# The SugarCubes Tool Box

Frédéric BOUSSINOT, Jean-Ferdy SUSINI
INRIA EMP-CMA/Meije
2004 route des lucioles
F-06902 Sophia-Antipolis
fb@sophia.inria.fr, jfsusini@sophia.inria.fr

**Abstract**

The SugarCubes are a set of Java classes used to implement reactive, event based, parallel systems. The SugarCubes can be seen as a low-level basis upon which more complex reactive formalisms can be implemented. They also provide a convenient framework for prototyping experimental extensions to various reactive formalisms. The SugarCubes are freely available on the Web.

**Keywords**: Parallelism, Reactive Programming, Events, Java

## 1 Introduction

In this paper, we consider software systems which are:

- *Event based.* In these systems, events are instantly broadcast. Communicating in thus like in radio transmissions, where emitters send information that is instantaneously received by all receivers. This communication paradigm gives a very modular way of system structuring. For example, adding new receivers to a system is totally transparent for the existing components (which is not the case with other communication mechanisms like mess passing or *rendezvous*).

- *Parallel, but thread-less*. Parallelism is a logical programming construct to implement activities which are supposed to proceed concurrently and not one after the other. Such parallel activities need not to be executed by distinct threads, but instead can be automatically interleaved to get the desired result. This avoids well-known problems related to threads.

- *Reactive*. Reactive systems are systems which continuously interact with their environment [HP]. A natural way of programming these systems is by combining reactive instructions whose semantics are defined by reference to activation/reaction couples identified to instants. The end of the reaction provoked by an activation gives a natural way for determining stable states, where a system is only waiting for the next activation to resume execution. Please note that the existence of stable states is of major importance for agent migration over the network.

The SugarCubes[1] are a set of Java[GJS] classes for implementing these systems. Basically, the SugarCubes add instants, parallelism and broadcast events to Java in accordance with the reactive paradigm of which the description can be found on the Web at the URL `http://www.inria.fr/meije/rc/`.

Presently, two main applications are implemented using the SugarCubes:

- Rsi-Java which is the implementation of Reactive Scripts[BH] on top of Java. Reactive scripts give a very flexible and powerful means to program over the Internet.

- Icobjs (for *icon objects*) which define a new, fully graphical way of programming. A demo based on icobjs is available on the Web at the URL `http://www.inria.fr/meije/rc/WebIcobj/`.

The SugarCubes are freely distributed as a tool box for reactive programming in Java.

In this paper we present the SugarCubes and give most part of the code for each of them. Section 2 describes the main classes of reactive instructions and reactive machines. The basic reactive instructions are given in section 3. Section 4 introduces event and related instructions. Finally, related works are considered in section 5.

[1] Why the name SugarCubes? Because many people like to add some sugar in their Java...

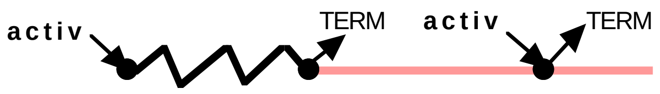
# 2 The Main Classes

The two main SugarCubes classes are `Instruction` and `Machine`. `Instruction` is the class of reactive instructions which are defined with reference to instants, and `Machine` is the class of reactive machines which run reactive instructions and define their execution environments.

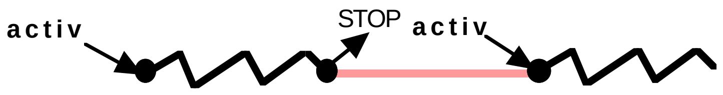
## 2.1 Instructions

An instruction can be activated (`activ` method), reset (`reset` method), or forced to terminate (`terminate` method). Each activation returns TERM, STOP, or SUSP, three return codes defined in the `ReturnCodes` interface:

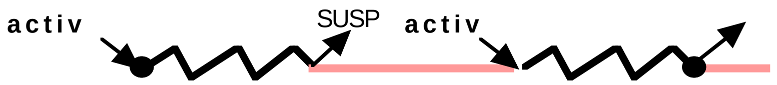
- TERM (for *terminated*) means that execution of the instruction is completed. Thus, activating it at following instants will have no effect and will again return TERM.



- STOP (for *stopped*) means that execution of the instruction is stopped at a stable state for the current instant, and could progress at the next instant upon activation.



- SUSP (for *suspended*) means that execution of the instruction has not reached a stable state for the current instant, and has to resume during that instant. This is for example the case when awaiting for an event not yet generated (see section 4.3): execution suspends to give other components the opportunity to generate it.



A call to the `terminate` method forces the instruction to completely terminate and therefore to return TERM when activated. A call to the `reset` method resets the instruction in its initial state.

`Instruction` implements `Cloneable` and thus instruction clones are available by the `clone` method. Finally, `Instruction` implements the method `toString` to print it. Actually, the printing format is the one of Reactive Scripts.

The `Instruction` class has the following structure:

```
abstract public class Instruction implements ReturnCodes, Cloneable
{
  protected boolean terminated = false;
  public void reset(){ terminated = false; }
  final public void terminate(){ terminated = true; }
  final public boolean isTerminated(){ return terminated; }
  ...
  abstract public String toString();
  abstract protected byte activation(Machine machine);

  final public byte activ(Machine machine)
  {
    if (terminated){ return TERM; }
    byte res = activation(machine);
    if (res == TERM){ terminated = true; }
    return res;
  }
}
```

Note that `activ` calls `activation` which is abstract and must be defined in derived classes. Note also that `activ` and `activation` have a parameter which is the reactive machine running the instruction. Reactive machines are described in section 2.2.

`UnaryInstruction` is an abstract class which extends `Instruction` and has a body which is also an instruction. It is used for example to define loops in section 3.5.

```
abstract public class UnaryInstruction extends Instruction
{
  protected Instruction body;
  public void reset(){ super.reset(); body.reset(); }
  ...
}
```

Similarly, `BinaryInstruction` is an abstract class which extends `Instruction` and has two components `left` and `right` which are also instructions. It is used to define sequence and parallelism in sections 3.2 and 3.3.

## 2.2 Machines

A reactive machine of the `Machine` class runs a program which is an instruction. Initially a program is the `Nothing` instruction (defined in section 3.1) which does nothing and terminates instantaneously. New instructions added to the machine (see below) are put in parallel with the previous program (using `Merge` defined in section 3.3).

Basically, `Machine` detects the end of the current instant, that is when all parallel instructions of the program are terminated or stopped. The behavior is as follows:

- The program is cyclically activated while there are suspended instructions in it (that is, while activations return SUSP).

- The end of the current instant is effective when all the parallel instructions in the program are terminated or stopped (no suspended instruction remains).

- At the end of each program activation, the machine tests if some new events where generated during this execution. If it was not the case, then there is no hope that future program activations will change the situation, and the end of the current instant can be safely decided. Then, a flag is set to let suspended instructions stop, knowing from that point that awaited events are absent.

Two variables `move` and `endOfInstant` are used to implement this behavior (see the method `activation` of `Machine` below):

```
while ((res = program.activ(this)) == SUSP){
   if (move) move = false; else endOfInstant = true;
}
```

Two methods are used to manage the two flags `move` and `endOfInstant`: newMove which sets the `move` flag to indicate that something new happens in the program (thus, the end of instant has to be postponed), and `isEndOfInstant` which tests the `endOfInstant` flag.

The `add` method adds a new instruction to the program; this new instruction is run in parallel with the previous program, using the `Merge` parallel instruction defined in section 3.3:

```
public void add(Instruction inst){
  program = new Merge(program,inst);
  newMove();
```

```
        }
```

Note that `newMove` is called to let the new instruction execute during the current instant.

`Machine` contains an environment named `eventEnv` to deal with events (events are described in section 4).

`Machine` extends `Instruction`, and to execute its program for one instant simply means to activate the machine.

The class `Machine` has the following structure:

```
public class Machine extends Instruction
{
  public Instruction program = new Nothing();
  protected EventEnv eventEnv = new EventEnv();
  protected int instant = 1;
  protected boolean endOfInstant = false, move = false;

  public void newMove()           { move = true; }
  public int currentInstant()     { return instant; }
  public boolean isEndOfInstant() { return endOfInstant; }
  ...
  public void add(Instruction inst){
     program = new Merge(program,inst);
     newMove();
  }

  public Event getEvent(String name){ ... }
  public boolean isGenerated(String name){ ... }
  public void generate(String name){ ... }
  public void putEvent(String name,Event event){ ... }

  protected byte activation(Machine machine){
     ...
    endOfInstant = move = false;
    while ((res = program.activ(this)) == SUSP){
       if (move) move = false; else endOfInstant = true;
    }
    instant++;
    ...
  }
}
```

## 3 Basic Instructions

In this section we introduce the basic instructions to stop and suspend execution for the current instant, the sequence and parallel instructions, atoms to execute Java statements, and loop instructions.

## 3.1 Nothing, Stop, and Suspend

`Nothing` does nothing: it is introduced only as the initial program value:

```
public class Nothing extends Instruction
{
  final public String toString(){ return "nothing"; }
  final protected byte activation(Machine machine){return TERM;}
}
```

`Stop` stops execution for the current instant by returning STOP. However, the instruction terminates, thus activation will return TERM at the next instant:

```
public class Stop extends Instruction
{
  final public String toString(){ return "stop"; }
  final protected byte activation(Machine machine){
    terminate(); return STOP;
  }
}
```

`Suspend` suspends execution for the current instant by returning SUSP. The code is similar to the one of `Stop` except that return code STOP is changed by SUSP.

## 3.2 Sequencing

Class `Seq` extends `BinaryInstruction` and implements sequencing. First, instruction `left` is activated; if it terminates, then control immediately goes to `right`:

```
public class Seq extends BinaryInstruction
{
  public Seq(Instruction left, Instruction right){
     super.left = left; super.right = right;
  }
  final public String toString(){ return left + "; " + right; }

  final protected byte activation(Machine machine)
  {
     if (left.isTerminated()) return right.activ(machine);
     byte res = left.activ(machine);
     if (res != TERM) return res;
     return right.activ(machine);
  }
}
```

## 3.3 Parallelism

Class `Merge` extends `BinaryInstruction` and implements basic parallelism: at each instant the two instructions `left` and then `right` are both activated in this order. It terminates when both `left` and `right` do terminate. The return code of the method `activation` is determined by the following array:

| right \ left | TERM | STOP | SUSP |
|---|---|---|---|
| TERM | TERM | STOP | SUSP |
| STOP | STOP | STOP | SUSP |
| SUSP | SUSP | SUSP | SUSP |

The class `Merge` has the following structure:

```
public class Merge extends BinaryInstruction
{
  private byte leftStatus = SUSP, rightStatus = SUSP;

  public Merge (Instruction left, Instruction right){
     super.left = left; super.right = right;
  }
  public void reset(){
     super.reset(); leftStatus = rightStatus = SUSP;
  }
  final public String toString(){
     return "(" + left + " || " + right + ")";
  }
  final protected byte activation(Machine machine)
  {
     if (leftStatus == SUSP) leftStatus = left.activ(machine);
     if (rightStatus == SUSP) rightStatus = right.activ(machine);
     if (leftStatus == TERM && rightStatus == TERM) return TERM;
     if (leftStatus == SUSP || rightStatus == SUSP) return SUSP;
     leftStatus = rightStatus = SUSP;
     return STOP;
  }
}
```

## 3.4 Atoms

The `Atom` abstract class extends `Instruction` and defines instructions that execute actions which are Java statements and terminate immediately.

```
abstract public class Atom extends Instruction
{
  abstract protected void action(Machine machine);

  final protected byte activation(Machine machine){
      action(machine); return TERM;
  }
}
```

Class `PrintAtom` extends `Atom` to print a string.

```
public final class PrintAtom extends Atom
{
  private String msg;
  public PrintAtom(String msg) { this.msg = msg; }
  final public String toString(){
    return "{System.out.print(\"" + msg + "\");}";
  }
  final protected void action(Machine machine){
    System.out.print(msg);
  }
}
```

## 3.5 Loops

The SugarCubes provide two kinds of loops: infinite loops and finite ones which both extend `UnaryInstruction`.

When the body of an infinite loop of class `Loop` is terminated, it is automatically and immediately restarted.

A loop is said to be *instantaneous* when its body terminates completely in the same instant it is started. Instantaneous loops are to be rejected because they would never converge to a stable state closing the instant. The `Loop` class detects instantaneous loops at run time, when the end of the loop body is reached twice during the same instant. In this case, the loop stops its execution for the current instant instead of looping for ever during the instant.

```
public class Loop extends UnaryInstruction
{
  protected boolean endReached = false;
  public Loop(Instruction body){ super.body = body; }
  final public String toString(){ return "loop " + body + " end"; }
  public void reset(){ super.reset(); endReached = false; }

  final protected byte activation(Machine machine)
  {
    byte res;
    for(;;){
      res = body.activ(machine);
      if (res == TERM){
         if (endReached){
            System.out.println("warning: instantaneous loop detected");
            res = STOP;
            break;
         }
         endReached = true;
         body.reset();
      }else break;
    }
    if (res == STOP){ endReached = false; }
    return res;
  }
```

Note the use of the method `reset` to restart the loop body after termination.

Finite loops of the `Repeat` class execute their body a fixed number of times. Unlike infinite loops, an instantaneously terminating body is not a problem, as it does not prevent the loop to terminate. Therefore, there is no detection of instantaneously terminating bodies of `Repeat` instructions.

## 3.6 An Example

The little example we consider consists of running three instants of a machine. First, one defines the machine and an instruction using `Stop`, `Seq`, `Merge`, and `PrintAtom`; then, the instruction is added to the machine; finally, three machine activations are provoked.

```
class Example
{
  public static void main (String argv[])
  {
    Machine machine = new Machine();

    Instruction inst =
      new Seq(
        new Merge(
          new Seq(new Stop(),new PrintAtom("left ")),
          new PrintAtom("right ")),
        new PrintAtom("end "));

    machine.add(inst);

    for (int i = 1; i<4; i++){
       System.out.print("instant "+i+": ");
       machine.activ(machine);
       System.out.println("");
    }
  }
}
```

Execution of this class gives:

```
instant 1: right
instant 2: left end
instant 3:
```

Note that termination of `Merge` only occurs at the second instant because of the `Stop` instruction in the first branch. Note also that printing of `end` occurs in the second instant: sequencing is instantaneous, that is control goes to the second component of the sequence as soon as the first one terminates.

A call to the method `toString` of instruction `inst` would produce the following Reactive Script, which is a more readable form of the instruction:

```
    (
       stop;{System.out.print("left ")}
    ||
       {System.out.print("right ")}
    );
    {System.out.print("end ")}
```

# 4 Event Programming

Event based programming is achieved in SugarCubes with the `Event` class of events, and with the class `Config` of event configurations which are boolean expressions of events.

## 4.1 Events

The SugarCubes provide the notion of an event with the following characteristics:

- events are automatically reset at the beginning of each instant; thus, events are not persistent data across instants.
- events can be generated by the `generate` method. This determines the event's presence for the current instant. Generating an event which is already present has no effect.
- an event is perceived in the same way by all parallel components: events are broadcast.
- events can be tested for presence, waited for, or used to preempt a reactive statement.
- one cannot decide that an event is absent during the current instant before the end of this instant (this is the only moment one is sure that the event has not been generated during the instant). Thus, reaction to absence is always postponed to the next instant. This is the basic principle of the reactive approach.

To indicate that an event is generated during the current instant, one sets its `generated` field to this instant number. In that way all events are automatically reset at the beginning of each new instant.

The value (present or absent) of an event can only be known safely as soon as it is generated, or otherwise at the end of the current instant. Event presence values are defined in the `EventConsts` interface.

```
public interface EventConsts
{
  final byte PRESENT  = 1;
  final byte ABSENT   = 2;
  final byte UNKNOWN  = 3;
}
```

The `presence` method returns PRESENT if the event is generated, ABSENT if it is not (which is known only at the end of the current instant), and UNKNOWN otherwise. Class `Event` is:

```
public class Event
implements ReturnCodes, EventConsts, Cloneable
{
  public String name;
  private int generated = 0;

  public Event(String name){ this.name = name; }
  public boolean isPresent(Machine machine){
    return generated == machine.currentInstant();
  }

  public void generate(Machine machine){
    generated = machine.currentInstant();
  }
  ...
  public byte presence(Machine machine)
  {
    if (isPresent(machine)) return PRESENT;
    if (machine.isEndOfInstant()) return ABSENT;
    return UNKNOWN;
  }
}
```

## 4.2 Configurations

Event configurations of the class `Config` are boolean expressions of events: a configuration is a simple event (class `PosConfig`), the negation *not* of an event (class `NegConfig`), or the *and* or the *or* of two configurations.

A configuration is said to be *fixed* when its value can be evaluated safely. Abstract class `Config` has the structure:

```
abstract public class Config implements ReturnCodes
{
  abstract public boolean fixed(Machine machine);
  abstract public boolean evaluate(Machine machine);
  ...
}
```

The two classes `PosConfig` and `NegConfig` are abstract classes which extend the class `UnaryConfig` of unary configurations. An unary configuration is fixed when the event method `presence` returns a value different from UNKNOWN.

```
abstract public class UnaryConfig extends Config implements EventConsts
{
  protected String eventName;
  public String name(){ return eventName; }
  ...
  public boolean fixed(Machine machine){
    return (machine.getEvent(eventName)).presence(machine) != UNKNOWN;
  }
}
```

Evaluation of a configuration of type `PosConfig` returns true if the event is generated in the machine. Evaluation of a configuration of type `NegConfig` returns true if the event is not generated in the machine.

Binary configurations are conjunctions (*and*) or disjunctions (*or*) of configurations. The `BinaryConfig` abstract class has two fields `c1` and `c2` of type `Config`.

A conjunction of the class `AndConfig` is fixed as soon as one component is fixed and evaluates to false: the other one does not need to be also fixed. Otherwise, the conjunction is fixed when both components are. Evaluation returns the *and* of the two components:

```
public class AndConfig extends BinaryConfig
{
  public AndConfig(Config c1, Config c2){ this.c1 = c1; this.c2 = c2; }
  public String toString(){ return "(" + c1 + " and " + c2 + ")"; }

  public boolean fixed(Machine machine)
  {
    boolean b1 = c1.fixed(machine);
    boolean b2 = c2.fixed(machine);
    if (b1 && !c1.evaluate(machine)) return true;
    if (b2 && !c2.evaluate(machine)) return true;
    return b1 && b2;
  }

  public boolean evaluate(Machine machine){
    return c1.evaluate(machine) && c2.evaluate(machine);
  }
}
```

The class `OrConfig` of configuration disjunction is not given here as it is very similar to `AndConfig`.

## 4.3 Event Based Instructions

In this section we introduce four instructions related to events: generation of an event, waiting for an event configuration, preemption of an instruction by an event configuration, and local event declaration.

### Event Generation

The `Generate` class extends `Atom` which means that event generation terminates instantaneously. Generating an event in a machine calls the `newMove` method to indicate that something new happens in the system; thus, instructions waiting for the event (see next section) will have the possibility to see it as present during the current instant.

```
public class Generate extends Atom
{
  private String eventName;
  public Generate(String eventName){ this.eventName = eventName; }
  ...
  final public String toString(){ return "generate " + eventName; }

  final protected void action(Machine machine){
    Event event = machine.getEvent(eventName);
    machine.newMove();
    event.generate(machine);
  }
}
```

## Waiting for Events

The `Await` class extends `Instruction` and contains a `Config` field. The `activation` method returns SUSP while the configuration is not fixed, then it evaluates it. If evaluation returns false, meaning that the configuration waited for is not satisfied, then the method stops. If evaluation returns true, meaning that the configuration waited for is satisfied, then `Await` terminates and returns TERM if the end of the current instant is not already reached, STOP otherwise. For example evaluation of the configuration corresponding to “`not e`” returns true if `e` was not generated, and activation returns STOP in this case. This is coherent with the basic principle of 4.1 which states that the absence of an event cannot be decided before the end of the current instant.

```
public class Await extends Instruction
{
  private Config config;

  public Await(Config config){ this.config = config; }
  final public String toString(){ return "await " + config; }
  ...
  final protected byte activation(Machine machine)
  {
    if (!config.fixed(machine)) return SUSP;
    if (!config.evaluate(machine)) return STOP;
    terminate();
    return machine.isEndOfInstant() ? STOP : TERM;
  }
}
```

## Preemption

`Until` extends `BinaryInstruction` and implements preemption: execution of `left`, called the *body*, is aborted when an event configuration becomes true. One says then that `left` is preempted by the configuration and, in this case, control goes to `right` which is called the *handler*.

The preemption implemented by `Until` is a weak one: the body is not prevented to react at the very instant of preemption.

```
public class Until extends BinaryInstruction
{
  private Config config;
  private boolean activeHandle = false;
  private boolean resumeBody = true;

  public Until(Config config,Instruction body,Instruction handler){
    this.config = config; left = body; right = handler;
  }

  public Until(Config config,Instruction body){
    this.config = config; left = body; right = new Nothing();
  }
  ...
  final public String toString(){
    if (right instanceof Nothing) return "do "+left+" until "+config;
    return "do "+left+" until "+config+" actual "+right+" end";
  }

  final protected byte activation(Machine machine)
  {
    if (activeHandle) return right.activ(machine);
    if (resumeBody){ // body is to be executed
      byte res = left.activ(machine); // weak preemption !
      if (res != STOP) return res;
      resumeBody = false;
    }
    if (!config.fixed(machine)) return SUSP;
    if (config.evaluate(machine)){ // actual preemption
      activeHandle = true;
      if (machine.isEndOfInstant()) return STOP;
      return right.activ(machine);
    }
    resumeBody = true; // to re-execute the body at next instant
    return STOP;
  }
}
```

Note that “weakness” of the preemption is coherent with the basic principle of 4.1 which states that the absence of an event cannot be decided before the end of the current instant. On the contrary, *strong* preemption (such as provided by the `watching` primitive of Esterel[BG]) does not let its body execute at all when the preemption condition is true. Thus, execution of the body during one instant needs to decide that the preemption condition is false at that very instant (that is, before its end).

## Event Declaration

`EventDecl` extends `UnaryInstruction` and defines an event which is local to its body. The local event is stored in the field `internal` of the class.

```
public class EventDecl extends UnaryInstruction
{
  private String internalName;
  private Event internal;

  public EventDecl(String internalName,Instruction body)
  {
    this.internalName = internalName;
    internal = new Event(internalName);
    this.body = body;
  }
  ...
  final public String toString(){
    return "event " +  internalName + " in " + body + " end";
  }

  final protected byte activation(Machine machine)
  {
    Event save = machine.getEvent(internalName);
    machine.putEvent(internalName,internal);
    byte res = body.activ(machine);
    machine.putEvent(internalName,save);
    return res;
  }
}
```

## An Example

In this example, one first adds to a machine an instruction which waits for an event named `e` and then prints “`e!`”. The machine is run and a copy of the previous instruction is also added to it. Then, the machine is run for the second time. Finally, an instruction which generates `e` is added, and the machine is run for the third time.

```
class Example1
{
  static int i = 1;
  static void run(Machine machine){
    System.out.print("instant "+(i++)+": ");
    machine.activ(machine);
    System.out.println("");
  }

  public static void main (String argv[])
  {
    Machine machine = new Machine();
    Instruction inst =
       new Seq(new Await(new PosConfig("e")),new PrintAtom("e! "));
    machine.add(inst);
    run(machine);
    machine.add((Instruction)inst.clone());
    run(machine);
    machine.add(new Generate("e"));
    run(machine);
  }
}
```

Execution gives:

```
instant 1:
instant 2:
instant 3: e! e!
```

Note that the two `Await` instructions are both fired during the same third instant when the event is generated (events are broadcast). Note also the use of the method `clone` to get a copy of the instruction.

# 5 Related Works

## Reactive-C and Synchronous Languages

The SugarCubes are actually a derivative of the Reactive-C language [Bo] which extends the language C with reactive instructions. Amongst these instructions are `stop`, `suspend`, `merge`, and `loop` which are also present in the SugarCubes. A program in Reactive-C (the `main` function) correspond to a reactive machine in SugarCubes.

Events are not primitive notions in Reactive-C as they are in SugarCubes. Actually, the notion of an event comes from the language Esterel[BG] which is a member of the synchronous language family[HAL]. Esterel allows immediate reaction to both presence and absence of events (called *signals*). The rejection of the immediate reaction to event absence, in order to avoid Esterel causality problems, is at the basis of the SL synchronous language[BDS]. This rejection is also at the basis of SugarCubes, which adopts SL event based primitives (`generate`, `await`, and `until`).

## Reactive Functional Programming

Reactive programming has been recently implemented in the functional language Standard ML[MTHM] as a reactive library[PUC]. Actually, the central notion is that of a *reactive expression* which is a SML expression which defines instants. The basic reactive primitives include `stop`, `suspend`, and `merge` with a semantics very close to the ones of Reactive-C and SugarCubes. The reactive library provides an opportunity to study the interaction of reactivity with higher-order functions and to implement prototype extensions to existing reactive and synchronous languages.

## Reactive Systems in Java

Several attempts are under work to implement the reactive paradigm in Java.

The Reactive Java formalism[PPLS] defines a collection of Java classes to describe modules and their interconnections, and primitives to handle synchronous and asynchronous communications. By contrast with SugarCubes, Reactive Java uses a separate thread for each module. A separate thread is also used to implement preemption, to look at the watched event and to send an interrupt when it `arrives`.

A language, also named Reactive'Java, is currently developed by the company Soft Mountain on top of Java to implement the reactive paradigm (informations are available on the Web at URL `http://www.soft-mountain.com`).

## Reactive Scripts and Nets of Reactive Processes

Reactive Scripts[BH] are strongly related to SugarCubes. In Reactive Scripts, instants are identified to interpreter reactions. Reactive Scripts are implemented on top of Java using the SugarCubes. Actually, Reactive Scripts can be seen as the interpreted version of SugarCubes. They are described in a paper available on the Web[BS1].

Nets of Reactive Processes are dataflow networks made of parallel processes sharing the same global instants and communicating through FIFO channels. These nets are an extension of Kahn's networks[KMQ] in which the empty channel test becomes possible. They are implemented using SugarCubes and are described in a paper available on the Web[BS2].

# 6 Conclusion

We have presented an overview of SugarCubes to program reactive instructions (`Instruction` class) run in parallel by reactive machines (`Machine` class) and communicating with broadcast events (`Event` class).

The SugarCubes can be seen as a low-level system upon which more complex reactive formalisms, such as Reactive Scripts, can be implemented. They also provide a convenient framework for prototyping experimental extensions to various reactive formalisms.

The SugarCubes (version 1) are freely available on the Web at the following URL `http://www.inria.fr/meije/rc/SugarCubes/`.

## Bibliography

[BG] G. Berry, G. Gonthier, *The Esterel Synchronous Language: Design, Semantics, Implementation,* Science of Computer Programming, 19(2), 1992.

[BH] F. Boussinot, L. Hazard, *Reactive Scripts*, Proc. RTCSA'96, Seoul , IEEE, 1996.

[Bo] F. Boussinot, *Reactive-C: An extension of C to program reactive systems*, Software Practice and Experience, 21(4): 401-428, 1991.

[BDS] F. Boussinot, R. De Simone, *The SL Synchronous Language*, IEEE Trans. Software Engineering, 22(4), 1996.

[BS1] F. Boussinot, J-F. Susini, *The SugarCubes Tool Box - Rsi-Java Implementation*, available on the Web at URL `http://www.inria.fr/meije/rc/ SugarCubes/`.

[BS2] F. Boussinot, J-F. Susini, *The SugarCubes Tool Box - Nets of Reactive Processes Implementation*, available on the Web at URL `http://www.inria.fr/ meije/rc/ SugarCubes/`.

[GJS] J. Gosling, B. Joy, G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[HAL] N. Halbwachs, *Synchronous Programming of Reactive System,* Kluwer Academic Pub., 1993.

[HP] D. Harel, A. Pnueli, *On the Development of Reactive Systems*, NATO ASI Series F, Vol. 13, Springer-Verlag, 1985.

[KMQ] G. Kahn, D. B. MacQueen, *Coroutines and Networks of Parallel Processes*, Proc. IFIP Congress 74, 1977.

[PPLS] C. Passerone, R. Passerone, L. Lavagno, A. Sangiovanni-Vincentelli, *Modelling Reactive Systems in Java*, IEEE International High Level Design Validation and Test Workshop, Oakland,1997.

[PUC] R. R. Pucella, *Reactive Programming in Standard ML*, Report of Bell Laboratories, Lucent Technologies, 1997.

[MTHM] R. Milner, M. Tofte, R. Halper, D. B. MacQueen, *The Definition of Standard ML (Revised)*, The MIT Press, Cambridge, Mass., 1997.