The SugarCubes Tool Box
RSI-JAVA Implementation

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

May 13, 1997

Contents

1 Introduction .............................................. 2
   1.1 Behaviors ............................................. 4
   1.2 Objects .............................................. 5

2 Behaviors ................................................ 6
   2.1 Behavior Class ......................................... 6
   2.2 Behavior Environments ............................... 7

3 Runs ..................................................... 9

4 Objects ................................................. 11
   4.1 The RsiObject Class ................................. 11

5 Machines ............................................... 13
   5.1 RsiMachine ............................................ 13
   5.2 Interpreter ............................................ 14

6 External Instructions ................................. 16
   6.1 External class ........................................ 17
   6.2 Extern Node .......................................... 18
   6.3 If Instruction ....................................... 19

7 Grammar ............................................... 20

8 Conclusion ............................................ 25
1 Introduction

Reactive scripts provide a flexible approach, allowing parallelism, distribution, object orientation, and preemption primitives.

The language contains an original combination of event-driven synchronous programming with broadcast communication and object based programming, including dynamic creation of objects.

Here are the basic commands of reactive scripts (see [1] for a full description):

- **await E** waits for event E to be generated. (events need not be declared before being used). Event *configurations* extend this to more general situations where one waits for the simultaneous occurrence of several events, the occurrence of one amongst several events, or for the absence of an event (resp. and, or, and not constructs).

- **generate E** generates event E. Generation of an event concerns only the current reaction, and is lost for the future (events are not persistent). Events are broadcast that is, execution of generate E fires all the await E commands that are stored in the interpreter. Finally, generation of an already generated event has no effect.

An important point however is that the action controlled by the absence of an event will be delayed to the next reaction, following the absence decision principle which states that *reaction to absence is always postponed to the next instant*. The end of the reaction is the precise moment one is sure the event is definitely absent and not to delay reactions to event absences would cause trouble (often called “causality problems”), as in:

```plaintext
await not E;generate E
```

where E would be generated during the same reaction it is absent. This would violate the basic broadcast hypothesis of reactive scripts, which states that the presence/absence of an event is the same in the whole system.

- External statements like assignments, procedure calls or printing statements are always put between “{” and “}”. In this paper, of course we use Java syntax.

- Commands are grouped in blocks : they are executed in sequence when separated by “;” and in parallel when separated by “∥” ; a parallel block terminates when all its branches do (parentheses are used for precedence purposes).

- The **stop** command defines the end of instant in a sequence for the current interpreter reaction. It is the new starting point for the next reaction. The
stop command is needed to trigger execution of a command by several occurrences of the same event. For example, the following command prints message Two! after two occurrences of E:

```java
await E; stop; await E; {Out.println("Two!")}
```

- The next command forces the interpreter to execute the next reaction as soon as the current one is over.

- Cyclic behaviors are defined using the loop operator. The loop body is run as soon as the command is entered, and when it terminates it is automatically restarted.

  A problem would appear if a loop body would terminate in the same reaction it is started (one speaks of an "instantaneous loop”), as in:

```java
loop {System.out.println("OK!")} end
```

Execution would cycle producing infinitely many OK!, and would prevent the interpreter to terminate the current reaction. Instantaneous loops are detected at run time.

- There are two test commands: if which tests for boolean conditions, and when which tests for events. Tests of events always obey the absence decision principle. For example, consider:

```java
when E then
generate F
else
generate G
end
```

Event F is generated if E occurs in the current reaction; otherwise, G is generated in the next reaction.

- The until statement is the basic preemption operator of reactive scripts. It executes its body and terminates for two reasons: either because the body terminates, and in this case termination of the until is immediate; or because a given event occurs (the case of actual preemption), and then termination of the until depends on the body, accordingly to the absence decision principle. For example consider the command:

```java
do
    await E; {System.out.println("E!")}
||
    await F; {System.out.println("F!")}
until G;
{Out.println("Terminated!")}
```
If \( G \) does not occur before both \( E \) and \( F \) do, then all works as if the \texttt{until} command was not there: \( E! \) is printed as soon as \( E \) occurs, \( F! \) is printed as soon as \( F \) occurs, and \texttt{Terminated!} is printed simultaneously with the last event, as then the body of the \texttt{until} terminates. On the contrary, if \( G \) occurs while \( E \) or \( F \) have not yet occurred, then the \texttt{until} command is exited at the next reaction and \texttt{Terminated!} is printed at that time.

An "actual" part can be added to a \texttt{until} command, to be executed only in case of actual preemption.

- Execution of a command can be controlled by the occurrence of an event, using the \texttt{control} operator. Actually, the body of a control command is executed only during reactions where the controlling event occurs. For example, the following command prints a message only when \( E \) occurs:

\[
\texttt{control}
\begin{align*}
\texttt{loop \{System.out.println("OK!");\} stop end}
\texttt{by E}
\end{align*}
\]

### 1.1 Behaviors

A \textit{behavior} is a declaration which associates a name to a command (or command block). For example the following behavior associates the name \( B \) to the command that waits for \( E \) to print \texttt{OK}!:

\[
\texttt{behavior B}
\begin{align*}
\texttt{await E;\{}\texttt{System.out.println("OK!");}\}
\texttt{end}
\end{align*}
\]

A \textit{new fresh copy} of the behavior \( B \) is started every time a \texttt{run B} command is executed, and thus several runs of the same behavior can coexist without interference. The binding between a \texttt{run B} command and the behavior \( B \) is established dynamically, when the command \texttt{run} is executed. A behavior can be re-declared with the following rules:

- Re-declaration does not erase existing bindings. Previous runs continue their execution without change.
- The effect of a declaration only takes place at the next reaction.
- Multiple declarations of the same behavior during a single reaction are rejected (they have no effect) as they could generate unclear non-deterministic situations.

New statements can be dynamically added to a behavior, to be executed in parallel with it. The syntax to add a statement \( i \) to a behavior \( b \) is "add \( i \)
to b". Like for behavior redeclarations, the effect of adding a statement to a behavior is postponed to the next instant.

Finally, note that behaviors can have event parameters (see [1] for details).

1.2 Objects

Traditionally, objects encapsulate data which are processed by their methods. In reactive scripts we put the focus on behavioral aspects, and the task of defining and using variables and data is transferred to Java.

An object combines a body statement which is automatically executed at each instant, with attached methods which must be explicitly called to be executed. It has the syntax:

```
object O
  body
  methods
    M1 ... Mn
  end
```

This statement defines an object O with attached methods M1 ... Mn, whose semantics is:

```
do
  body
  || control run M1 by O-M1
  || ...
  || control run Mn by O-Mn
until O-destroy
```

Event O-destroy is used to destroy the object; see below. Note that, as opposed to behavior declarations, objects are statements which are immediately run.

The syntax to call the method M of an object O is send M to O, and its semantics is simply generate O-M.

These semantics definitions have several consequences:

- Because of the control statement, a method M of O is given one reaction each time the statement send M to O is executed, and is not executed otherwise. Moreover, execution is immediate: once a method is called during a reaction, it runs during this reaction.

- The send command terminates immediately (as the generate statement), and does not wait for the called method to start to execute. It is a “send and forget” order in which the caller can immediately continue to execute (thus, this is an asynchronous call). Note that non-determinism can occur, as to send two orders in sequence during the same reaction does not prevent the second one to be processed before the first (see [1] for an example).
• Methods are “one shot”: only the first call to a method is actual, the others having no effect (because generating the same event several times is equivalent to generating it once). The “one shot” property is important to prevent objects to enter into interlocking situations where, for example, two objects call each other for ever.

Objects are removed from the interpreter by destroying them, using the destroy command. The removal of an object does not prevent it to execute for the current reaction: the removal becomes effective only at the next reaction.

This presentation shows that objects and methods enter in a rather natural way into the broadcast event driven approach, although different in spirit from it as method calls are not broadcast but sent to precise targets. Reactive scripts give both ways of programming in an unified framework.

In this paper we describe the implementation of reactive scripts on top of Java, using Sugarcubes defined in the companion paper:

• The Sugarcubes Tool Box - Definition

Actually, we directly use Sugarcubes for most part of the language. The main items to implements are behaviors, runs, and objects.

2 Behaviors

2.1 Behavior Class

Behavior extends Atom and its action is to register itself in the SimMachine (defined in section 5.1) which executes it.

A behavior has a body which is an instruction, and it can have event parameters.

New instructions can be put in parallel into a behavior using the add method.

At creation, the actual parameter vector is copied into the array formal-Params for efficiency.

public class Behavior extends Atom
{
    protected String behavName;
    protected Instruction body;
    protected Param[] formalParams = null;

    public Behavior(String behavName, Vector paramList, Instruction body)
    {
        this.behavName = behavName;
        this.body = body;
        if (paramList != null){
formalParams = new Param[paramList.size()];
    paramList.copyInto(formalParams);
}

public Instruction body(){ return body; }
public Param[] formalParams(){ return formalParams; }

public void add(Instruction inst){
    body = new Merge(body,inst);
}

final public String toString(){ ... }

public Object clone()
{
    Behavior inst = (Behavior)super.clone();
    inst.body = (Instruction)body.clone();
    return inst;
}

final protected void action(Machine machine){
    ((RsiMachine)machine).behavEnv.newBehav(behavName,this);
}

2.2 Behavior Environments

A behavior environment of the class BehavEnv associates behaviors to names in the behavEnv hash table.

The newBehavs hash table contains the behaviors which are defined during current instant, and the definitions of which will only take effect at next instant.

Vector addToBehav contains the instructions which are added to the behavior during the current instant.

The transferBehavs method is called at the beginning of each instant, in order to transfer behavior definitions made during last instant. First, behaviors are transferred from newBehavs, then instructions from addToBehav (NamedInst is an auxiliary class to link together an instruction and a name).

newBehav method changes a behavior declaration for next instant. There must be only one definition during current instant, otherwise the effect is to set the behavior body to nothing.

addToBehav method adds an instruction to a behavior; this will take place at next instant.

behavNamed method returns the behavior associated to a name.
public class BehavEnv
{
    private Hashtable behavEnv = new Hashtable();
    private Hashtable newBehavs = new Hashtable();
    private Vector addToBehav = new Vector();

    public void transferBehavs()
    {
        Enumeration nameEnum = newBehavs.keys();
        while (nameEnum.hasMoreElements())
        {
            String name = (String)nameEnum.nextElement();
            Instruction body = (Instruction)newBehavs.remove(name);
            behavEnv.put(name, body);
        }
        for (int i = 0; i < addToBehav.size(); i++)
        {
            NamedInst c = (NamedInst)addToBehav.elementAt(i);
            Behavior behav = behavNamed(c.name());
            if (beHAV != null) behav.add(c.inst());
        }
        addToBehav = new Vector();
    }

    void newBehav(String name, Behavior behav)
    {
        if (newBehavs.containsKey(name))
        {
            System.out.println("behavior "+name+
            " defined twice in the same instant");
            newBehavs.put(name, new Nothing());
        }
        newBehavs.put(name, behav);
    }

    public void addToBehav(String name, Instruction inst)
    {
        addToBehav.addElement(new NamedInst(name, inst));
    }

    public Behavior behavNamed(String name)
    {
        Behavior i = (Behavior) behavEnv.get(name);
        if (i==null)
        {
            System.out.println("behavior "+name+" does not exist");
        }
    }
3 Runs

Run instructions are the way to call behaviors. They extends UnaryInstruction and have parameters, whose types are defined in the ParamTypes interface:

```java
public interface ParamTypes
{
    final public byte IN_PARAM    = 1; // input
    final public byte OUT_PARAM   = 2; // output
    final public byte INOUT_PARAM = 3; // inputoutput
    final public byte LOCAL_PARAM = 4; // local
}
```

At creation, an object of class Run copies its actual parameter vector in the array paramList.

The bindParams method checks that there is the same number of actual and formal parameters, and if it is the case, for each parameter, it changes the instruction body into an InputDecl, and OutputDecl, or an InOutDecl, according to the parameter type.

The activation method first tests that the number of runs created during current instant is not exceeded, then it gets the actual behavior and binds the parameters. Finally, it executes the body.

```java
public class Run extends UnaryInstruction implements ParamTypes
{
    public String name;
    private Param [] formalParams = null;
    private String [] actualParams = null;
    /** Number of formal parameters (initially 0). */
    private int len = 0;
    private boolean bindingDone = false;

    public Run(String name) {
        this.name = name; body = new Nothing();
    }

    public Run(String name,Vector paramList) {
        this.name = name;
```
body = new Nothing();
if (paramList != null){
    len = paramList.size();
    actualParams = new String[len];
    paramList.copyInto(actualParams);
}
}

public void reset(){ super.reset(); bindingDone = false; }

final public String toString(){ ... }

public boolean equals(Instruction inst) { ... }

private void bindParams(Machine machine)
{
    int formalLen =
    formalParams = null ? 0 : formalParams.length;
    if (len != formalLen){
        System.out.println("bad arg number (expected: "+formalLen+")");
        body = new Nothing();
        return;
    }
    for (int i = 0; i<len; i++){
        String internal = formalParams[i].name;
        int kind = formalParams[i].kind;
        if (kind == IN_PARAM){
            body = new InputDecl(internal,actualParams[i],body);
        }else if (kind == OUT_PARAM){
            body = new OutputDecl(internal,actualParams[i],body);
        }else if (kind == INOUT_PARAM){
            body = new InOutDecl(internal,actualParams[i],body);
        }
    }
}

final protected byte activation(Machine mach)
{
    RsiMachine machine = (RsiMachine)mach;
    if (!bindingDone){
        if (machine.numberofRuns++ >= machine.maxNumberOfRuns){
            System.out.println(10}
"too much runs in the same instant (max is " + 
machine.maxNumberofRuns + "));
    return STOP;
}
Behavior beh = machine.behavEnv.behavNamed(name);
if (beh == null) return TERM;
bindingDone = true;

    body = (Instruction)beh.body().clone();
    formalParams = beh.formalParams();
    bindParams(machine);
}
return body.activ(machine);
}

4 Objects

Reactive script objects are implemented by the class RsIoObject. A Java object

can be associated to a reactive script object and association is by name: both

objects are referenced by the same name in the reactive interpreter. Inside a

RsIoObject, the keyword this references the Java object associated with. For

example, the external statement "(this.meth(...))" appearing in an object x

calls the method meth of the associated Java object x.

Call redirections to Java objects is implemented by the Selector interface:

public interface Selector
{
    public String select(String methodName, Vector args);
    public Selector getobjectInSelectorContext(String name);
}

A full description of the External class to handle Java calls is in section 6.

4.1 The RsIoObject Class

RsIoObject extends UnaryInstruction and implements Selector. The activation

method sets this to the appropriate Java object before running the body

and restores the old binding on return. The method select retransmits method

calls to the Java object with same name.

Object fields (managed in a Hashtable) can be dynamically added to RsIoObject

instances, using the setField and getField methods.
public class RsiObject extends UnaryInstruction
implements Selector
{
    protected String name;
    protected External externalContext = null;
    protected Hashtable objectVar = new Hashtable();
    public void setExternalContext(External eC)
    {
        externalContext = eC;
    }

    public RsiObject(String name, Instruction body, External eC)
    {
        this.name = name;
        this.body = body;
        externalContext = eC;
    }

    public String toString()
    {
        return "object " + name + " " + body + " end";
    }

    public String select(String methName, Vector args)
    {
        if(methName.equals("getName")) return name;
        if(methName.equals("setField")) return setField(args);
        if(methName.equals("newField")) return newField(args);
        if(methName.equals("removeField")) return removeField(args);
        return externalContext.getExternObject(name).
            select(methName, args);
    }

    public Selector getObjectIdSelectorContext(String name)
    {
        return (Selector)objectVar.get(name);
    }

    final protected byte activation(Machine machine)
    {
        Selector sel = externalContext.getThisContext();
        externalContext.setThisContext(this);
        byte res = body.activ(machine);
        externalContext.setThisContext(sel);
        return res;
    }
}
According to the semantics defined in 1.2, an object x whose body is instruction body, with methods m1,...,mk, is expanded into:

```java
do
  body
  || control m1 by x_m1
  || ...
  || control mk by x_mk
until x-destroy
```

The following methods are used at parsing to build the corresponding Rsiobject instruction:

```java
private Instruction objectShell(String name, Instruction body) {
  return new Rsiobject(name, new Until(name + "-destroy", body));
}
```

```java
public Instruction object(String name, Instruction body, Vector list) {
  Instruction res = body;
  for (int i = 0; i < list.size(); i++) {
    Run run = (Run) list.elementAt(i);
    res = new Merge(res, new Control(name + "-" + run.name, run));
  }
  return objectShell(name, res);
}
```

```java
public Instruction object(String name, Instruction body) {
  return objectShell(name, body);
}
```

### 5 Machines

We first define the class Rsimachine which extends the class Machine of Sugar-cubes, then we use it to define Interpreter which is the main class of Rs-i-Java.

#### 5.1 Rsimachine

Rsimachine extends Machine and has an environment of behaviors and a counter numberOfRuns to control how many runs are created at each instant (the max
value is set to 500). Activation means to reset numberOfRuns to 0, to transfer behaviors, and to activate the machine.

```java
public class RsiMachine extends Machine {
    public BehavEnv behavEnv = new BehavEnv();

    protected final int maxNumberOfRuns = 500;
    protected int numberOfRuns = 0;

    protected byte activation(Machine machine) {
        numberOfRuns = 0;
        behavEnv.transferBehavs();
        return super.activation(machine);
    }
}
```

### 5.2 Interpreter

Interpreter extends RsiMachine and defines two flags next and runInterp. To set next to true forces immediate reaction for the next instant, as soon as current instant is over. To set runInterp to true forces the interpreter to run the program.

The init method initializes the keyboard and the instantPrompt method prompts the current instant number.

At creation, the extern node System is created to be able to print messages (see section 6.2).

Activation creates a new parser to parse the input and then calls the oneStep method which activates the machine.

The main loop creates an interpreter and activates it for ever.

```java
public class Interpreter extends ExtendedRsiMachine {
    protected External externalContext = null;
    public External getExternalContext() {return externalContext;}

    protected boolean next = false;
    public void next() { next = true; }

    protected boolean runInterp = false;
    public void runInterp() { runInterp = true; }

    public Interpreter()
```
{  
    externalContext = new External(this);
    new ExternNode("System", externalContext);
    init();
    instantPrompt();
}

protected void instantPrompt(){
    System.out.print(instant + ": "); System.out.flush();
}
...
protected byte activation(Machine machine)
{
    do{
        if (!next) parseEntry();
        oneStep();
    } while (next);
    return STOP;
}

protected void oneStep()
{
    if (next || runInterp){
        next = runInterp = false;
        super.activation(this);
        instantPrompt();
    }
}

public static void main (String argv[])
{
    Interpreter interp = new Interpreter();
    while (true) { interp.activation(interp); }
}
}

We end this section with the next instruction which forces the interpreter to immediately execute the next instant as soon as the current one is over.

Next is like Nothing but it sets the next flag of the executing machine.

public class Next extends Instruction
{
    final public String toString() { return "next"; }
}
final protected byte activation(Machine machine)
{
    ((Interpreter)machine).next();
    return TERM;
}

6 External Instructions

Reactive Scripts “external statements” are used to interact with the external system (for data management, graphic interactions, etc.). To implement this in Java, we build a mechanism that allows Java objects to register themselves as external objects on which reactive scripts can invoke methods through external calls. Please remark that we do not take advantage of the new features of Java 1.1 such as introspection, to be able to use our package on Java 1.0 Virtual Machines.

External statements are enclosed between braces and use a reduced Java-like syntax, as in the following examples:

- {123} returns a number.
- {false} or {true} returns a boolean.
- {"string"} returns a string.
- {object.method([argument list])} returns the result of a method invocation on a Java registered object.

Invocation of a method on a registered object return one of the three basic type results: an integer, a string or a boolean. The following examples show how external statements are used:

- loop { registeredObjectName.getIterationNumber() } times stop end
  The external evaluation return an integer which is the number of loop iterations.

- await {object.getAnEventName()}
  The external evaluation returns a string which is the name of the event to await.

- if {rectangle.overlaps(10,10,100,100)} then...else...end
  The external evaluation returns a boolean for the test.
6.1 External class

Each instance of Interpreter has a field which is an instance of External and is used as an interface between reactive scripts and external Java objects. When an external statement is encountered, the string between braces is passed to this object as a parameter of the parseExpression method. While parsing, when an identifier is found, the External object checks if this identifier references a registered object (Java objects that implement the Selector interface have to register themselves to the External instance through the register method) and if so, calls the select method on this registered object with the method name and its parameters as arguments. Each implementation of the select method is responsible for the correct redirection to Java methods.

public class External implements Selector
{
    private Hashtable objEnv = new Hashtable();
    public Interpreter currentInterp = null;

    public External(Interpreter anInterp)
    {
        currentInterp = anInterp;
        register("External",this);
    }

    public void setThisContext(Selector sel0bj){
        if(sel0bj!=null) objEnv.put("this",sel0bj);
        else objEnv.remove("this");
    }

    public Selector getThisContext(){
        return (Selector)objEnv.get("this");
    }

    public Selector getExternObject(String aName){
        return (Selector)objEnv.get(aName);
    }

    public boolean unregister(String ObjectName)
    {
        if(ObjectName.equals("External")) {
            System.out.println(  
                "Unable to remove the External object!");
            return false;
        }
    }
}

17
objEnv.remove(objectName);
    return true;
}

public boolean register(String objectName, Selector selObj)
{
    if(objectName.equals("this"))
    {
        System.out.println(
            "no object can be registered as \"this\"");
        return false;
    }
    Object o = objEnv.get(objectName);
    if (o==null) objEnv.put(ObjectName,selObj);
    else
        System.err.println("already registered object: ");
        return false;
    return true;
}
........

6.2 Extern Node

ExternNode is a typical example of External use to interact with Java objects. ExternNode implements Selector and has two methods, print and println. It must be subclassed to implement other methods. The System object defined in Interpreter is an instance of it (thus, System.out.println("") and System.out.print("") calls are correct).

public class ExternNode
    implements Selector
{
    protected String name;
    public ExternNode(String name){this.name = name;}

    public String select(String methodName, Vector args)
    {
        if(methodName.equals("print")){
            if(args.isEmpty()) { System.out.println(""); return null; }
            else System.out.println((String) args.firstElement());
            return (String) args.firstElement();
    }
if(methodName.equals("println"))
    if(args.isEmpty()) { System.out.println(""); return null; }
    else System.out.println((String)args.firstElement());
    return (String)args.firstElement();
}
System.out.println("unknown method: "+methodName);
return null;
}

public Selector getobjectInSelectorContext(String name)
{
    if(name.equals("out")) return this;
    else return null;
}

6.3 If Instruction
The boolean test If extends BinaryInstruction and condition evaluation is
performed by calling the parseBooleanExpression method of External.

public class If extends BinaryInstruction
{
    public String condition;
    private boolean condEvaluated = false;
    private boolean value;

    public If(String cond, Instruction t, Instruction e)
    {
        condition = cond;
        left = t;
        right = e;
    }

    public void reset(){ super.reset(); condEvaluated = false; }

    final public String toString(){
        return "if "+condition+" then "+left+" else "+right+" end";
    }

    final protected byte activation(Machine machine)
    {
        if (!condEvaluated){
            condEvaluated = true;
            
            return -1;
        }
    }
value = External.parseBooleanExpression(condition);
}
return value ? left.activ(machine) : right.activ(machine);
}

7 Grammar

The grammar is given in CUP format (see http://www.cc.gatech.edu/gvu/-people/Faculty/hudson/java_cup) which implements YACC in Java.

statement::=
  no_parallel_statement:s { RESULT.inst = s.inst; }
  | parallel_statement:s   { RESULT.inst = s.inst; }
  |

no_parallel_statement::=
  sequence:s { RESULT.inst = s.inst; }
  | stat:s   { RESULT.inst = s.inst; }
  |

parallel_statement::=
  no_parallel_statement:s1 PAR no_parallel_statement:s2
  { RESULT.inst = new Merge(s1.inst,s2.inst); }
  | no_parallel_statement:s1 PAR parallel_statement_list:s2
  { RESULT.inst = new Merge(s1.inst,s2.inst); }
  |

parallel_statement_list::=
  no_parallel_statement:s1 PAR no_parallel_statement:s2
  { RESULT.inst = new Merge(s1.inst,s2.inst); }
  | parallel_statement_list:s1 PAR no_parallel_statement:s2
  { RESULT.inst = new Merge(s1.inst,s2.inst); }
  |

sequence::=
  pure_sequence:s   { RESULT.inst = s.inst; }
  | stat:s SEMI   { RESULT.inst = s.inst; }
  | pure_sequence:s SEMI { RESULT.inst = s.inst; }
  |

pure_sequence::=
  stat:s1 SEMI stat:s2
  { RESULT.inst = new Seq(s1.inst,s2.inst); }
  | pure_sequence:s1 SEMI stat:s2
\{ \text{RESULT.inst = new Seq(s1.inst,s2.inst); } \}
;

\text{stat::=}
\text{\hspace{1em}NOTHING} \quad \{ \text{RESULT.inst = new Nothing(); } \}
\text{\hspace{1em}STOP} \quad \{ \text{RESULT.inst = new Stop(); } \}
\text{\hspace{1em}HALT} \quad \{ \text{RESULT.inst = new Loop(new Stop()); } \}
\text{\hspace{1em}NEXT} \quad \{ \text{RESULT.inst = new Next(); } \}
\text{\hspace{1em}LPAREN \text{statement:s RPAREN} \quad \{ \text{RESULT.inst = s.inst; } \}}
\text{\hspace{1em}break:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}extcode:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}if:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}loop:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}repeat:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}event_declarations:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}in_declarations:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}out_declarations:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}inout_declarations:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}generate:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}when :s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}until:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}wait:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}control:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}behavior:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}add:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}run:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}send:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}object:s} \quad \{ \text{RESULT.inst = s.inst; } \}
\text{\hspace{1em}destroy:s} \quad \{ \text{RESULT.inst = s.inst; } \}
;

\text{/* java code */}
\text{extcode::= EXTCODE:e \{ \text{RESULT.inst = new ExternAtom(e.str_val); } \}}
;

\text{/* test statement */}
\text{if::= \begin{array}{l}
\text{IF EXTCODE:cond then_branch:t else_branch:e END}
\end{array}
\{ \text{RESULT.inst = new If(cond.str_val,t.inst,e.inst); } \}
;
\text{then_branch::=}
\text{/* Empty */ \{ \text{RESULT.inst = new Nothing(); } \}
\text{\hspace{1em}THEN \text{statement:s} \{ \text{RESULT.inst = s.inst; } \}}
\}
\text{else_branch::=}
\text{/* Empty */ \{ \text{RESULT.inst = new Nothing(); } \}
\text{\hspace{1em}ELSE \text{statement:s} \{ \text{RESULT.inst = s.inst; } \}}
\}

21
/* event configurations */
config:=
  basic_config:c {} RESULT.conf = c.conf; }
| config:c1 OR basic_config:c2
  {} RESULT.conf = new OrConfig(c1.conf,c2.conf); }
| config:c1 AND basic_config:c2
  {} RESULT.conf = new AndConfig(c1.conf,c2.conf); }
;
basic_config:=
  IDENTIFIER :i {} RESULT.conf = new PosConfig(i.str_val); }
| NOT IDENTIFIER:i {} RESULT.conf = new NegConfig(i.str_val); }
| LPAREN config:c RPAREN {} RESULT.conf = c.conf; }
;
/* event based statements */
await:= AWAIT config:c {} RESULT.inst = new Await(c.conf); }
;
EVENT identifier_list:1 IN statement:s
  {} RESULT.inst = semActions.eventDecl(l.list,s.inst); }
;
in_declaration:=
  INPUT IDENTIFIER:intern IS IDENTIFIER:extern IN statement:s
  {} RESULT.inst = 
    new InputDecl(intern.str_val,extern.str_val,s.inst); }
;
out_declaration:=
  OUTPUT IDENTIFIER:intern IS IDENTIFIER:extern IN statement:s
  {} RESULT.inst = 
    new OutputDecl(intern.str_val,extern.str_val,s.inst); }
;
inout_declaration:=
  INPUTOUTPUT IDENTIFIER:intern IS IDENTIFIER:extern IN statement:s
  {} RESULT.inst = 
    newInOutDecl(intern.str_val,extern.str_val,s.inst); }
;
generate:= GENERATE IDENTIFIER:i
  {} RESULT.inst = new Generate(i.str_val); }
;
when:= WHEN config:c then_branch:t else_branch:e

{: RESULT.inst = new When(c.conf, t.inst, e.inst); :} ;

until::= DO statement::s UNTIL config::c actual::a
{: RESULT.inst = new Until(c.conf, s.inst, a.inst); :} ;

actual::=  
/*Empty*/{: RESULT.inst = new Nothing(); :}
| ACTUAL statement::s END {: RESULT.inst = s.inst; :} ;

control::= CONTROL statement::s BY IDENTIFIER::i
{: RESULT.inst = new Control(i.str_val, s.inst); :} ;

*/ identifier lists */

identifier_list::=
IDENTIFIER::i {: RESULT.list.addElement(i.str_val); :}
| identifier_list::1 COLON IDENTIFIER::i
{: RESULT.list = semActions.addToIdentifierList(l.list, i.str_val); :} ;

*/ loops */

loop::= LOOP statement::s END
{: RESULT.inst = new EventDecl("_break", new Until("_break",  
 new Loop(s.inst))); :}

break::= BREAK
{: RESULT.inst = new Seq(new Generate("_break"), new Stop()); :} ;

repeat::= LOOP EXITCODE::e TIMES statement::s END
{: RESULT.inst = new EventDecl("_break", new Until("_break",  
 new Repeat(External.parseIntExpression(e.str_val), s.inst))); :} ;

/*********************** behaviors *******************/

behavior::=  
BEHAVIOR IDENTIFIER::i behavior_interface::l statement::s END
{: RESULT.inst = new Behavior(i.str_val, l.list, s.inst); :} ;

behavior_interface::= /* Empty */{: RESULT.list = null; :}
| interface_list::1 {: RESULT.list = l.list; :} ;

interface_list::=

23
interface:i {: RESULT.list = i.list; ;}
| interface_list:1 interface:i
{: RESULT.list = Param.concat(l.list,i.list); ;}
;
interface::=
    IN identifier_list:1 SEMI
    {: RESULT.list = Param.convert(ParamTypes.IN_PARAM,l.list); ;}
| OUT identifier_list:1 SEMI
    {: RESULT.list = Param.convert(ParamTypes.OUT_PARAM,l.list); ;}
| INOUT identifier_list:1 SEMI
    {: RESULT.list = Param.convert(ParamTypes.INOUT_PARAM,l.list); ;}
;
call::=
    IDENTIFIER:i {: RESULT.inst = new Run(i.str_val); ;}
| IDENTIFIER:i LPAREN identifier_list:1 RPAREN
    {: RESULT.inst = new Run(i.str_val,l.list); ;}
;
run::= RUN call:c {: RESULT.inst = c.inst; ;} ;

add::= ADD statement:s TO IDENTIFIER:i
    {: RESULT.inst = new AddTo(i.str_val,s.inst); ;} ;

/**************************** objects ****************************/
object::= 
    OBJECT IDENTIFIER:o statement:s END
    {: RESULT.inst = semActions.object(o.str_val,s.inst); ;}
| OBJECT IDENTIFIER:o object_behavior:s METHODS object_methods:1 END
    {: RESULT.inst = semActions.object(o.str_val,s.inst,l.list); ;}
;
object_behavior::=
    /* Empty */ {: RESULT.inst = new Nothing(); ;}
| statement:s {: RESULT.inst = s.inst; ;}
;
object_methods::=
    call:c {: RESULT.list.addElement(c.inst); ;}
| object_methods:1 call:c
    {: l.list.addElement(c.inst); RESULT.list = l.list; ;}
;
send::= SEND IDENTIFIER:m TO IDENTIFIER:o
    {: RESULT.inst = new Generate(o.str_val + "-" + m.str_val); ;} ;

8 Conclusion

We have implemented the RSI-JAVA interpreter using SUGARCUBES in a straightforward way. Several sub-classes of Interpreter have also been defined mainly to disconnect the keyboard from the interpreter, to be able to enter new commands and new scripts while the interpreter is running.

We plan to add to the present RSI-JAVA implementation the following features:

- The possibility to “freeze” an object that is to get, at the end of the current instant, a script which represents “what remains to be done” for it.
- The way to send a script to a remote RSI-JAVA interpreter through the network. In conjunction with the “freeze” operator, this would give us the way to migrate reactive scripts.
- The possibility to “compile” a script to get sequential Java code.

The RSI-JAVA interpreter is used to implement the icobj programming demo available at http://www.inria.fr/meije/rc/webicobj and described in the companion paper:

- The SUGARCUBES Tool Box - Icobj Programming Implementation

References

Index

BehavEnv, 7
Behavior, 6

External, 17, 18
ExternNode, 18

if, 19
Interpreter, 14, 17, 18

NamedInst, 7
Next, 15

ParamTypes, 9

RsiMachine, 13
RsiObject, 11
Run, 9

Selector, 11, 17

UnaryInstruction, 11