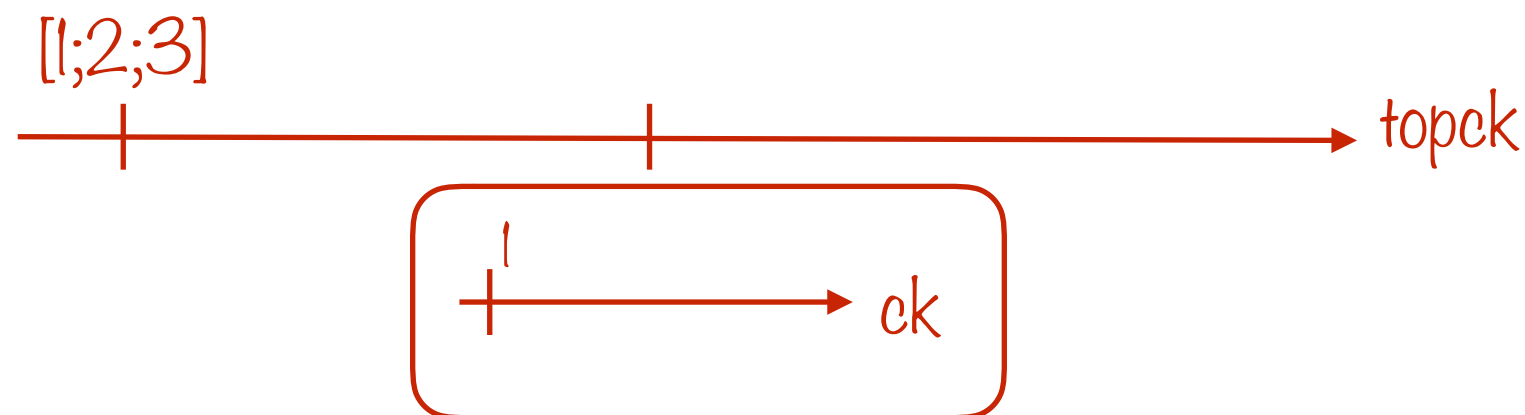


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck

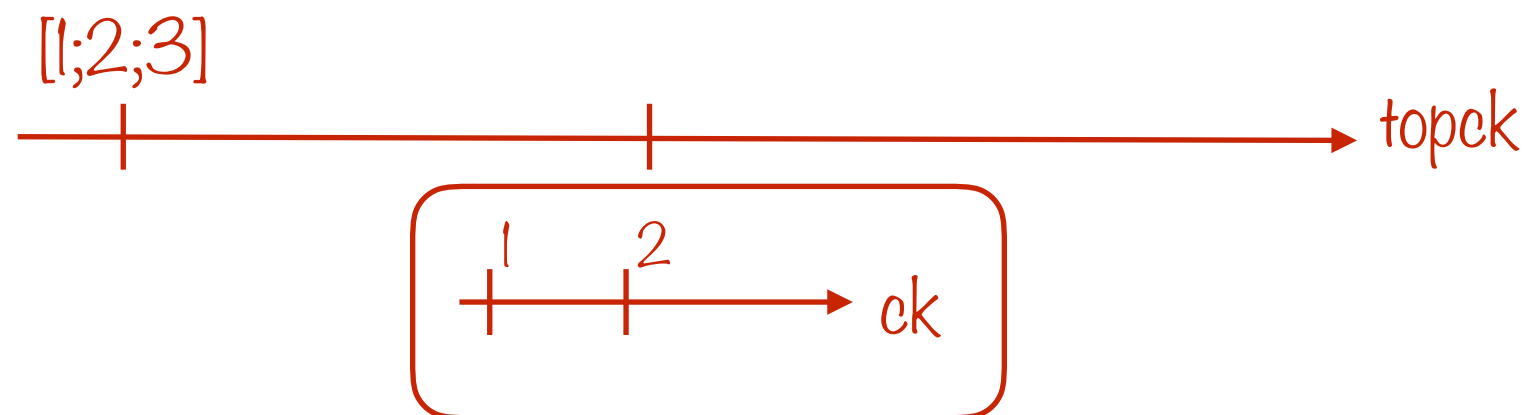


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck

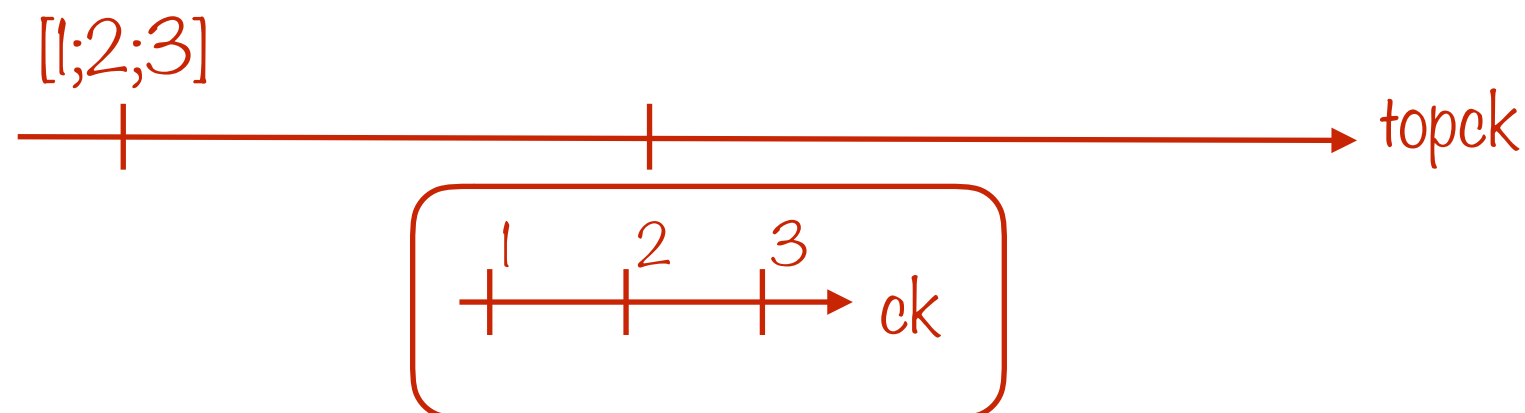


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain


Here 3 steps of ck for one of topck



Idea: hide local time steps

```
let process f s_out =  
  domain ck do  
    loop  
      emit s_out 1; pause ck;  
      emit s_out 2; pause ck;  
      emit s_out 3; pause topck  
    end  
  done
```

Pause parametrised
by a domain



Here 3 steps of ck for one of topck

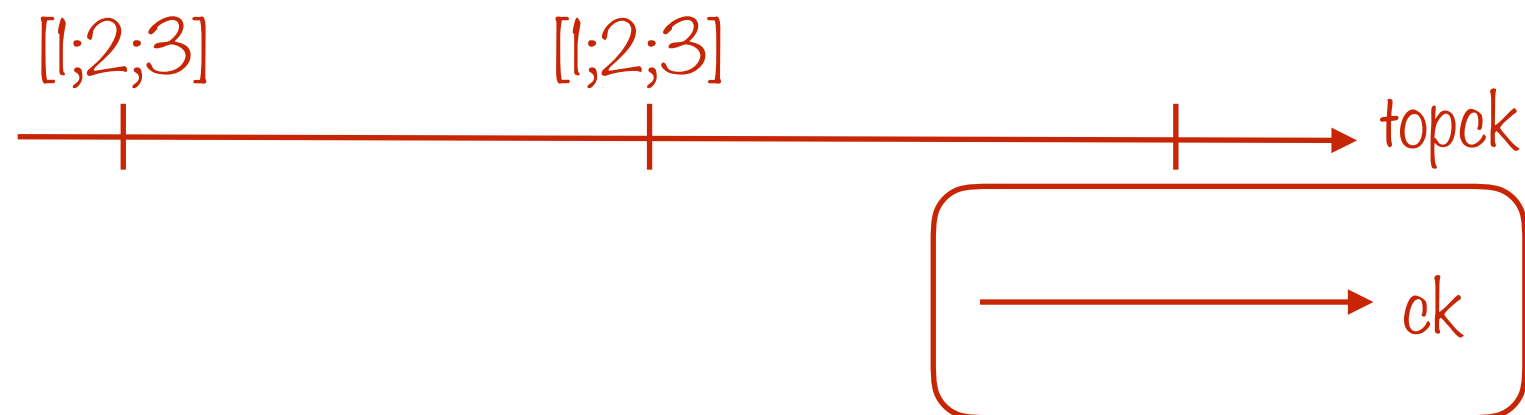


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck

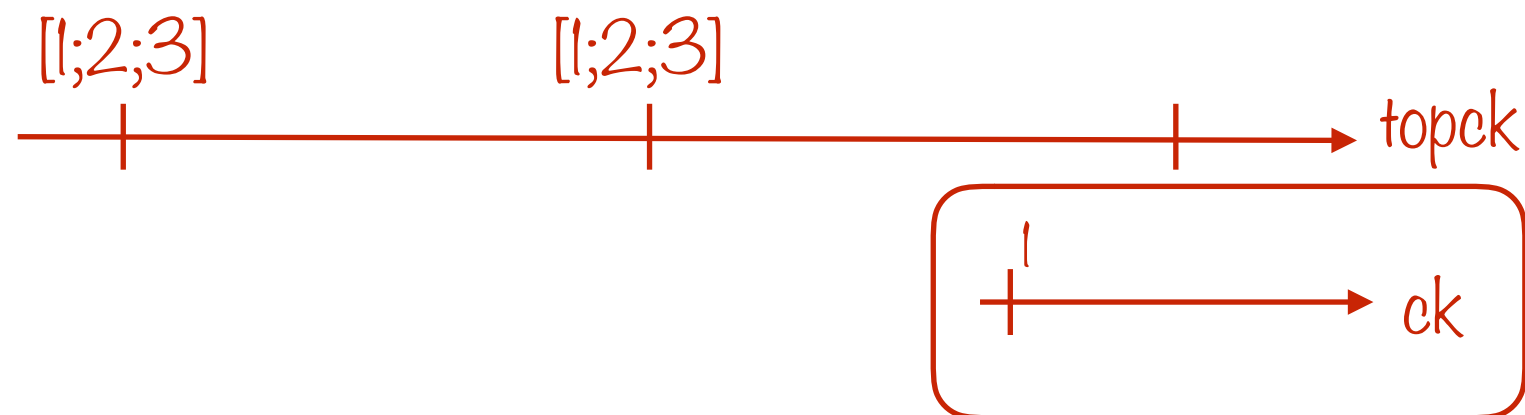


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck

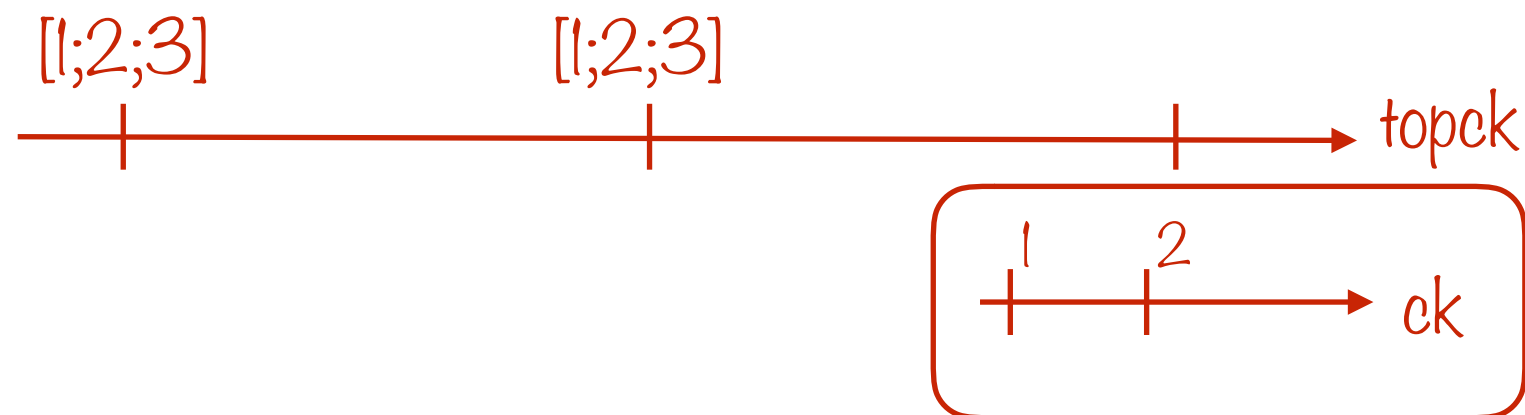


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```

Pause parametrised
by a domain

Here 3 steps of ck for one of topck

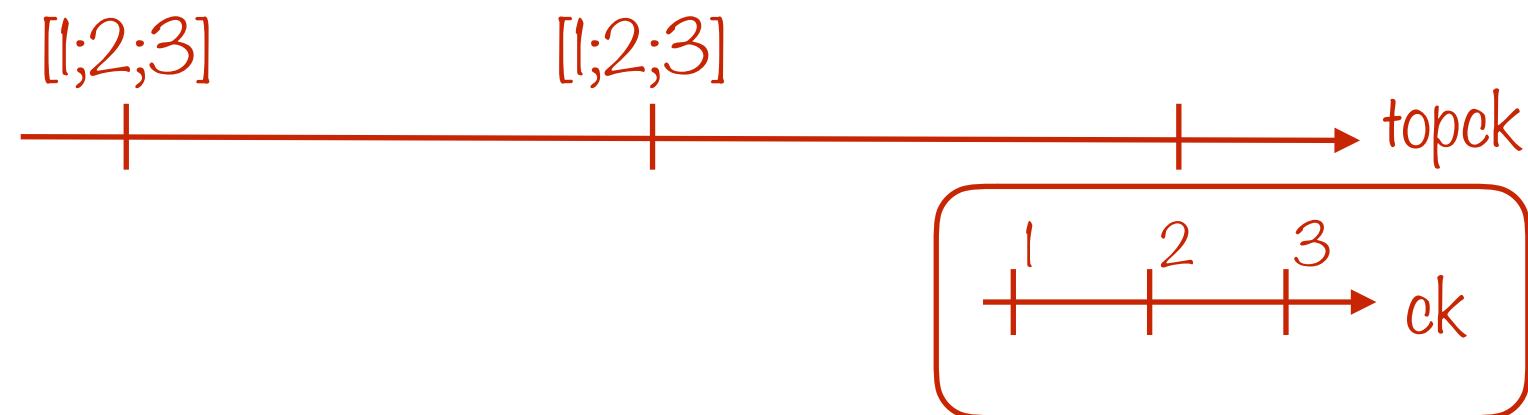


Idea: hide local time steps

```
let process f s_out =
  domain ck do
    loop
      emit s_out 1; pause ck;
      emit s_out 2; pause ck;
      emit s_out 3; pause topck
    end
  done
```


Pause parametrised
by a domain

Here 3 steps of ck for one of topck



```
let process main =  
  ...  
  domain computation_ck do  
    signal env default zero_force gather add_force in  
    for i = 1 to 10 dopar  
      run body env (planet_sprite vp) (random_planet ())  
    done  
    ||  
    run body env (sun_sprite vp) (sun ())  
  done  
  ...
```

```
let process main =  
  ...  
  domain computation_ck do  
    signal env default zero_force gather add_force in  
    for i = 1 to 10 dopar  
      run body env (planet_sprite vp) (random_planet ())  
    done  
    ||  
    run body env (sun_sprite vp) (sun ())  
  done  
  ...
```



```

let process main =
  ...
  domain computation_ck do
    signal env default zero_force gather add_force in
    for i = 1 to 10 dopar
      run body env (planet_sprite vp) (random_planet ())
    done
    ||
    run body env (sun_sprite vp) (sun ())
  done
  ...

```

Hide internal step for integration

```

let process compute_k4 env st =
  (* step 1 *)
  let k1_v = run (compute_a env st.b_pos st.b_weight) in
  let k1_p = st.b_vel in
  (* step 2 *)
  let k2_p = add_v st.b_vel (sc_mult (dt /. 2.0) k1_v) in
  let x_2 = add_v st.b_pos (sc_mult (dt /. 2.0) k1_p) in
  let k2_v = run (compute_a env x_2 st.b_weight) in
  ...

```

Example RK4: 4 steps

<http://reactiveml.org>



Carrés noirs et blancs

After Roger Vilder


A simple integrator
 $x(n+1) = x(n) + \text{delta}$

```
let process sum state delta =  
  loop  
    emit state (last ?state +. delta);  
  pause  
end
```

val sum: (float, float) event -> float -> unit process

A simple integrator
 $x(n+1) = x(n) + \text{delta}$

```
let process sum state delta =  
  loop  
    emit state (last ?state +. delta);  
    pause  
end
```



Discrete-Time step

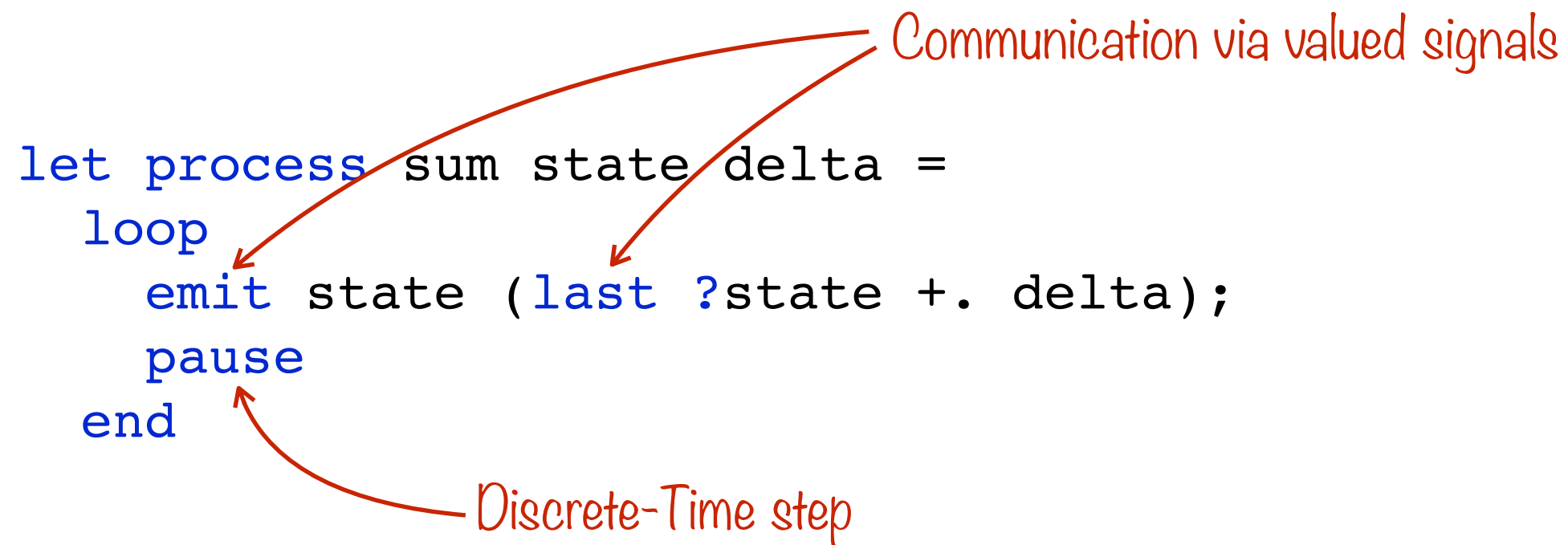
val sum: (float, float) event -> float -> unit process

A simple integrator
 $x(n+1) = x(n) + \text{delta}$

Communication via valued signals

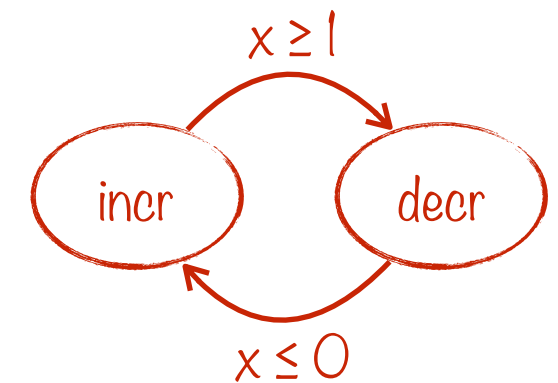
```
let process sum state delta =  
  loop  
    emit state (last ?state +. delta);  
  pause  
end
```

Discrete-Time step



val sum: (float, float) event -> float -> unit process

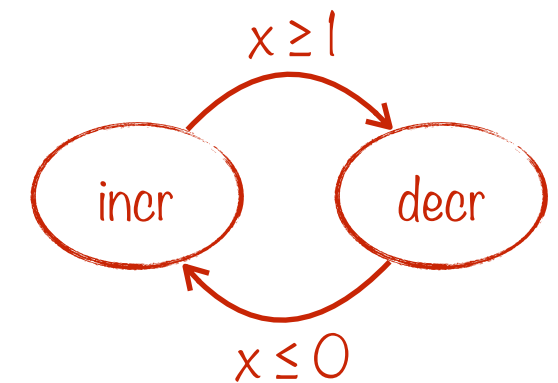
Two states automaton



```
let process incr_decr state delta =  
  let rec process incr =  
    do run sum state delta  
    until state(x) when x >= 1. -> run decr done  
  
  and process decr =  
    do run sum state (-. delta)  
    until state(x) when x <= 0. -> run incr done  
  
  in run incr
```

```
val incr_decr: (float, float) event -> float -> unit process
```

Two states automaton

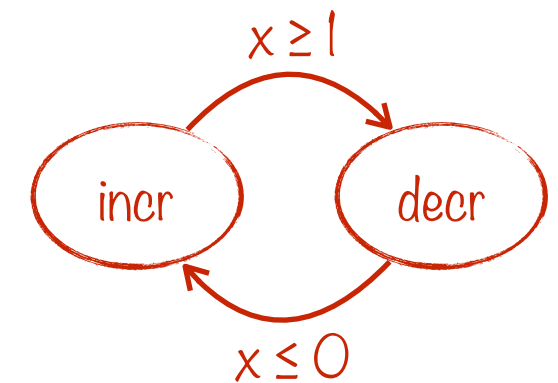


Two mutually recursive processes

```
let process incr_decr state delta =  
  let rec process incr =  
    do run sum state delta  
    until state(x) when x >= 1. -> run decr done  
  and process decr =  
    do run sum state (-. delta)  
    until state(x) when x <= 0. -> run incr done  
in run incr
```

val incr_decr: (float, float) event -> float -> unit process

Two states automaton



Two mutually recursive processes

```
let process incr_decr state delta =  
  let rec process incr =  
    do run sum state delta  
    until state(x) when x >= 1. -> run decr done  
  and process decr =  
    do run sum state (-. delta)  
    until state(x) when x <= 0. -> run incr done  
  in run incr
```

Preemption: do ... until

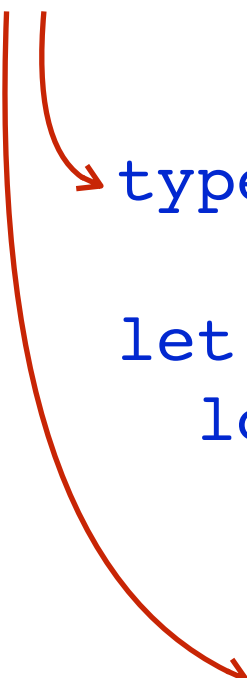
```
val incr_decr: (float, float) event -> float -> unit process
```

```
type dir = Up | Down | Left | Right

let process draw dir x y size state =
  loop
    await state(p) in
    begin match dir with
      | Up -> Graphics.fill_rect x y size (size *: p)
      | ...
    end
  end
end
```

val draw: dir -> int -> int -> int -> ('a, float) event -> unit process

Types definition and match similar to OCaml



```

type dir = Up | Down | Left | Right

let process draw dir x y size state =
  loop
    await state(p) in
    begin match dir with
      | Up -> Graphics.fill_rect x y size (size *: p)
      | ...
    end
  end
end

```

val draw: dir -> int -> int -> int -> ('a, float) event -> unit process

A simple Pump

```
let rec process pump dir x y size state delta =  
  run incr_decr state delta ||  
  run draw dir x y size state
```

```
val pump: dir -> int -> int -> int -> (float, float) event -> float -> unit process
```

A simple Pump

```
let rec process pump dir x y size state delta =  
  run incr_decr state delta ||  
  run draw dir x y size state
```

Parallel composition




```
val pump: dir -> int -> int -> int -> (float, float) event -> float -> unit process
```

```
let rec process splittable split dir x y size init =  
  signal state default 0. gather (+.) in  
  do  
    emit state init;  
    run pump dir x y size state (random_speed ())  
  until split ->  
    run cell split x y size (last ?state)  
done
```

```
and process cell split x y size init =  
  let half = size/2 in  
  run splittable split Left x y half init ||  
  run splittable split Down (x+half) y half init ||  
  run splittable split Up x (y+half) half init ||  
  run splittable split Right (x+half) (y+half) half init
```

Run the process cell

when the signal split is emitted




```
let rec process splittable split dir x y size init =  
  signal state default 0. gather (+.) in  
  do  
    emit state init;  
    run pump dir x y size state (random_speed ())  
  until split ->  
    run cell split x y size (last ?state)  
  done
```

```
and process cell split x y size init =  
  let half = size/2 in  
  run splittable split Left x y half init ||  
  run splittable split Down (x+half) y half init ||  
  run splittable split Up x (y+half) half init ||  
  run splittable split Right (x+half) (y+half) half init
```

Run the process cell

when the signal split is emitted



```
let rec process splittable split dir x y size init =  
  signal state default 0. gather (+.) in  
  do  
    emit state init;  
    run pump dir x y size state (random_speed ())  
  until split ->  
    run cell split x y size (last ?state)  
done
```

```
and process cell split x y size init =  
  let half = size/2 in  
  run splittable split Left x y half init ||  
  run splittable split Down (x+half) y half init ||  
  run splittable split Up x (y+half) half init ||  
  run splittable split Right (x+half) (y+half) half init
```

Split in 4 half-pumps
in the 4 directions