Programming with Junior*

Frédéric Boussinot†  Laurent Hazard‡  Jean-Ferdy Susini§

Draft - May 16, 2001

1 Introduction

The reactive approach proposes a flexible paradigm for programming reactive systems [5], especially those which are dynamic (that is, the number of components and their connections can change during execution). Reactive programming provides programmers with concurrency, broadcast events, and several primitives for gaining fine control over reactive programs executions. At the basis of reactive programming is the notion of a reaction: reactive programs are reacting to activations issued from the external world. Program reactions are often called *instants*. The two main notions are reactive instructions whose semantics refer to instants, and reactive machines whose purpose is to execute reactive instructions in an environment made of instantaneously broadcast events.

*Junior* [6] is a Java-based language for programming reactive behaviors. It is closely related to *SugarCubes* [2] and is actually a descendant of it. Basically, programming with *Junior* means:

- writing a *reactive instruction*, which describes an application program;
- declaring a *reactive machine*, to run the program;
- dropping the program into the machine;

---

*With support from France-Telecom R&D, and from EC (PING IST Project)
†EMP/CMA-INRIA
‡FT-R&D
§EMP/CMA-INRIA
• running the machine; this is usually performed using a non terminating loop, which cyclically makes the machine and the program react.

Programming in Junior has a dynamic aspect: machine programs can be augmented by new reactive instructions added during machine execution. New instructions dropped into a machine do not have to wait for the termination of the presently running program, but are run concurrently with it.

Junior concurrent reactive instructions can communicate using broadcast events that are processed by reactive machines. Broadcasting is a powerful and fully modular means for communication and synchronization of concurrent components. Broadcasting in Junior has a special coherency property: during a machine reaction, the same event cannot be tested both present and absent, even by two distinct concurrent instructions.

Junior defines primitive constructs allowing for code (reactive instruction) migration over the network.

1.1 Jr

Junior is pure Java and is provided with an API named Jr. Using Jr, programmers can define reactive instructions and reactive machines, and have possibility to run them. Junior is a programming language, defining constructs for reactive programming. It can also be seen as a Java programming framework. From this last point of view, Junior provides Java programmers with an alternative to the standard threading mechanism. The benefit is that Junior gives solutions to some well-known problems of Java threads (see [7] for a description of these problems).

Structure of the paper is as follows: section 2 gives an overview of Junior; the Jr API is presented in section 3; finally, section 4 considers several points of programming in Junior. An annex describes two classes which define the programming interface.

2 Junior Overview

This section introduces reactive instructions, reactive machines, and events, and it considers several related points, such as concurrency and migration.

2.1 Reactive Instructions

Reactive instructions are state-based statements, run (one also says, activated) by reactive machines. Some cyclic instructions are never ending across
in instants, while others are reaching a final state after several activations; in this case, one says that they are completely terminated. Because states are embedded in them, reactive instructions are not reentrant: they must be copied, in order to get new execution instances.

Reactive instructions are composed from a small set of basic constructors. For example, the constructor Seq puts two reactive instructions A and B in sequence: A is executed up to complete termination (remember that it may take several machine reactions), and then B is executed. The associated state of Seq encodes termination of the first component: the state changes when the first component completely terminates; then, following executions directly go to the second component, without considering the first one.

Reactive instructions are Java objects implementing the Program interface. They are built using static methods of class Jr\(^1\). For example, to define the sequence of two reactive instruction A, B, one writes:

\[
\text{Program } p1 = \text{Jr.Seq}(A,B);
\]

Programming in Junior leads to a Lisp-like programming style, with a lot of parenthesis, which can be rather boring when writing large-sized reactive behaviors. To avoid this problem, more concise syntax have been proposed, which eventually translate into Junior programs; this point is considered in Deliverable D4.1.4 which presents the Rejo language.

Among the reactive instructions are the ones, called atoms, used to interface with Java. Basically, an atom executes an action which possibly performs some interaction with the Java environment. The action is executed once and the atom immediately terminates after first reaction. Execution of an atom is atomic: once started, execution of an atom always terminates without any interference with other atoms.

One special atom is Print, defined for printing purposes; it is mainly used for tracing execution and it will be very useful in this presentation to get observable behaviors. For example, to print “hello, world!”, one writes in Junior:

\[
\text{Jr.Print("hello, world");}
\]

### 2.2 Reactive Machines

Two constructors are provided by Jr to build machines: Machine and SafeMachine which both implements interface Machine. The two kinds\(^1\)

\(^1\)To simplify, one also calls “constructors” the static methods of Jr which call constructors of Junior classes.
of machines differs in multithreaded contexts; this point will be discussed later, in section 4.7; for the moment, let us consider standard machines.

A reactive instruction can be given at construction; when it is the case, it becomes the initial program of the machine. Reactions of reactive machines (often simply called "machine", for short) are obtained using the react method of interface Machine.

One can now gives a minimalist example of executable Junior code:

```java
import junior.*;

public class HelloWorld
{
    public static void main(String[] args){
        Machine machine = Jr.Machine(Jr.Print("hello, world!"));
        for (int i = 0; i < 3; i++){
            System.out.print("instant "+i+: ");
            machine.react();
            System.out.println(" ");
        }
    }
}
```

Package junior is imported to be able to use classes Jr, Machine, and Print. Then, a standard Java class is defined with a main method. The body of main starts by defining a reactive machine with an initial program which just prints a message. Finally, the reactive machine is activated 3 times (a trace shows the sequence of machine reactions). One obtains the output:

```
instant 0: hello, world!
instant 1:
instant 2:
```

The message is printed at the first instant, that is during the first machine reaction. The two next reactions are empty, as the machine program is completely terminated at the end of first reaction. An other way to define the previous machine would be to drop the program into the machine, using the add method:

```java
Machine machine = Jr.Machine();
machine.add(Jr.Print("hello, world!"));
```

The result would be exactly the same.

The Stop instruction is the basic way to delay execution for the next instant. Executing a Stop terminates execution for the current instant;
however, execution is not completely terminated at this stage, and the Stop instruction becomes the new starting point for the next instant. The state associated to Stop encodes the end of the current instant: it changes at the end of the first instant of execution, indicating that the instruction is completely terminated. Running the previous machine with the program:

```java
Jr.Seq(Jr.Print("hello, "),
       Jr.Seq(Jr.Stop(), Jr.Print("world!")))
```

would produce:

```
instant 0: hello,
instant 1: world!
instant 2:
```

The Stop instruction splits the execution in two distinct instants: "hello," is printed during the first one, while "world!" is printed during the second one.

### 2.3 Dynamicity

Dropping a program into a machine can occur at any time. For example, one can trigger the dropping of an instruction by pressing on a button displayed on screen; the following code is a Java 1.1 "action listener" to perform this:

```java
class Drop implements ActionListener
{
    Machine machine;
    Program program;
    public Drop(Machine machine, Program program)
    {
        this.machine = machine; this.program = program;
    }
    public void actionPerformed(ActionEvent e)
    {
        machine.add(program.copy());
    }
}
```

The copy method of Program produces new fresh copies of reactive instructions. Thus, a new program is dropped each time actionPerformed is called.

Now, here is the code fragment to define a graphical button, which prints a message when pressed:
Machine machine = Jr.Machine();
Button button = new Button("press here");
button.addActionListener(    
    new Jr.Drop(machine,Jr.Print("hello, world\n")));
Frame frame = new Frame();
frame.add(button);

An important point is that without copying the dropped instruction in method actionPerformed, only the first press action would print a message, as reactive instructions are not reentrant; indeed, to drop in a machine an instruction which is already terminated, has no effect.

2.4 Events

Event are non persistent data with a binary status present or absent, possibly changing at each instant. An event becomes present during one instant as soon as it is generated by a program component during this instant. A strong coherency property holds: during one instant, the same event cannot be tested as present by one component and as absent by another component. In other words: events are broadcast.

A way to implement the coherency property of Junior machines is as follows:

- A new unknown event status is introduced; the machine assigns it to all events at the beginning of each instant.
- The status of an event is changed to present as soon as it is generated.
- When the machine detects that no new event can be generated, it changes to absent the status of all unknown events and decides the end of current instant.

Note that end of instant and absence of events are decided together, in the same step; this has important consequence, which will be discussed later in section 4.2.

The Jr API gives several ways to deal with events. In the simplest one, events are identified by strings. For example, to generate an event named e, one writes: Jr.Generate("e") and to wait for it: Jr.Await("e"). In this last instruction, control is stopped while event e is not generated, and the instruction is completely terminated when e becomes present. The Await instruction has an associated state which codes for termination, that is, for the awaited event generation.

The following code illustrates event broadcasting:
Button button1 = new Button("accumulate");
button1.addActionListener(new Drop(machine,
    Jr.Seq(Jr.Await("event"), Jr.Print("hello, world!\n"))));
Button button2 = new Button("flush");
button2.addActionListener(new Drop(machine, Jr.Generate("event")));

So, a new copy of:

    Jr.Seq(Jr.Await("event"), Jr.Print("hello, world!"))

is dropped into the machine each time the first button is pressed. When
the second button is pressed, a new Generate instruction is dropped into
the machine, which triggers the previously dropped waiting instructions.
The broadcast property of events insures that all waiting instructions are
triggered in the same instant: actually, the machine program is empty after
pressing the second button.

2.5 Concurrency

Junior owns a concurrency constructor, named Par (for parallelism) which
puts two reactive instructions A and B in parallel: A and B are executed
each instant, and the parallel construct is completely terminated when
both A and B are. The state of Par is the union of the state of A and of
the state of B. The order in which, at each instant, A and B are executed is
left unspecified. Thus, the result of executing:

    Jr.Par(Print("hello, "), Jr.Print("world!"))

can be as well "hello, world!" or "world!hello, ".

Reactive instructions dropped in a machine are put in parallel with the
machine program. For example:

    machine.add(Jr.Par(A, B))

is strictly equivalent to:

    machine.add(A); machine.add(B)

To simplify programming and reasoning about reactive programs, an
instruction dropped into a machine during the course of a reaction is not
immediately run by the machine; actual adding of the instruction to the
machine program is delayed to the beginning of the next instant.
2.6 Interfacing with Java

Interfacing reactive instructions with Java is based on the notion of an implicit Java object set by a link instruction. The implicit Java object can be directly accessed and transformed by atoms executed by the link body; it is also used by expressions evaluations triggered by reactive instructions and returning values for them.

Implicit Object

The Jr.Link(object, p) reactive instruction defines object as being the implicit Java object associated to the reactive instruction p.

Actions executed by atoms must implement the execute method with signature void execute(Environment env). The environment allows the action to get access to the implicit Java object (if defined), using method linkedObject. For example, consider:

```java
class Incr implements Action {
    public void execute(Environment env) {
        Button b = (Button) env.linkedObject();
        int v = Integer.parseInt(b.getLabel());
        b.setLabel(""+(v+1));
    }
}
```

The implicit object is obtained by the call “env.linkedObject()” which returns an instance of type Object. Actually, one supposes that the implicit object of Incr is always a button of type java.awt.Button; thus, to get access to it, one must cast it to Button. Now, in order to cyclically increment the label of button, it becomes sufficient to add the following instruction in the reactive machine:

```
Jr.Link(button, Jr.Loop(Jr.Seq(Jr.Atom(new Incr()), Jr.Stop())))
```

Expressions

There are several kinds of expressions used to interface reactive instructions with Java. For example, an integer expression is used to define the number of iterations a repeat loop has to perform. A boolean expression used by If reactive instructions is another example; it must implement method with signature public boolean evaluate(Environment env) of class BooleanWrapper. The moment evaluation of an expression takes place
is under control of the reactive instruction using it. For example, here is
the definition of a boolean expression that returns the value of the variable
named automatic of its implicit object:

class Automatic implements BooleanWrapper
{
    public boolean evaluate(Environment env){
        return ((Robot)env.linkedObject()).automatic;
    }
}

2.7 Migration

In Junior, it is possible to extract instructions from machines, getting
what remains to be executed. This is the inverse of the dynamic dropping
of instructions provided by the add method of machines. Actually,
not all instructions can be extracted, but only freezeable ones. Instruction
Jr.Freezable("event",p) defines p as a freezeable instruction which can
be extracted from the executing machine by generating event. The result
of the extraction is a new reactive instruction representing what remains to
be done by p from the point it is frozen, up to complete termination. The
extracted instruction is called the residual of p.

Three points are important:

• Only currently running freezeable instructions can be removed from
machines.

• The removal of a freezeable instruction does not prevent it to react at
the very instant the freezing event occurs.

• The residual obtained from a freezeable instruction is only available at
the instant which immediately follows generation of the freezing event.

Here is, for example, an action which transfers (by calling the static
method Migration.transfer) the residual of a freezeable instruction (freez-
ing event is freeze):

class Migrate implements Action
{
    public void execute(Environment env){
        Program residual = env.getFrozen(Jr.StringIdentifier("freeze"));
        if (residual != null) Migration.transfer(residual);
    }
}
Method `getFrozen` returns the residual associated to an event, if it exists; it returns `null` otherwise.

As a consequence of the previous definition of a freezable instruction, the following actions must be performed in sequence in order to extract a freezable instruction:

1. generate the freezing event;
2. wait for next instant;
3. get the residual associated to the freezing event.

Here is, for example, how to define a button which, when pressed, transfers the residual of a freezable instruction (by running the previous `Migrate` action):

```java
button.addActionListener(
    new Drop(machine,
        Jr.Seq(Jr.Generate("freeze"),
            Jr.Seq(Jr.Stop(), Jr.Atom(new Migrate()))));
)
```

The following method assigns a freezable behavior to a button:

```java
public void setButtonBehavior(Program p){
    machine.add(Jr.Link(button, Jr.Freezable("freeze", p)));
}
```

Suppose that there are two buttons and that a cyclic behavior is initially transferred to one of the two buttons, by calling method `transfer` (which in turn calls `setButtonBehavior`):

```java
transfer(Jr.Loop(Jr.Seq(Jr.Atom(new Incr()), Jr.Stop())));
```

Now, the result of pressing on the currently running button is to transfer the cyclic behavior to the other button, which thus becomes the new active button.

This little example shows how code migration can be based on the notion of a freezable instruction. Of course, migration through the network needs more: there must be a way to "serialize" reactive instructions, in order to transmit them (serialization is provided for free by Java). One does not consider this question in more details here.
2.8 Preemption and Control

Junior defines two operators to get fine control over reactive instructions: one is a preemption operator which forces a reactive instruction to terminate when an event is present; the other one allows a reactive instruction to execute according to presence of an event.

Preemption

To freeze a reactive instruction preempts it: execution of the frozen instruction is definitely abandoned by the executing machine. However, there is a side-effect which is to produce the residual of the frozen instruction. Instruction Until of Junior has the same behavior than Freezable, but does not computes any residual. It has the form: Jr.Until("event",body,handler), where body and handler are two reactive instructions. Execution of body is abandoned (one says, it is preempted) as soon as event becomes present; in this case, control directly goes to handler which is then executed.

For example, consider:

```python
Program controller =
    Jr.Loop(
        Jr.Until("suspend",
            Jr.Loop( Jr.Seq(Jr.Generate("step"),Jr.Stop()),
                    Jr.Seq(Jr.Stop(),Jr.Await("resume"))));
```

At each instant, inner loop generates event step. It is preempted by event suspend. In case of preemption, execution stops, and, at next instant, waiting for resume starts. Finally, the Until instruction completely terminates when resume occurs. As Until is enclosed by the external loop, it is then restarted, and cyclic generation of step begins again. Thus, controller actually implements a two-states behavior:

- In the first state, step is generated at each instant; transition to second state occurs when suspend is generated.

- Execution remains in the second state, while resume is not generated, and it returns to the first state when it is.

Control

The Junior Control instruction gives a way to execute a reactive instruction only at instants where a given event is present. At others instants, the reactive instruction just stays in the same state, without executing anything.
For example, in the following program, body is controlled by event step and is executed only when it is generated:

```java
Program controlledBody =
    Jr.Seq(Jr.Control("step", body), Jr.Generate("term"));
```

Control is completely terminated when body is so, and, then, event term is generated. Actually, the Control instruction can be seen as "filtering" instants for its body: the body can proceed only when Control let instants reach it.

## 3 Junior API

This section describes Jr, the API (Application Programming Interface) for programming in Junior.

### 3.1 Identifiers

In section 2, events were identified by strings; however, in Junior, the basic way to identify events is more abstract and uses identifiers. Identifiers implement the Identifier interface which defines equality and hashcode for identifiers. There is a unique assumption: equality of two identifiers of the same object must always return true. Equality and hashcode methods must be implemented as in the basic Java class Object (this allows, as example, to store identifiers in hashtables).

```java
public interface Identifier extends java.io.Serializable
{
    boolean equals(Object object);
    public int hashCode();
}
```

Here is the implementation of identifiers as strings:

```java
public class StringIdentifier implements Identifier
{
    public String id;
    public StringIdentifier(String id) { this.id = id; }
    public int hashCode() { return id.hashCode(); }
    public String toString() { return id; }
    public boolean equals(Object object)
    {
        if (object instanceof StringIdentifier)
```
3.2 Programs

Reactive instructions are manipulated as programs, satisfying the following interface:

```java
public interface Program extends java.io.Serializable {
    Program copy();
}
```

Thus, basically programs can be copied (to get new executable instances; programs are not reentrant) and are serializable, a property useful for migration, as indicated in 2.7.

3.3 Machines

A reactive machine runs a program, defines instants for it, and broadcasts generated events. New programs which are dynamically added to the machine are put in parallel with the current running program. Running program components can be removed from the machine, after being frozen. Reactive machines implement the `Machine` interface:

```java
public interface Machine {
    boolean react();
    void add(Program program);
    void generate(Identifier event);
    void generate(Identifier event, Object val);
    Program getFrozen (Identifier event);
}
```

- A reaction of the machine is performed in response to a call of the `react` method. The call returns true if the machine program is completely terminated, and false otherwise. Machine reactions define the instants of execution of the program in it.

- The `add` method adds its parameter to the machine program. Added programs are put in parallel with the currently running program.
• Methods `generate` generate events in the machine. Values, which are
Java objects, can be associated to event generations. Values associated
to events are accessible by the environment provided by the machine
(described in 3.5).

• Method `getFrozen(event)` returns the program constructed from resi-
duals of instructions frozen (by generating `event`) during previous re-
action (using `Freezable` instruction; see 3.11); if several instructions
were frozen, residuals are put in parallel in the result of `getFrozen`.
If no instructions were frozen during previous reaction, `getFrozen`
returns the `null` value.

Machine's methods can be directly called by atoms executed by the ma-
chine itself. The following specifications are introduced to cover these calls:

• Added instructions are collected during machine reaction and are in-
troduced in the machine at the beginning of the next instant. In this
way, program executions are independent of the actual timing of drop-
ping actions during instants.

• Events generated by methods `generate` become immediately present
in the machine.

• During a reaction, the effect of an other call to `react` is unspecified
(`react` is not reentrant).

Actually, machines are implemented in two ways: standard machines
and safe machines.

**Standard Machines**

Standard machines are not thread-safe and thus must be used cautiously in
multithreaded contexts. The constructors for standard machines are:

```java
Machine Jr.Machine(Program p)
Machine Jr.Machine()
```

In the first constructor, the argument is the initial machine program. In the
second, initial program is `Jr.Nothing()`.
Safe Machines

Safe machines are thread-safe, and are thus protected against concurrent accesses. Safe machines are simpler to use in multithreaded contexts (for example, when the AWT is involved). Basically, safe machines offer two main protections:

- Methods are protected against simultaneous accesses by concurrent threads. For example, two threads can concurrently add new programs without interference.
- Event generations issued by calls of `generate` are immediately processed when they occur before the end of current instant is decided. When occurring after the end of current instant, they are delayed to the next instant. In this way, there is no risk to lose an event generation issued by the machine environment.

Moreover, safe machines offer two more protections:

- Added programs are first copied, which implies that there is no possibility to get a reference on a program added in a safe machine.
- During a reaction, an other call to `react` has no effect; thus, `react` can be called by an atom without danger.

Empty interface `SafeMachine` flags synchronized machines:

```java
public interface SafeMachine {} extends Machine
```

The constructors for safe machines are:

- `SafeMachine Jr.SafeMachine(Program p)`
- `SafeMachine Jr.SafeMachine()`

In the first constructor, the argument is the initial machine program. In the second, initial program is `Jr.Nothing()`.

### 3.4 Basic Programs

The basic `Junior` programs are:

- `Nothing` does nothing and immediately terminates.
- `Stop` does nothing, stops for the current instant, and terminates at next instant.
• **Seq** puts a program in sequence with an other. Execution of the second program is immediately started when the first program becomes completely terminated.

• **Par** puts two programs in parallel. The parallel instruction is completely terminated when its two left and right branches are also completely terminated.

• **Loop** defines an infinite loop. Execution of the body is immediately restarted from its beginning when it is completely terminated.

Corresponding constructors are:

```plaintext
Program Jr.Nothing()
Program Jr.Stop()
Program Jr.Seq(Program first, Program second)
Program Jr.Par(Program left, Program right)
Program Jr.Loop(Program body)
```

It is the programmer responsibility to get control over loops that never converge, executing their body during one instant without never ending. These loops are called *instantaneous loop*: executing an instantaneous loop would prevent other program component to be executed. Instantaneous loops are discussed in 4.4.

To ease programming, the sequence and parallel constructors are extended to arrays:

```plaintext
Program Jr.Seq(Program[] array)
Program Jr.Par(Program[] array)
```

If size of the array is 0, then both constructors return **Jr.Nothing()**; if size of the array is 1, then both return the only element; if there are more than one element, then they are combined to form the result. For example, if array T contains 3 elements A, B, and C, then **Jr.Seq(T)** returns **Jr.Seq(A, Jr.Seq(B, C))**, and **Jr.Par(T)** returns **Jr.Par(A, Jr.Par(B, C))**.

Constructor **Halt** is also defined to ease programming in some cases; it defines a program that never terminates:

```plaintext
Program Jr.Halt()
```

Actually, **Jr.Halt()** is equivalent to **Jr.Loop(Jr.Stop())**.
3.5 Environments

Environments allows atoms and expressions to access values associated to generated events, residuals of frozen instructions, and implicit object. Environments are parameters of execute and of evaluate methods of actions and expressions. Interface Environment is:

```java
public interface Environment {
    Object[] previousValues (Identifier id);
    Object[] currentValues (Identifier id);
    Program getFrozen (Identifier id);
    Object linkedObject();
}
```

- `previousValues` returns the array of values generated for one event during previous instant. These values are only available during current instant (that is, the instant following actual generations), and are lost later. An empty array (of size 0) is returned if no valued generation has occurred during previous instant.

- `currentValues` returns the array of values generated for one event during current instant. As for `previousValues`, an empty array is returned when no value is available.

- `getFrozen` returns the same information than the corresponding method of Machine (see 3.3).

- `linkedObject` returns the implicit object, set by Link instructions (see 3.10). The null value is returned in case there is no associated implicit object.

3.6 Action and Atoms

Actions are used by atoms to interact with Java in an atomic manner. Constructor for atoms is:

```java
Program Jr.Atom(Action action)
```

The specific constructor for the Print atoms is:

```java
Program Jr.Print(String msg)
```

Interface Action is defined by:
3 JUNIOR API

3.7 Wrappers

Wrappers return a value when their evaluate method is called. They give possibility to reactive instructions to compute values during execution (not at construction). For example, an integer wrapper (with an evaluate method returning a long) is used by the Repeat finite loop instruction to compute the number of iterations of the body. There is a general rule: wrappers are evaluated (that is, the evaluate method is called) just before executing for the first time the reactive instruction in which they are embedded.

Basic Wrappers

There are four interfaces for wrappers returning identifiers, booleans, integers, or general Java objects. In all cases, the evaluate method has the machine environment as parameter.

```java
class PrintAction implements Action{
    public String msg;
    public PrintAction(String msg){ this.msg = msg; }
    public String toString(){ return System.out.println("msg"); }
    public void execute(Environment env){ System.out.println(msg); }
}
```

```java
class IdentifierWrapper extends java.io.Serializable{
    Identifier evaluate(Environment env);
}
```

```java
class BooleanWrapper extends java.io.Serializable{
    boolean evaluate(Environment env);
}
```
public interface IntegerWrapper extends java.io.Serializable
{
    public long evaluate(Environment env);
}

public interface ObjectWrapper extends java.io.Serializable
{
    Object evaluate(Environment env);
}

Constant Wrappers

A constant wrapper always returns the same value given at construction. There are 4 constructors for constant wrappers:

    IdentifierWrapper Jr.ConstWrapper(Identifier id)
    ObjectWrapper Jr.ConstWrapper(Object obj)
    IntegerWrapper Jr.ConstWrapper(long n)
    BooleanWrapper Jr.ConstWrapper(boolean b)

3.8 If

At first instant of execution, If reactive instruction evaluates a boolean wrapper, and, accordingly to the result, chooses the "then" or "else" branch for execution. Execution may take more than one instant, and at next instants, the chosen branch is continued. Constructors are:

    Program Jr.If(BooleanWrapper cond, Program thenInst, Program elseInst)
    Program Jr.If(BooleanWrapper cond, Program thenInst)

The second constructor is a syntactic facility: Jr.If(cond,thenInst) is equivalent to Jr.If(cond,thenInst,Jr.Nothing()).

3.9 Repeat

Repeat reactive instructions implement finite loops. The number of iterations is returned by an integer wrapper which is evaluated at the first instant of execution, before body execution. There are two constructors:

    Program Jr.Repeat(IntegerWrapper wrapper, Program body)
    Program Jr.Repeat(long count, Program body)

The second constructor is a syntactic facility: Jr.Repeat(n, body) is equivalent to Jr.Repeat(Jr.ConstWrapper(n), body). It is used when the number of iteration is a constant.
3.10 Link

A link associates a Java object to a reactive statement; the object becomes the implicit Java object (returned by method `linkedObject` of the environment; see 3.5) while executing the instruction. There are two constructors:

```java
Program Jr.Link(ObjectWrapper wrapper, Program body)
Program Jr.Link(Object object, Program body)
```

The second constructor is an abbreviation: `Jr.Link(obj, body)` is equivalent to `Jr.Link(Jr(ConstWrapper(obj), body)`. It is used when `obj` is determined at construction.

3.11 Freezable

A freezable instruction executes its body, while a freezing event is not generated. In case the event is generated, then the body is extracted from the program, after having finished to react for current instant, and the residual (what remains to do for future instants) is stored in the environment. The residual can be obtained at next instant, by calling method `getFrozen` of the environment (for atoms; see 3.5) or of the machine (for interfacing with Java; see 3.3).

```java
Program Jr.Freezable(IdentifierWrapper wrapper, Program body)
Program Jr.Freezable(Identifier event, Program body)
```

The second constructor is an abbreviation: `Jr.Freezable(event, body)` is equivalent to `Jr.Freezable(Jr(ConstWrapper(event), body)`. It is used when `event` is determined at construction.

3.12 Generate

An event is a data which is present or absent during instants. An event becomes present during one instant as soon as it is generated during this instant; otherwise, it is absent. Events are broadcast: during the course of an instant, the same event cannot be tested with two different presence status (present or absent). Values can be associated to event generations: they are collected during current instant and are available through the machine environment (see 3.5).
Generate without value

Event generations without values are managed by the following constructors:

Program Jr.Generate(IdentifierWrapper iwrap)
Program Jr.Generate(Identifier event)

The second constructor is an abbreviation: Jr.Generate(event) is equivalent to Jr.Generate(Jr.ConstWrapper(event)). It is used when event is determined at construction.

Generate with value

There are 4 constructors for generations with values:

Program Jr.Generate(IdentifierWrapper iwrap, ObjectWrapper owrap)
Program Jr.Generate(Identifier event, ObjectWrapper owrap)
Program Jr.Generate(IdentifierWrapper iwrap, Object value)
Program Jr.Generate(Identifier event, Object value)

The three last constructors are abbreviations, mapping values to wrappers in the standard way; for example: Jr.Generate(event, value) is equivalent to:

Jr.Generate(Jr.ConstWrapper(event), Jr.ConstWrapper(value))

3.13 Configurations

Configurations are boolean combinations of events, used by some reactive instructions (for example, Await, defined in 3.14). There are 4 kinds of configurations:

- Presence is the basic configuration, associated to an event. It is satisfied when the event is present, and unsatisfied when it is absent, that is when the current instant is terminated.
- And is the conjunction of two configurations; it is satisfied when both are satisfied, and unsatisfied when at least one is unsatisfied.
- Or is the disjunction of two configurations; it is satisfied when one (or both) is satisfied, and unsatisfied when both are unsatisfied.
- Not is the negation of a configuration; its is satisfied when the configuration is unsatisfied, and unsatisfied when the configuration is satisfied.
Configurations are used through interface:

```java
class Configuration {
    public interface Configuration extends java.io.Serializable {
        Configuration copy();
    }
}
```

Constructors for configurations are:

```java
Configuration Jr.Prepare(IdentifierWrapper wrapper)
Configuration Jr.Prepare(Identifier event)
Configuration Jr.And(Configuration c1, Configuration c2)
Configuration Jr.Or(Configuration c1, Configuration c2)
Configuration Jr.Not(Configuration c)
```

The second constructor is an abbreviation: Jr.Prepare(event) is equivalent to Jr.Prepare(Jr.ConstWrapper(event)). It is used when event is determined at construction.

### 3.14 Await

The `Await` instruction waits for a configuration to be satisfied; the waiting can take several instants; execution is stopped while the configuration is unsatisfied and it terminates when it is satisfied (of course, it can never terminate if the configuration is never satisfied).

```java
Program Jr.Await(Configuration config)
Program Jr.Await(IdentifierWrapper wrapper)
Program Jr.Await(Identifier event)
```

The second and third constructors are abbreviations: Jr.Await(wrapper) is equivalent to Jr.Await(Jr.Prepare(wrapper)), and Jr.Await(event) is equivalent to Jr.Await(Jr.Prepare(event)).

### 3.15 Until

The `Until` instruction preempts its body when a configuration becomes satisfied. In case of preemption, execution of the body is complete for current instant, but is abandoned for future instants; moreover, in this case, execution switches to the handler part of the Until instruction.

```java
Program Jr.Until(Configuration config, Program body, Program handler)
Program Jr.Until(Configuration config, Program body)
```
The second constructors is an abbreviation: \texttt{Jr.
Until} (\texttt{conf, body}) is equivalent to \texttt{Jr.
Until} (\texttt{conf, body, Jr.
Nothing}()).

Identifier wrappers and identifiers can be used in place of configurations:

\begin{verbatim}
Program Jr.
Until(IdentifierWrapper wrapper, Program body, Program handler)
Program Jr.
Until(Identifier event, Program body, Program handler)
\end{verbatim}

These constructors are abbreviations, mapping values to wrappers in the
standard way; for example: \texttt{Jr.
Until} (\texttt{wrapper, body, handler}) is equiva-
 lent to:

\begin{verbatim}
Jr.
Until(Jr.
Presence(wrapper), body, handler)
\end{verbatim}

Finally, as previously, the handler part can be omitted, and in this case it
is \texttt{Jr.
Nothing}(). This gives the two constructors:

\begin{verbatim}
Program Jr.
Until(IdentifierWrapper wrapper, Program body)
Program Jr.
Until(Identifier event, Program body)
\end{verbatim}

### 3.16 When

The \texttt{When} instruction is similar to \texttt{If} except that an event configuration is
tested instead of a boolean expression. First, \texttt{When} evaluates a configura-
tion, and if it is satisfied, execution goes to the "then" branch; otherwise,
execution goes to the "else" branch. An important point is that evaluation
of a configuration can take the entire instant if the absence of some events
is to be considered. In this case, execution of the chosen branch can only
start at the instant that follows configuration evaluation.

\begin{verbatim}
Program Jr.
When(Configuration config, Program thenInst, Program elseInst)
\end{verbatim}

Identifier wrappers and identifiers can be used in place of configurations:

\begin{verbatim}
Program Jr.
When(IdentifierWrapper wrapper, Program thenInst, Program elseInst)
Program Jr.
When(Identifier event, Program thenInst, Program elseInst)
\end{verbatim}

These constructors are abbreviations, mapping values to wrappers in the
standard way.
The "else" part can be omitted, and in this case it is \texttt{Jr.
Nothing}(). This
gives constructors:

\begin{verbatim}
Program Jr.
When(Configuration config, Program thenInst)
Program When(IdentifierWrapper wrapper, Program thenInst)
Program When(Identifier event, Program thenInst)
\end{verbatim}
3.17 Control

The body of a Control instruction is executed only at instants where a controlling event is present.

Program Jr.Control(IdentifierWrapper wrapper, Program body)
Program Jr.Control(Identifier event, Program body)

The second constructor is an abbreviation: Jr.Control(event, body) is equivalent to Jr.Control(Jr.Presence(event), body).

3.18 Local

The Local instruction defines a local event in the scope of its body:

Program Jr.Local(Identifier event, Program body)

3.19 Events identified by Strings

Events can be systematically denoted by Java strings. Denotation is based on the following functions:

- **String to Identifier**: the StringIdentifier function defined in 3.1 transforms strings into identifiers.

- **String to IdentifierWrappers**: the StringWrapper function transforms strings into identifier wrappers; it is defined by:

  ```java
  IdentifierWrapper StringWrapper(String str){
      return Jr.ConstWrapper(StringIdentifier(str));
  }
  ```

- **String to Configuration**: the following StringConfig function transforms strings into configurations.

  ```java
  Configuration StringConfig(String str){
      return new Presence(StringWrapper(str));
  }
  ```

Using the previous functions, one has the new constructors:
Program Generate(String event, ObjectWrapper cwrap)
Program Jr.Generate(String event, Object val)
Program Jr.Generate(String event)
Program Jr.Await(String event)
Program Jr.Control(String event, Program body)
Program Jr.Local(String event, Program body)
Program Jr.Freezeable(String event, Program body)
Program Jr.Until(String event, Program body, Program handler)
Program Jr.Until(String event, Program body)
Program Jr.When(String event, Program thenInst, Program elseInst)
Program Jr.When(String event, Program thenInst)
Configuration Jr.Presence(String event)
Configuration Jr.Not(String event)

4 Details of Junior Programming

This section contains a discussion on several aspects of Junior programming.

4.1 Identifiers

In Junior, events are, in an abstract way, denoted by identifiers (see 3.1). The reason to choose this abstract vision is that it would ease interfacing with a distributed execution platform having its own notion of an identifier.

To make programming easier, Jr gives several means for building events. With the simplest one, events are identified by Java strings, and several methods are provided to build identifier wrappers and configurations out of strings.

4.2 Reaction to Absence

In Junior, an event cannot be considered as absent before the end of the instant. This is a coherent point of view as, otherwise, the event could be generated after being considered as absent; it would thus be absent and present during the same instant, a situation that would contradict the definition of an event. A different point of view is the one of Synchronous Languages[4], for example Esterel[1], in which events (called signals) can be considered as absent in the course of an instant, provided that a static analysis of the program, performed at compile time, has indicated that no generation of the event remains possible. Junior chooses not to use static analysis technics; this is the reason why, in Junior, one has to wait for the
end of instant to be sure that events are really absent. There are several
consequences:

- Instructions in sequence with an absence test are always postponed to
the next instant. For example, consider:

\[
\text{Jr.Seq(} \text{Jr.When("event", Jr.Nothing()), Jr.Generate("absent"))}
\]

Consider the case where \textit{event} is absent. Then, as the test of absence
"takes the entire current instant", \texttt{absent} cannot be generated during
the current instant, but it is during the next. Situation would be
exactly the same with:

\[
\text{Jr.When("event", Jr.Nothing(), Jr.Generate("absent"))}
\]

- To determine the status of a configuration (satisfied or unsatisfied) can
also "take the entire instant", if absent events are to be considered.
For example, in \texttt{Jr.Await(Jr.Not("event"))} termination occurs at
the instant following the one where \textit{event} is absent.

Consequences concerning the \texttt{Until} instruction are considered in 4.3.

### 4.3 Preemption

The \texttt{Until} and \texttt{Freezable} instructions always let their body react, even if
the triggering event (the preemption event for \texttt{Until}, and the freezing event
for \texttt{Freezable}) is present\(^3\). In some cases, this can have consequences for
instruction termination; for example, consider:

\[
\text{Jr.Until("trigger", Jr.Loop(Jr.Stop()))}
\]

Execution is stopped while \texttt{trigger} is not generated, and \texttt{Until}
completely terminates when \texttt{trigger} is generated. The point is that, in this case,
generation and termination both occur at the same instant. The situation
is different with:

\[
\text{Jr.Until("trigger", Jr.Await("never"))}
\]

\(^3\)One says that the preemption of \texttt{Until} is "weak", compared to the "strong" one of
the \texttt{abort} construct of Esterel.
4 DETAILS OF JUNIOR PROGRAMMING

4.4 Reactivity

Convergence

The basic assumption of Junior is that execution during an instant always converges and leads to a stable state from which next instant can start. There are two possibilities for the assumption to be violated:

- Executing an action which never converge will forbid the machine to finish current instant. These situations are out of control of the reactive program and it is the programmer responsibility to avoid them. Here is an example of "pathological" action:

  ```java
  public class NeverEnding implements Action
  {
    public void execute(Environment env){ while(true){/* empty */} }
  }
  ```

  One could think of machine implementations that would force action termination according to some criteria (as, for example, a fixed delay given for each action execution). Present implementations do not consider this aspect, which is left for further research.

- Instantaneous loops are loops with immediately terminating bodies (see 3.4). Instantaneous loops have to be eliminated. In Junior, it is the programmer responsibility to control this point. A way to do this could be to extend class Loop and to systematically put a Stop instruction in parallel with the loop body. This is the standard approach of SugarCubes, in which instantaneous loop elimination is the default behavior.

  Note that there is no possibility to fall in a non-convergent situation by recursive addings of programs, as additions are systematically postponed to the next instant.

27
Response Time

In addition to convergence of reactive programs, one sometimes want to be sure that instants will not take too much time. That is, one looks for a kind of response time warranty. This is presently out of scope in Junior.

4.5 Events and Values

Values can be associated to event generations. All values associated to one event are collected during the instant, and are available at next instant through the environment (previousValues method; see 3.5). Let's for example suppose that the following instruction is run:

\[
\text{Jr.Seq(Jr.Generate("E",01), Jr.Generate("E",02))}
\]

Then, at next instant previousValues("E") would return the array [01, 02].

Generations without associated values are transparent; for example, [02] would be the result of running:

\[
\text{Jr.Seq(Jr.Generate("E"), Jr.Generate("E",02))}
\]

There are two difficulties:

- Values are returned with type Object; it is the programmer responsibility to make the correct cast, when it is needed. The reflexive facilities of Java could help in using these values correctly.

- The order in which values are collected depends on the order Par instructions choose their branches. It is thus not possible to make any assumption on it.

Junior also provides a way to get values associated to event generations performed during current instant (currentValues method; see 3.5). To access values during current instant is extremely error prone because one cannot be sure that all generated values are actually got. For example, the fact that currentValues returns an empty array does not mean that event is absent, because it may be generated later during the instant.

4.6 Implementation of Parallelism

The Par operator does not specify order in which branches are executed; this introduces nondeterminism in program behaviors as program semantics
depends on the order \texttt{Par} branches are executed. Actually, presence of operators such as \texttt{Par} is rather natural in the context of parallel programming, because they reflect nondeterminism associated to parallelism. However, it is possible for \texttt{Par} implementations to always choose the same fixed order; the benefit is then to reduce nondeterminism of program. This is the option taken by two existing implementations (called \texttt{REWRITE} and \texttt{REPLACE}). However, in a third implementation (called \texttt{SIMPLE}), the order of parallel branches execution can change, depending on the actual programs executed. Thus, \texttt{Junior} programmers should not rely on the order \texttt{Par} branches are executed.

In \texttt{REWRITE} and \texttt{REPLACE} implementations, \texttt{Par} executes its two branches in a fixed order; more precisely, at each instant, \texttt{Par(A,B)} first starts executing the first branch \texttt{A}, then the second one \texttt{B}. For example, consider:

\begin{verbatim}
Jr.Par(Jr.Print("hello, "),Jr.Print("world"))
\end{verbatim}

In \texttt{REWRITE} and \texttt{REPLACE} implementations, the only possible output is "hello, world!". With \texttt{SIMPLE}, both "hello, world!" and "world! hello," are possible. However, events can be used when one needs a precise order of atoms interleaving. For example, consider:

\begin{verbatim}
Jr.Par( 
  Jr.Seq(Jr.Generate("e"),Jr.Print("hello, "))
  Jr.Seq(Jr.Await("e"),Jr.Print("world")),
)
\end{verbatim}

Actually, because of the synchronization introduced by event \texttt{e}, the only possible output is "hello, world!". This output is thus the only one produced by any implementation of \texttt{Junior}.

\section{Safe Machines}

There is no need for a safe machine when an executable program is run by one unique thread. However, there are many cases where several execution threads are used: for example, Java applets create several threads to process graphics and graphical events; concurrent threads also naturally appear when network is involved.

Let us return to example of section 2.3; the machine should be protected to avoid uncontroled interferences of buttons press actions with program execution. Moreover, as the Java graphical toolkit \texttt{Awt} is involved, it is mandatory to give the graphical thread enough time to work. This leads to the following executable code:
public class Dynamic {
    public static void main(String[] args) {
        Machine machine = Jr.SafeMachine();
        Button button = new Button("press here");
        button.addActionListener(
                new Drop(machine, Jr.Print("hello, world!\n")));
        Frame frame = new Frame();
        frame.add(button);
        frame.pack(); frame.show();
        while (true) {
            machine.react();
            try { Thread.sleep(100); } catch (Exception e) {} 
        }
    }
}

The Thread.sleep method is called to give enough time to the graphical thread; value of the parameter is actually dependent of the execution machine.

### 4.8 Reactive Instructions vs Threads

By getting fine control over threads execution, programmers can define their own threads scheduling algorithms. This leads to a flexible and reflexive approach to concurrency, in which the basic execution scheme of the system is under control of it. Unfortunately, to get fine control over thread execution while staying in a safe programming context appears to be an extremely difficult task in Java. In first versions, Java provided programmers with 3 primitives to kill, suspend, and resume threads (Thread.stop, Thread.suspend, and Thread.resume). It has been discovered that these primitives are highly error prone, and they have been removed from later versions of the language[7]. A comparison of Java threads with reactive instructions (in the particular context of SugarCubes) can be found in [3]. Here, one limits discussion to the specific problem of gaining fine control over instructions, the one that causes troubles with Java threads.

Using Until and Control, programmers can get fine control over reactive instructions execution. For example, one can easily define a kind of "reactive thread" which can be started, stopped, suspended, and resumed, as Java threads are (remember that in Java Thread.stop kills threads). Actually, reusing the two programs controlledBody and controller defined in 2.8, the behavior of a reactive thread can be simply defined as:

    Jr.Seq(Jr.Await("start"),
Local events are defined using the Local instruction. One uses here the full version of Until, in which preemption is governed by a configuration of events; here, configuration is an Or which is satisfied when stop or term are present (or both).

The behavior of a reactive thread is thus:

- The thread starts when event start is generated.
- The thread completely terminates when its body terminates (then, event term is generated), or when the thread is stopped (event stop is generated).
- The thread is suspended (generations of the local event step are stopped) when suspend is generated; it is resumed when event resume is generated (generations of step restart).

This example shows that reactive instructions of Junior are actually an alternative to Java threads for concurrent programming when fine control over execution is needed.

5 Conclusion

Junior is set of Java classes for reactive programming. It is used through an API which defines a small set of constructors, with a precise semantics. Junior defines a set of high-level primitives for defining reactive programs, such as the parallelism and the preemption operators. This leads to a programming which is flexible and which provides users with a fine control over program executions.
References


[8] Information on Jonathan is available at http://www.objectweb.org
# Contents

1 Introduction ................................................. 1  
  1.1 Jr ..................................................... 2  

2 Junior Overview ............................................. 2  
  2.1 Reactive Instructions .................................... 2  
  2.2 Reactive Machines ....................................... 3  
  2.3 Dynamicity ............................................ 5  
  2.4 Events ............................................... 6  
  2.5 Concurrency .......................................... 7  
  2.6 Interfacing with Java .................................... 8  
  2.7 Migration ............................................. 9  
  2.8 Preemption and Control ................................ 11  

3 Junior API .................................................... 12  
  3.1 Identifiers ........................................... 12  
  3.2 Programs ............................................ 13  
  3.3 Machines ............................................. 13  
  3.4 Basic Programs ....................................... 15  
  3.5 Environments .......................................... 17  
  3.6 Action and Atoms ...................................... 17  
  3.7 Wrappers ............................................. 18  
  3.8 If .................................................... 19  
  3.9 Repeat ............................................... 19  
  3.10 Link ............................................... 20  
  3.11 Freezable ........................................... 20  
  3.12 Generate ............................................ 20  
  3.13 Configurations ........................................ 21  
  3.14 Await ............................................... 22  
  3.15 Until ............................................... 22  
  3.16 When ............................................... 23  
  3.17 Control ............................................. 24  
  3.18 Local ............................................... 24  
  3.19 Events identified by Strings ......................... 24  

4 Details of Junior Programming .............................. 25  
  4.1 Identifiers ........................................... 25  
  4.2 Reaction to Absence .................................... 25  
  4.3 Preemption .......................................... 26  
  4.4 Reactivity .......................................... 27  
  4.5 Events and Values .................................... 28  
  4.6 Implementation of Parallelism ......................... 28  
  4.7 Safe Machines ........................................ 29  
  4.8 Reactive Instructions vs Threads ...................... 30  

33
5 Conclusion 31
Index

Action, 17
actionPerformed, 5
add, 4
And, 22
Atom, 17
atom, 3
Await, 6, 22

BooleanWrapper, 8, 18
completely terminated instruction, 3
Configuration, 22
constant wrapper, 19
ConstWrapper, 19
Control, 11, 24
copy, 5, 13, 22
currentValues, 17

Environment, 17
Events identified by Strings, 24
execute, 8, 17

Freezable, 9, 20, 26
Generate, 6, 21
generate, 13
getFrozen, 10, 13, 17

Identifier, 12
identifier, 12, 25
IdentifierWrapper, 18
If, 8, 19
implicit object, 8
instantaneous loop, 16, 27
IntegerWrapper, 18

Jr, 2, 12
junior package, 4

Link, 8, 20
linkedObject, 8, 17, 20
Local, 24, 31
Loop, 16

Machine, 4, 13
main, 4

Not, 22
Nothing, 16

ObjectWrapper, 19
Or, 22

Par, 7, 16, 28, 29
preemption, 26
Presence, 22
previousValues, 17
Print, 3, 17
Program, 5, 13

react, 4, 13
reaction to absence, 25
Repeat, 19
residual, 9

safe machine, 15, 29
SafeMachine, 4, 15
Seq, 3, 16
Stop, 5, 16
StringConfig, 24
StringIdentifier, 12
StringWrapper, 24

Thread.sleep, 30

unsynchronized machine, 14
Until, 11, 22, 26

When, 23, 26
Annex: The Jr and JrBase Classes

Class JrBase

```java
public class JrBase {
    // Constant wrappers
    static public IdentifierWrapper ConstWrapper(Identifier id) {
        return new ConstIdentifierWrapper(id);
    }
    static public ObjectWrapper ConstWrapper(Object obj) {
        return new ConstObjectWrapper(obj);
    }
    static public IntegerWrapper ConstWrapper(int n) {
        return new ConstIntegerWrapper(n);
    }
    static public BooleanWrapper ConstWrapper(boolean b) {
        return new ConstBooleanWrapper(b);
    }
    // Machines
    static public Machine Machine(Program p) {
        return new MachineImpl(p);
    }
    static public SafeMachine SafeMachine(Program p) {
        return new SafeMachineImpl(p);
    }
    // Atom
    static public Program Atom(Action action) {
        return new Atom(action);
    }
    // Nothing, Stop, Seq, Par, Loop
    static public Program Nothing() {
        return new Nothing();
    }
    static public Program Stop() {
        return new Stop();
    }
    static public Program Seq(Program first, Program second) {
        return new Seq(first, second);
    }
    static public Program Par(Program left, Program right) {
        return new Par(left, right);
    }
    static public Program Loop(Program body) {
        return new Loop(body);
    }
    // Repeat
    static public Program Repeat(IntegerWrapper wrapper, Program body) {
        return new Repeat(wrapper, body);
    }
```
} // If
static public Program If(BooleanWrapper cond,
    Program thenInst, Program elseInst){
    return new If(cond,thenInst,elseInst);
}
// Generate
static public Program Generate(IdentifierWrapper iwrap,
    ObjectWrapper owrap){
    return new Generate(iwrap,owrap);
}
// Await, Until, When
static public Program Await(Configuration config){
    return new Await(config);
}
static public Program Until(Configuration config,
    Program body, Program handler){
    return new Until(config,body,handler);
}
static public Program When(Configuration config,
    Program thenInst, Program elseInst){
    return new When(config,thenInst,elseInst);
}
// Local
static public Program Local(Identifier event, Program body){
    return new Local(event,new EventDataImpl(),body);
}
// Control
static public Program Control(IdentifierWrapper wrapper, Program body){
    return new Control(new Presence(wrapper),body);
}
// Freezable
static public Program Freezable(IdentifierWrapper wrapper, Program body){
    return new Freezable(new Presence(wrapper),body);
}
// Link
static public Program Link(ObjectWrapper wrapper, Program body){
    return new Link(wrapper,body);
}
// Print
static public Program Print(String str){
    return new Atom(new Print(str));
}
// Configurations
static public Configuration Presence(IdentifierWrapper wrapper){
    return new Presence(wrapper);
}
static public Configuration And(Configuration c1, Configuration c2){
    return new And(c1,c2);
static public Configuration Or(Configuration c1, Configuration c2){
    return new Or(c1,c2);
}
static public Configuration Not(Configuration c){
    return new Not(c);
}

Class Jr

public class Jr extends JrBase
{
    static public Identifier StringIdentifier(String str){
        return new StringIdentifier(str);
    }
    static public IdentifierWrapper StringWrapper(String str){
        return ConstWrapper(StringIdentifier(str));
    }
    static public Configuration StringConfig(String str){
        return new Presence(StringWrapper(str));
    }
    // Repeat
    static public Program Repeat(long count, Program body){
        return JrBase.Repeat(ConstWrapper(count),body);
    }
    // if
    static public Program If(BooleanWrapper cond,Program thenInst){
        return JrBase.If(cond,thenInst,Nothing());
    }
    // Generate
    static public Program Generate(Identifier event,ObjectWrapper owrap){
        return JrBase.Generate(ConstWrapper(event),owrap);
    }
    static public Program Generate(IdentifierWrapper iwrap,Object value){
        return JrBase.Generate(iwrap,ConstWrapper(value));
    }
    static public Program Generate(Identifier event,Object value){
        return JrBase.Generate(ConstWrapper(event),ConstWrapper(value));
    }
    static public Program Generate(Identifier event, Wifi iwrap){
        return JrBase.Generate(iwrap,ConstWrapper(Wifi.NO_VALUE));
    }
    static public Program Generate(Identifier event){
        return Generate(event,ConstWrapper(Wifi.NO_VALUE));
    }
    // Await
    static public Program Await(IdentifierWrapper wrapper){

return JrBase.Await(Presence(wrapper));
}  
static public Program Await(Identifier event) {
    return JrBase.Await(Presence(event));
}

// Until
static public Program Until(Configuration config, Program body) {
    return JrBase.Until(config, body, Nothing());
}

static public Program Until(IdentifierWrapper wrapper, Program body, Program handler) {
    return JrBase.Until(Presence(wrapper), body, handler);
}

static public Program Until(IdentifierWrapper wrapper, Program body) {
    return Until(wrapper, body, Nothing());
}

static public Program Until(Identifier event, Program body, Program handler) {
    return Until(ConstWrapper(event), body, handler);
}

static public Program Until(Identifier event, Program body) {
    return Until(ConstWrapper(event), body, Nothing());
}

// When
static public Program When(Configuration config, Program thenInst) {
    return JrBase.When(config, thenInst, Nothing());
}

static public Program When(IdentifierWrapper wrapper, Program thenInst, Program elseInst) {
    return JrBase.When(Presence(wrapper), thenInst, elseInst);
}

static public Program When(IdentifierWrapper wrapper, Program thenInst) {
    return When(wrapper, thenInst, Nothing());
}

static public Program When(Identifier event, Program thenInst, Program elseInst) {
    return When(ConstWrapper(event), thenInst, elseInst);
}

static public Program When(Identifier event, Program thenInst) {
    return When(ConstWrapper(event), thenInst, Nothing());
}

// Control
static public Program Control(Identifier event, Program body) {
    return JrBase.Control(ConstWrapper(event), body);
}

// Freezable
static public Program Freezable(Identifier event, Program body) {
    return JrBase.Freezeable(ConstWrapper(event), body);
}
// Link
static public Program Link(Object object, Program body){
    return JBase.Link(ConstWrapper(object),body);
}
// Configurations
static public Configuration Presence(Identifier event){
    return JBase.Presence(ConstWrapper(event));
}
/********** STRINGS ***********/
static public Program Generate(String event, ObjectWrapper owrap){
    return JBase.Generate(StringWrapper(event), owrap);
}
static public Program Generate(String event, Object val){
    return JBase.Generate(StringWrapper(event), ConstWrapper(val));
}
static public Program Generate(String event){
    return Generate(StringWrapper(event));
}
static public Program Await(String event){
    return JBase.Await(StringConfig(event));
}
static public Program Control(String event, Program body){
    return JBase.Control(StringWrapper(event), body);
}
static public Program Local(String event, Program body){
    return JBase.Local(StringIdentifier(event), body);
}
static public Program Freezeable(String event, Program body){
    return JBase.Freezeable(StringWrapper(event), body);
}
// until
static public Program Until(String event, Program body, Program handler){
    return JBase.Until(StringConfig(event), body, handler);
}
static public Program Until(String event, Program body){
    return JBase.Until(StringConfig(event), body, Nothing());
}
// when
static public Program When(String event, Program thenInst, Program elseInst){
    return JBase.When(StringConfig(event), thenInst, elseInst);
}
static public Program When(String event, Program thenInst){
    return JBase.When(StringConfig(event), thenInst, Nothing());
}
// configurations
static public Configuration Presence(String event){
    return JBase.Presence(StringWrapper(event));
}
static public Configuration Not(String event){

40
return JrBase.Not(Presence(event));
}
}