Junior Rewrite Semantics*

Frédéric Boussinot† Jean-Ferdy Susini‡

Draft - October 9, 2000

Abstract

This paper describes the basic formal semantics of **Junior**, a formalism for reactive programming in Java. Semantics consists in rewriting rules given for **Junior** reactive instructions.

1 Introduction

Junior[2] is a formalism for reactive programming in Java. This paper describes a formal semantics for it. One adopts some simplifications with **Jr**, the **Junior** API[3]; the followings are not considered:

- Events identifiers others than constant strings. Thus, no identifiers nor wrappers are considered, only strings.
- The possibility to dynamically add new programs in machines. Thus, the *add* method of *Machine* is not considered.
- Simplified forms of **Jr** constructors, where $Jr.Nothing()$ replaces an absent parameter.
- Means for interfacing with Java, except atoms. Thus, the *Link* instruction is not considered.

One first defines the abstract syntax of **Junior**; then, rewriting rules are given for each reactive instruction.

*With support from France-Telecom R&D, and from EC (PING IST Project)
†EMP/CMA-INRIA
‡EMP/CMA-INRIA

2 Syntax

Formal semantics uses abstract syntax instead of concrete one.

2.1 Abstract Syntax

In the abstract syntax, a is an action, t and u are terms, n is an integer, S is an event, and C is a configuration.

- For the following instructions, abstract syntax coincides with concrete one:
  - $Nothing$
  - $Stop$
  - $Atom(a)$
  - $Loop(t)$
  - $Repeat(n, t)$
  - $Generate(S)$
  - $Await(C)$
  - $When(C, t, u)$
  - $Control(S, t)$
  - $Seq(t, u)$
- In abstract syntax, the Par operator holds the termination flags (defined later in section 2.3) of its two branches (α is the termination flag of t and β is the one of u):
  - $Par_{\alpha,\beta}(t, u)$
- Abstract syntax adds a new form to the preemption $Until$ operator. In this new form, written $Until*$, the body has already reacted but decision of actual preemption is still pendant:
  - $Until(C, t, u)$
  - $Until * (C, t, u)$
- An auxiliary information is added to local event declarations, to store local event values:
  - $Local - (S, t)$
  - $Local + (S, t)$

  In these terms, sign $+$ indicates that the local event is generated, and sign $-$ that it is not.
- As with $Until$, abstract syntax adds a new form to the $Freezable$ operator. In this new form, written $Freezable*$, production of the residual is still pendant:

- $Freezable(S, t)$
- $Freezable * (S, t)$

- One defines a new top-level instruction $Instant$ to cyclically run an instruction t while suspended, and to detect the end of the current instant:
  - $Instant(t)$
- The abstract syntax for a reactive machine with program t is:
  - $ExecContext(t)$

2.2 Translation from Concrete to Abstract Syntax

The translation function tr from concrete to abstract syntax is recursively defined as follows:

- $tr(Jr.Nothing()) = Nothing$
- $tr(Jr.Stop()) = Stop$
- $tr(Jr.Atom(a)) = Atom(a)$
- $tr(Jr.Seq(t, u)) = Seq(tr(t), tr(u))$
- $tr(Jr.Par(t, u)) = Par_{SUSP,SUSP}(tr(t), tr(u))$
- $tr(Jr.Loop(t)) = Loop(tr(t))$
- $tr(Jr.Repeat(n, t)) = Repeat(n, tr(t))$
- $tr(Jr.Generate(S)) = Generate(S)$
- $tr(Jr.Await(C)) = Await(C)$
- $tr(Jr.When(C, t, u)) = When(C, tr(t), tr(u))$
- $tr(Jr.Until(C, t, u)) = Until(C, tr(t), tr(u))$
- $tr(Jr.Control(S, t)) = Control(S, tr(t))$
- $tr(Jr.Local(S, t)) = Local - (S, tr(t))$
- $tr(Jr.Freezable(S, t)) = Freezable(S, tr(t))$
- $tr(Jr.Machine(t)) = ExecContext(tr(t))$

In the abstract parallel operator, the two termination flags are initially set to $SUSP$ (see 2.3).Iin local declarations, the local event is initially not generated (corresponding to sign $-$).

2.3 Format of Rewritings

The basic semantics of **Junior** has the standard format of *conditional rewriting rules*[4]. It is called REWRITE. One writes:

$$t, E \xrightarrow{\alpha} t', E'$$

which means that:

- t' is the instruction which remains to be executed after executing t in the environment E. t' is called the *residual* of t, and one says that t *rewrites* in t'.
- E' is the new environment after execution of t.
- α is a termination flag with 3 possible values:
  - `TERM` means that execution is terminated for the current instant and that nothing remains to do for the next instant.
  - `STOP` means that execution is terminated for the current instant but that something remains to do at next instant.
  - `SUSP` means that execution is not terminated for the current instant and thus must be resumed during it.

2.4 Environments

An environment E is made of the following components:

- A set containing present events.
- A boolean flag $eoi(E)$ which is true if the end of the current instant has been decided, and false otherwise.
- A boolean flag $move(E)$ which is set to true to indicate that some change has appeared in the system; in this case, the end of the current instant must be dalayed to let the system possibility to react to this change.
- A table used to store the instructions which are frozen during the instant.

The following notations are defined:

- To note that an event S is present in E, one simply writes $S \in E$.
- $E + S$ is the environment equal to E except that event S is added to it; $E - S$ is equal to E except that S is removed from it. $E[move = b]$ is equal to E except that flag *move* is set to b. $E[eoi = b]$ is equal to E except that flag *eoi* is set to b.
- $E/F[S]$ is equal to $E + S$ if $S \in F$, and is equal to $E - S$ otherwise; $E/F[S]$ is thus equal to E, except for S which is determined by F.
- $\gamma(\alpha, \beta)$ equals $SUSP$ if either α or β is equal to $SUSP$; it equals $TERM$ if both α and β are equals to $TERM$; it equals $STOP$ in all other cases; actually, it is defined by the array:

	SUSP	STOP	TERM
SUSP	SUSP	SUSP	SUSP
STOP	SUSP	STOP	STOP
TERM	SUSP	STOP	TERM

- $\delta_1(\alpha, \beta)$ equals $SUSP$ if α is $STOP$ and β is $STOP$ or $TERM$; it equals α otherwise. $\delta_2(\alpha, \beta)$ equals $SUSP$ if β is $STOP$ and α is $STOP$ or $TERM$; it equals β otherwise.
- $E\{S := t\}$ associates the frozen instruction t to event S; $E\{S\}$ is the frozen instruction associated to S.

3 Semantics of Basic Statements

3.1 Nothing

Nothing immediately terminates and does nothing. The rule is:

$$Nothing, E \xrightarrow{\texttt{TERM}} Nothing, E \tag{1}$$

3.2 Stop

The *Stop* statement stops execution for the current instant, and nothing remains to be done at next instant:

$$Stop, E \xrightarrow{\texttt{STOP}} Nothing, E \tag{2}$$

3.3 Atoms

An atom immediately terminates after performing an action. The rule is:

$$Atom(a), E \xrightarrow{\texttt{TERM}} Nothing, E \tag{3}$$

with the side-effect of executing atomically action a.

3.4 Sequence

The sequence is defined by two rules, depending on the termination of the first branch.

If the first branch terminates, then the second one is immediately run:

$$\frac{t, E \xrightarrow{\texttt{TERM}} t', E' \qquad u, E' \xrightarrow{\alpha} u', E''}{Seq(t, u), E \xrightarrow{\alpha} u', E''} \tag{4}$$

If the first branch is not terminated, then so is the sequence:

$$\frac{t, E \xrightarrow{\alpha} t', E' \qquad \alpha \neq \texttt{TERM}}{Seq(t, u), E \xrightarrow{\alpha} Seq(t', u), E'} \tag{5}$$

3.5 Parallelism

Both branches are suspended

If both branches of the parallel operator are suspended (which is the initial situation at each instant), then execution can nondeterministically chooses to execute one of them, setting the *move* flag to indicate that end of instant must be postponed in order to let the other branch a chance to be executed. The two corresponding rules are:

$$\frac{t, E \xrightarrow{\alpha} t', E'}{Par_{\mathtt{SUSP},\mathtt{SUSP}}(t, u), E \xrightarrow{\mathtt{SUSP}} Par_{\alpha,\mathtt{SUSP}}(t', u), E'[move = true]} \tag{6}$$

$$\frac{u, E \xrightarrow{\beta} u', E'}{Par_{\mathtt{SUSP},\mathtt{SUSP}}(t, u), E \xrightarrow{\mathtt{SUSP}} Par_{\mathtt{SUSP},\beta}(t, u'), E'[move = true]} \tag{7}$$

If both branches of the parallel operator are suspended, execution can also execute both. The rule which executes the first branch, then the second, is called **Merge**:

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad u, E' \xrightarrow{\beta} u', E''}{Par_{\mathtt{SUSP},\mathtt{SUSP}}(t, u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t', u'), E''} \tag{8}$$

The second rule which executes the second branch, then the first one, is called **InvMerge**:

$$\frac{u, E \xrightarrow{\beta} u', E' \quad t, E' \xrightarrow{\alpha} t', E''}{Par_{\mathtt{SUSP},\mathtt{SUSP}}(t, u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t', u'), E''} \tag{9}$$

Only one branch is suspended

If only one branch is suspended, then it is simply run (and, in this case, the *move* flag is left unchanged):

$$\frac{\beta \neq \mathtt{SUSP} \quad t, E \xrightarrow{\alpha} t', E'}{Par_{\mathtt{SUSP},\beta}(t, u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t', u), E'} \tag{10}$$

$$\frac{\alpha \neq \mathtt{SUSP} \quad u, E \xrightarrow{\beta} u', E'}{Par_{\alpha,\mathtt{SUSP}}(t, u), E \xrightarrow{\gamma(\alpha,\beta)} Par_{\delta_1(\alpha,\beta),\delta_2(\alpha,\beta)}(t, u'), E'} \tag{11}$$

Note that, in all the rules of Par, production of a flag different from `SUSP` is only possible when both branches have also produced a flag different from `SUSP`; this reflects the synchronous characteristics of the parallel operator which, at each instant, executes its two branches.

3.6 Loop

A loop executes its body and rewrites in a sequence if it does not terminate immediately:

$$\frac{t, E \xrightarrow{\alpha} t', E' \qquad \alpha \neq \texttt{TERM}}{Loop(t), E \xrightarrow{\alpha} Seq(t', Loop(t)), E'} \tag{12}$$

When the loop body terminates immediately, the loop is restarted:

$$\frac{t, E \xrightarrow{\texttt{TERM}} t', E' \qquad Loop(t), E \xrightarrow{\alpha} u, E''}{Loop(t), E \xrightarrow{\alpha} u, E''} \tag{13}$$

3.7 Repeat

Repeat terminates immediately if the counter is less or equal to zero:

$$\frac{n \leq 0}{Repeat(n, t), E \xrightarrow{\texttt{TERM}} Nothing, E} \tag{14}$$

Otherwise, the semantics is the one of a sequence:

$$\frac{n > 0 \qquad Seq(t, Repeat(n-1, t)), E \xrightarrow{\alpha} u, E'}{Repeat(n, t), E \xrightarrow{\alpha} u, E'} \tag{15}$$

4 Event Statements

4.1 Configurations

A configuration is either:

- A positive configuration, that is an event S.
- A negative configuration, that is the negation *not* C of a configuration.
- A conjunction C_1 *and* C_2 of two configurations.
- A disjunction C_1 *or* C_2 of two configurations.

Two functions $fixed$ and $eval$ are defined for configurations. A configuration can be evaluated (function $eval$) only when it is fixed (function $fixed$):

Fixed

$fixed(C, E)$ is true when configuration C can be evaluated in the environment E:

- $fixed(S, E) \equiv S \in E \; or \; eoi(E)$
- $fixed(not\; C, E) \equiv fixed(C, E)$
- $$\begin{array}{rcl} fixed(C_1 \; and \; C_2, E) & \equiv & fixed(C_1, E) \; and \; fixed(C_2, E) \\ & or & fixed(C_1, E) \; and \; eval(C_1, E) = false \\ & or & fixed(C_2, E) \; and \; eval(C_2, E) = false \end{array}$$

- $$\begin{array}{rcl} fixed(C_1 \ or \ C_2, E) & \equiv & fixed(C_1, E) \ and \ fixed(C_2, E) \\ & or & fixed(C_1, E) \ and \ eval(C_1, E) \\ & or & fixed(C_2, E) \ and \ eval(C_2, E) \end{array}$$

Note that in the basic case of an event S, $fixed(S, E)$ is true if S is present or if the end of instant is set; this last case means that S is absent.

Eval

$eval(C, E)$ returns the value of configuration C in the environment E:

- $eval(S, E) \equiv S \in E$
- $eval(not \ C, E) \equiv not \ eval(C, E)$
- $eval(C_1 \ and \ C_2, E) \equiv eval(C_1, E) \ and \ eval(C_2, E)$
- $eval(C_1 \ or \ C_2, E) \equiv eval(C_1, E) \ or \ eval(C_2, E)$

Auxiliary Functions

Three auxiliary functions are also defined:

- $sat(C, E) \equiv fixed(C, E) \ and \ eval(C, E)$
- $unsat(C, E) \equiv fixed(C, E) \ and \ eval(C, E) = false$
- $unknown(C, E) \equiv fixed(C, E) = false$

Note that $unknown(C, E)$ is true if and only if $sat(C, E)$ and $unsat(C, E)$ are both false. Note also that in the basic case of an event S, one has:

- $sat(S, E)$ = true means that S is in E: S is present;
- $unsat(S, E)$ = true means that S is not in E and that $eoi(E)$ is true: S is absent.

4.2 Generate

A *Generate* statement adds the generated event in the environment and immediately terminates:

$$Generate(S), E \xrightarrow{\texttt{TERM}} Nothing, (E + S)[move = true] \tag{16}$$

4.3 Events Tests

The "then" branch is executed if the configuration is satisfied; execution is immediate if satisfaction occurs before the end of the current instant, and is delayed to the next instant otherwise:

$$\frac{sat(C, E) \quad eoi(E) = false \quad t, E \xrightarrow{\alpha} t', E'}{When(C, t, u), E \xrightarrow{\alpha} t', E'} \tag{17}$$

$$\frac{sat(C,E) \quad eoi(E) = true}{When(C,t,u), E \xrightarrow{\texttt{STOP}} t, E} \tag{18}$$

The "else" branch is chosen if the configuration is not satisfied; execution is immediate if unsatisfaction occurs before the end of the current instant, and is delayed to the next instant otherwise:

$$\frac{unsat(C,E) \quad eoi(E) = false \quad u, E \xrightarrow{\alpha} u', E'}{When(C,t,u), E \xrightarrow{\alpha} u', E'} \tag{19}$$

$$\frac{unsat(C,E) \quad eoi(E) = true}{When(C,t,u), E \xrightarrow{\texttt{STOP}} u, E} \tag{20}$$

Note that the two previous rules returning $STOP$ when $eoi(E)$ is false basically forbid immediate reaction to events absences.

The test is suspended if the configuration is unknown:

$$\frac{unknown(C,E)}{When(C,t,u), E \xrightarrow{\texttt{SUSP}} When(C,t,u), E} \tag{21}$$

4.4 Await

Await terminates if the configuration is satisfied; termination is immediate if satisfaction occurs before the end of the current instant, and is delayed to the next instant otherwise :

$$\frac{sat(C,E) \quad eoi(E) = false}{Await(C), E \xrightarrow{\texttt{TERM}} Nothing, E} \tag{22}$$

$$\frac{sat(C,E) \quad eoi(E) = true}{Await(C), E \xrightarrow{\texttt{STOP}} Nothing, E} \tag{23}$$

Await stops if the configuration is unsatisfied:

$$\frac{unsat(C,E)}{Await(C), E \xrightarrow{\texttt{STOP}} Await(C), E} \tag{24}$$

Await is suspended if the configuration is unknown:

$$\frac{unknown(C,E)}{Await(C), E \xrightarrow{\texttt{SUSP}} Await(C), E} \tag{25}$$

4.5 Control

The body is executed if the controlling event is present:

$$\frac{sat(S,E) \quad t, E \xrightarrow{\alpha} t', E'}{Control(S,t), E \xrightarrow{\alpha} Control(S,t'), E} \tag{26}$$

Control stops if the event is absent:

$$\frac{unsat(S,E)}{Control(S,t),E \overset{\texttt{STOP}}{\longrightarrow} Control(S,t),E} \tag{27}$$

Control is suspended if the event is unknown:

$$\frac{unknown(S,E)}{Control(S,t),E \overset{\texttt{SUSP}}{\longrightarrow} Control(S,t),E} \tag{28}$$

4.6 Until

Until behaves as the body if it does not stop:

$$\frac{t,E \overset{\alpha}{\longrightarrow} t',E' \qquad \alpha \neq \texttt{STOP}}{Until(C,t,u),E \overset{\alpha}{\longrightarrow} Until(C,t',u),E'} \tag{29}$$

If the body stops, *Until* behaves as the auxiliary instruction $Until*$ (considered in 4.7):

$$\frac{t,E \overset{\texttt{STOP}}{\longrightarrow} t',E' \qquad Until*(C,t',u),E' \overset{\alpha}{\longrightarrow} v,E''}{Until(C,t,u),E \overset{\alpha}{\longrightarrow} v,E''} \tag{30}$$

Note that the body t is executed in both rules; preemption of *Until* is said to be *weak*, by contrast with the strong preemption used in the synchronous approach, which basically implies instantaneous reaction to absence.

4.7 Until*

The rules for $Until*$ are the following:

The handler is immediately executed if the configuration is satisfied before the end of instant:

$$\frac{sat(C,E) \qquad eoi(E) = false \qquad u,E \overset{\alpha}{\longrightarrow} u',E'}{Until*(C,t,u),E \overset{\alpha}{\longrightarrow} u',E'} \tag{31}$$

$Until*$ stops and rewrites in the handler, if the configuration is satisfied while end of instant is true:

$$\frac{sat(C,E) \qquad eoi(E) = true}{Until*(C,t,u),E \overset{\texttt{STOP}}{\longrightarrow} u,E} \tag{32}$$

$Until*$ stops and rewrites in *Until*, if the configuration is unsatisfied:

$$\frac{unsat(C,E) \qquad eoi(E) = true}{Until*(C,t,u),E \overset{\texttt{STOP}}{\longrightarrow} Until(C,t,u),E} \tag{33}$$

$Until*$ is suspended while the configuration is unknown:

$$\frac{unknown(C,E)}{Until*(C,t,u),E \overset{\texttt{SUSP}}{\longrightarrow} Until*(C,t,u),E} \tag{34}$$

4.8 Local

The local event is not generated in $Local-$ and present in $Local+$. The local event is set to the appropriate value before body execution, and it is saved after. The value of the event is always left unchanged in the external environment.

If the body suspends, then the value of the local event value is stored in the produced term:

$$\frac{t, E-S \xrightarrow{\texttt{SUSP}} t', E' \quad S \notin E'}{Local-(S,t), E \xrightarrow{\texttt{SUSP}} Local-(S,t'), E'/E[S]} \tag{35}$$

$$\frac{t, E-S \xrightarrow{\texttt{SUSP}} t', E' \quad S \in E'}{Local-(S,t), E \xrightarrow{\texttt{SUSP}} Local+(S,t'), E'/E[S]} \tag{36}$$

$$\frac{t, E+S \xrightarrow{\texttt{SUSP}} t', E'}{Local+(S,t), E \xrightarrow{\texttt{SUSP}} Local+(S,t'), E'/E[S]} \tag{37}$$

When the body terminates or stops, then the local event is reset for the next instant:

$$\frac{t, E-S \xrightarrow{\alpha} t', E' \quad \alpha = \texttt{TERM} \;\; or \;\; \alpha = \texttt{STOP}}{Local-(S,t), E \xrightarrow{\alpha} Local-(S,t'), E'/E[S]} \tag{38}$$

$$\frac{t, E+S \xrightarrow{\alpha} t', E' \quad \alpha = \texttt{TERM} \;\; or \;\; \alpha = \texttt{STOP}}{Local+(S,t), E \xrightarrow{alpha} Local-(S,t'), E'/E[S]} \tag{39}$$

4.9 Freezable

The semantics of $Freezable$ is close to the one of $Until$. $Freezable$ behaves as the body if it does not stops:

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad \alpha \neq \texttt{STOP}}{Freezable(S,t), E \xrightarrow{\alpha} Freezable(S,t'), E'} \tag{40}$$

If the body stops, $Freezable$ behaves as the auxiliary instruction $Freezable*$ (considered in 4.10):

$$\frac{t, E \xrightarrow{\texttt{STOP}} t', E' \quad Freezable*(S,t'), E' \xrightarrow{\alpha} v, E''}{Freezable(S,t), E \xrightarrow{\alpha} v, E''} \tag{41}$$

Note that the body t is executed in both rules; As $Until$, $Freezable$ performs weak preemption.

4.10 Freezable*

The rules for $Freezable*$ are the following:

The instruction immediately terminates if the freezing event is present before the end of instant:

$$\frac{S \in E \quad eoi(E) = false}{Freezable*(S,t), E \xrightarrow{\texttt{TERM}} Nothing, E\{S := Par(t, E\{S\})\}} \tag{42}$$

Note that the residual instruction t is put in parallel in the frozen instructions table.

The instruction terminates at next instant if the freezing event is present while the end of instant is set:

$$\frac{S \in E \quad eoi(E) = true}{Freezable*(S,t), E \xrightarrow{\texttt{STOP}} Nothing, E\{S := Par(t, E\{S\})\}} \tag{43}$$

$Freezable*$ stops and rewrites in $Freezable$, if the freezing event is absent:

$$\frac{S \notin E \quad eoi(E) = true}{Freezable*(S,t), E \xrightarrow{\texttt{STOP}} Freezable(S,t), E} \tag{44}$$

$Freezable*$ is suspended while the freezing event is unknown:

$$\frac{S \notin E \quad eoi(E) = false}{Freezable*(S,t), E \xrightarrow{\texttt{SUSP}} Freezable*(S,t), E} \tag{45}$$

5 Execution Context

Execution context rewritings have the form $e \overset{b}{\Longrightarrow} e'$ meaning that reaction of the execution context e leads to the new execution context e'; b is a boolean which is true if the execution context e' is completely terminated.

5.1 Execution Context

An execution context executes one instant of its program in a new fresh environment.

$$\frac{Instant(t), Fresh \xrightarrow{\alpha} Instant(t'), E}{ExecContext(t) \overset{b}{\Longrightarrow} ExecContext(t')} \tag{46}$$

In this rule:

- $Fresh$ is the environment with an empty event set and such that $eoi(Fresh)$ and $move(Fresh)$ are both false, and with an empty frozen instructions table.
- b is true if α is `TERM`, and false otherwise.

5.2 Instant

Execution of an instruction during one instant means cyclic execution while it is suspended. Moreover, when execution suspends, end of instant is detected if no new move setting was performed.

The instant is terminated when the instruction is stopped or terminated:

$$\frac{t, E \xrightarrow{\alpha} t', E' \quad \alpha \neq \texttt{SUSP}}{Instant(t) \xrightarrow{\alpha} Instant(t'), E'} \tag{47}$$

Execution is immediately restarted if $SUSP$ is returned and end of instant is set if no new move setting was performed:

$$\frac{t, E \xrightarrow{\texttt{SUSP}} t', E' \quad move(E') = false \quad Instant(t'), E'[eoi = true] \xrightarrow{\alpha} u, E''}{Instant(t) \xrightarrow{\alpha} u, E''} \tag{48}$$

Execution is immediately restarted if `SUSP` is returned and *move* is reset if a move appeared:

$$\frac{t, E \xrightarrow{\texttt{SUSP}} t', E' \quad move(E') = true \quad Instant(t'), E'[move = false] \xrightarrow{\alpha} u, E''}{Instant(t) \xrightarrow{\alpha} u, E''} \tag{49}$$

6 Conclusion

Junior is a kernel model for reactive programming. It basically defines concurrent reactive instructions communicating with broadcast events. At the basis of **Junior** is the rejection of immediate reaction to absence, which is one of the major difference with synchronous formalisms[1].

The Par parallel operator in **Junior** is basically non-deterministic. This is the main difference with SugarCubes[5] which has a deterministic $Merge$ parallel operator.

References

[1] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Pub., 1993.

[2] L. Hazard, J-F. Susini, F. Boussinot, *The Junior reactive kernel*, Inria Research Report 3732, July 1999.

[3] L. Hazard, J-F. Susini, F. Boussinot, *Programming In Junior*, 2000.

[4] G. Plotkin, *A Structural Approach to Operational Semantics*, Report DAIMI FN-19, Aarhus University, 1981.

[5] F. Boussinot, J-F. Susini, *The SugarCubes Tool Box - A reactive Java framework*, Software Practice & Experience, 28(14), 1531-1550, 1998.

Contents