Junior Automata

Frédéric Boussinot

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Junior Automata

Frédéric Boussinot

N° 4031

Octobre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4031--FR+ENG

Junior Automata*

Frédéric Boussinot†

Thème 1 — Réseaux et systèmes
Projet Mimosa

Rapport de recherche n° 4031 — Octobre 2000 — 23 pages

Abstract: One describes the way to produce finite states machines from programs written in Junior, a formalism for reactive programming in Java. The paper proposes the notion of a partial automaton for dealing with large size numbers of states.

Key-words: Reactive Programming, Automaton, Concurrency, Java

* With support from France-Telecom R&D
† EMP/CMA-INRIA

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Automates en Junior

Résumé : On décrit comment produire des machines d'états finis à partir de programmes écrits en Junior, un formalisme permettant une programmation réactive en Java. On introduit la notion d'automate partiel, utile lorsque le nombre d'états est très grand.

Mots-clés : Programmation réactive, Automate, Parallélisme, Java

1 Introduction

Junior[4, 2] is a formalism for reactive programming in Java. It basically defines concurrent reactive instructions communicating with broadcast events. This paper describes production of finite states machines from Junior programs. It is implemented in Java using Junior itself.

Finite states machines, also called *automata*, are made of states and transitions: at each computing step, control starts from current state and executes the associated transition, leading to a new state which is the starting point for next step.

Basically, transitions are *trees* made of tests and of elementary computing actions. To execute a transition, one starts from the root of the tree and one executes elementary actions in sequence, following a path reaching to a leave which defines new state for next instant. Figure 1 shows the graphical representation of a transition :

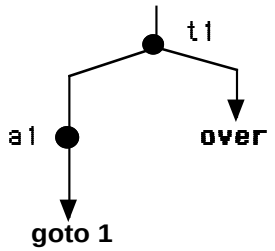

Figure 1: Transition

Execution starts by evaluating test `t1`; if `t1` returns true, the left branch is choosen, action `a1` is executed, and next state is set to state 1; if `t1` returns false, then right branch is choosen, and next state is set to final state, called `over`. The key point is that running a transition means to execute *sequential code*, by contrast with reactive instructions of Junior which basically are concurrent code.

In standard automata graphical representation, states are shown as circles, linked by transitions. Figure 2 is a representation of an automaton. Initial state has number 0, and `over` is final state. Transition starting from state 0 is the one of figure 1. Transition starting from state 1 consists in executing `a2`, then `a3`, then setting next state to state 0. Finally, transition from `over` is the empty transition.

Producing an automaton from a reactive program thus can be seen as *compiling concurrent code into sequential code*. The benefit is that one gets a more efficient execution (concurrency is managed once for all at compile time, not at run time) and possibility to interface with analysis and verification tools specially designed for automata. However, there are several drawbacks:

- Producing automata is not always possible; for example, recusivity can be an obstacle to production of an automaton.

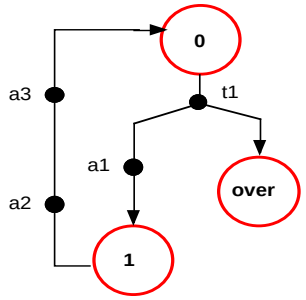

Figure 2: Automaton

- Automata can be very large, getting huge code size.
- Automata are less modular than initial programs: passing from parallel to sequential code forbids later uses in some contexts.

Automata are thus a compromise between efficiency on the one hand, and size and reusability on the other hand. In this paper, one make the proposal of an new kind of automata, called *partial automata*, for dealing with large sized automata.

The structure of the text is as follows: automata are introduced in section 2 and reactive compilers to produce them in section 3. Section 4 describes automata printers in Java and evaluators. Finally, partial automata and compilers are introduced in section 5.

2 Automata

An automaton is basically a table of states, where each state stores an associated transition. Automata are instances of class `Automaton` with basic methods:

```
Automaton()
Vector stateTable()
```

First method is the constructor for automata. Second method returns the states table implemented as a Java vector; elements of the vector are states defined in 2.1. Automata are produced by the reactive compiler described in section 3.

2.1 States

States are instances of class `State`; a state has a number and a transition starting from it. The number is a new fresh integer, given by the system at construction. Actually, each state of a Junior automaton is representing a reactive instruction provided when constructed. Methods for using states are:

```
State(Instruction instruction)
Instruction instruction()
void assignTransition(Transition t)
Transition transition()
int num()
```

First method is the constructor for states; parameter is the reactive instruction from which state is built; second method returns this reactive instruction. Third method sets the transition which is returned by fourth method. Finally, last method returns the state number.

2.2 Transitions

Transitions define sequential code associated to states. A transition has the form of a tree in which leaves are gotos or over, finishing execution for current instant; transition nodes are either binary nodes for testing events or boolean conditions, or unary nodes for actions or event generations. Execution of a transition starts from the root and follows a path reaching a leave which defines either a new state for next instant, or the `over` state which means that program is completely terminated.

Transitions implements the following interface:

```
public interface Transition {
   final int CONT = -2, OVER = -1;
   int execute(Environment env);
 }
```

Method `execute` executes the transition; for leaves, it returns OVER if over is reached, and the reached state otherwise; for others nodes, it returns CONT.

Now, one presents the various transitions. This description is useful when implementing new ways of printing or running transitions; it can be skipped when it is no the case.

2.2.1 Action

```
ActionTransition(Action action)
```

This transition executes `action` and returns CONT when executed.

2.2.2 Event Test

```
EventTestTransition(Identifier event, Transition transition)
```

Event `event` is the tested event, and `transition` is the transition that must be executed when `event` is present; in this case, execution returns the result of `transition` execution; otherwise (event is absent), it returns CONT.

2.2.3 Condition Test

```
ExpTestTransition(BooleanWrapper condition, Transition transition)
```

Wrapper `condition` is the tested condition; `transition` is the transition that must be executed when `condition` is true; in this case, execution returns the result of `transition` execution; otherwise (`condition` is false), it returns CONT.

2.2.4 Generation

```
GenerateTransition(Identifier event)
```

Event `event` is the generated event; execution returns CONT.

2.2.5 Goto

```
GotoTransition(int target)
```

This transition is a leave that terminates transition execution for current instant; new state for next instant is state with number `target`; execution returns `target`.

2.2.6 Over

```
Over()
```

This transition is a leave that indicates that automaton execution is over; execution returns OVER.

2.2.7 Sequence

```
SeqTransition(Transition left,Transition right)
```

This transition is the sequence of transitions `left`, then `right`. If execution of `left` returns CONT, it returns the value returned by `right` execution; otherwise, it returns the value returned by `left` execution.

2.2.8 Link

```
LinkedTransition(Object object)
```

This transition sets the linked object, accedeed by atoms through environment; it corresponds to `Link` instructions of Junior; execution returns CONT.

2.2.9 Empty Transition

```
EmptyTransition()
```

The empty transition; execution returns CONT.

2.3 Basic Printer

The basic printer prints an automaton as a list of transitions of the form:

```
statei: transition
```

where `i` is the state number and `transition` is the transition from the state. More precisely:

- Tests are of the form `if x then tr end`, where tr is a transition and x is an event or a boolean condition.
- Atomic actions are printed as they are.
- Generation of event e is printed as `generate e`.
- Goto are of the form `goto statei`; a final state is printed `over`.
- The sequence of two transition `t1` and `t2` is printed `t1;t2`.

For example, the states table of the automaton shown in figure 2 is:

```
state0: if t1 then a1; goto state1 end; over;
state1: a2; a3; goto state0;
```

Basic printers are instances of class `BasicPrinter` with methods:

```
BasicPrinter(Automaton automaton)
void print(String name)
```

First method is the basic printer constructor; automaton to be printed is passed as parameter. Second method is called to print automaton; parameter is the name given to the printed automaton. For example, to print an automaton `aut` with name `EXAMPLE1`, one writes:

```
new BasicPrinter(aut).print("EXAMPLE1");
```

which produces:

```
automaton EXAMPLE1:
state0: if e then over; end; goto state0;
end of automaton EXAMPLE1
```

This automaton actually corresponds to the Junior program `Jr.Await("e")`. It has only one state, named `state0`. Transition from `state0` tests event `e`; if `e` is present, then automaton is over; otherwise, automaton remains in `state0`.

3 Reactive Compilers

Reactive compilers symbolically execute programs, producing automata with equivalent behavior. Reactive compilers are instances of class `Rcompiler`, which has methods:

```
Rcompiler(Program program)
Automaton compile()
```

First method is the reactive compiler constructor; parameter is the program to compile. Second method returns automaton produced when compiling the program. For example, the automaton corresponding to program `Jr.Await("e")` can be obtained by:

```
Automaton aut = new Rcompiler(Jr.Await("e")).compile();
```

3.1 Compiling Process

Basically the compiling process is the following:

- Two sets of states are defined: the set of *created states* and the set of *analyzed states*. Initial state associated to the initial program is created; it is put in the set of created states.
- While there exists created states which are not analyzed, one of them is chosen and analyzed. When all created states are analyzed, if it happens, then the set of created states is returned as being the constructed automaton.

Analyzing a state means:

- Instruction associated to the state starts to be executed. When a non-local event is tested for presence during execution, one considers the two cases where event is, or is not, generated by the external world. Analysis forks assuming on one branch that event is generated, and on the other branch that it is not (noticing assumption on the event, in order to forbid future assumptions on it). Analysis of the two cases are reflected in the transition produced by a test on the event considered.
- `If` instructions are processed in the same way, except that the event test is replaced by a condition test.
- A `goto` is produced when execution of a branch is finished for current instant; the target is obtained by comparing the program which remains to be executed for next instants (the *residual* program) with already created states; if there exists a state with an equal program, then it is the target; if no state is found, then a new state is created with the residual program as associated instruction; this new state is put in the set of created states. Finally, `over` is used instead of `goto` if execution is completely terminated.

Now, one describes several important aspects of the compiling process.

3.2 Parallelism

Parallelism does not exist anymore in automata: it has been compiled in sequential code transitions. Consider for example the program which is waiting in parallel for two distinct events:

```
Jr.Par(Jr.Await("e"),Jr.Await("f"))
```

The corresponding automaton is:

```
automaton EXAMPLE2:
state0: if e then if f then over; end; goto state1; end;
        if f then goto state2; end; goto state0;
state1: if f then over; end; goto state1;
state2: if e then over; end; goto state2;
end of automaton EXAMPLE2
```

In `state1`, the automaton waits only for `f` because `e` was previously present. Similarly, in `state2`, the automaton waits only for `e`.

Note the "states explosion" phenomenon which appears when compiling parallelism; for example, with:

```
Jr.Par(Jr.Await("e1"),
...
Jr.Par(Jr.Await("en"),
       Jr.Await("en+1")...));
```

one gets an automaton with $2^{n+1} - 1$ states.

3.3 Events

Presence status of events are taken in account during the compiling process. For example, consider:

```
Jr.Seq(Jr.Generate("e"),Jr.Await("e"))
```

There is no test for event `e` in the produced automaton:

```
automaton EXAMPLE3:
state0: generate e; over;
end of automaton EXAMPLE3
```

Indeed, testing `e` is not needed, as it is generated in all cases.

In the special case of local events, presence status are completely solved at compile time. For example, consider:

```
Jr.Local("e",
 Jr.Local("f",
   Jr.Par(
     Jr.Seq(Jr.Await("f"),Jr.Generate("e")),
     Jr.Seq(Jr.Generate("f"),Jr.Await("e"))
)))
```

There is a communication from second branch of `Par` to the first branch, using event `f`; there is also a communication from first branch to the second, using event `e`; these two-way communications occuring in the same instant are what is called an "instantaneous dialog"[1]. The produced automaton is:

```
automaton EXAMPLE4:
state0: over;
end of automaton EXAMPLE4
```

As `e` and `f` are local events, they do not appear in the automaton.

3.4 Finite Loops

There are two ways for compiling finite loops, depending on the type of the expression defining the number of loop cycles.

3.4.1 Expanded form

Expanded form is obtained when expression defining the number of cycles is a constant n (integer or `ConstIntegerWrapper`); then, in the produced automaton, it is as if the loop body were expanded in n occurrences. For example, consider:

```
Jr.Repeat(3,Jr.Seq(Jr.Generate("absent"),Jr.Stop()));
```

Automaton produced is:

```
automaton EXAMPLE5:
state0: generate absent; goto state1;
state1: generate absent; goto state2;
state2: generate absent; goto state3;
state3: over;
end of automaton EXAMPLE5
```

Note that in expanded forms, the number of states highly depends on the value of the constant.

3.4.2 Compact form

Compact form is produced when expression defining the number of cycles is not a constant. In this case, the number of states does not depend on the value of the expression. For example, consider:

```
Jr.Repeat(new IntegerExpression("3"),
            Jr.Seq(Jr.Generate("e"),Jr.Stop()))
```

Then, automaton produced is:

```
automaton EXAMPLE5:
state0: _counter0=3; if _counter0--<0 then over; end; generate e; goto state1;
state1: if _counter0--<0 then over; end; generate e; goto state1;
end of automaton EXAMPLE5
```

3.4.3 Comparison of the two forms

One could think that expanded forms always leads to larger programs than compact forms[1]. This is false, as shown by the following example. Let us first call `huge` an instruction which produces a large size automaton. Then, consider the 3 following instructions:

```
Program fast =
  Jr.Seq(Jr.Repeat(5,Jr.Stop()),
  Jr.Seq(Jr.Atom(ForAut.Print("fast")),Jr.Generate("exit")));

Program slow =
  Jr.Seq(Jr.Repeat(50,Jr.Stop()),
  Jr.Seq(huge,
  Jr.Seq(Jr.Atom(ForAut.Print("slow")),Jr.Generate("exit"))));

Program prog =
  Jr.Local("exit",
      Jr.Loop(Jr.Seq(Jr.Until("exit",Jr.Par(fast,slow)),Jr.Stop())));
```

Instruction `fast` prints a message after 5 instants and then generates `exit`. Instruction `slow` waits for 50 instants, then runs previous `huge` instruction before printing a message and generating the same event `exit`. Instruction `prog` runs `fast` and `slow` in parallel and preempts them by `exit` (which is local). As 5 is less than 50, message `slow` is never printed because preemption always occurs before the printing action of `slow` gets a chance to be executed. Compiling `prog` gives:

```
automaton TOTAL:
state0: goto state1;
state1: goto state2;
state2: goto state3;
state3: goto state4;
state4: goto state5;
state5: System.out.print("fast"); goto state6;
state6: goto state1;
end of automaton TOTAL
```

The point is that size of the automaton is independant of `huge`, which is actually never executed. Now, replacing the two constants 5 and 50 by expressions returning same values, one gets an automaton which always has more states than the one of `huge`. Thus, the automaton produced in this case is larger than the previous one.

Producing `TOTAL` from `prog` can be seen as a (very elementary) proof that message `slow` is never printed by `prog`; note that values of the two constants must necessarilly be considered to manage such a proof.

[1] at least, when number of cycles is greater than 2.

3.5 Minimality

It is natural to consider as equivalent two states from which the automaton will for ever perform the same actions, provided inputs are the same. More precisely, two states are equivalent[2] if, with same input, the automaton performs the same elementary actions, reaching two new states which are also equivalent.

Equivalent states are clearly redondant in an automaton: it is possible to suppress all but one of them (changing gotos on redondant states, by gotos on the remaining state) without changing the automaton behavior. This transformation is called *minimisation*, and one says that an automaton is *minimal* if it does not contains two equivalent states.

Unfortunately, automata produced from Junior programs are not minimal, as shown by the very simple example:

```
Jr.Par(Jr.Await("e"),Jr.Nothing())
```

The produced automaton is:

```
automaton NotMinimal:
state0: if e then over; end; goto state1;
state1: if e then over; end; goto state1;
end of automaton NotMinimal
```

Actually, `state0` corresponds to the initial program, in which none of the two parallel branches are executed; `state1` corresponds to situation where `e` is not present but second branch has been executed. These two states are clearly equivalent as their associated transitions are identical. Once minimized, one gets the automaton:

```
automaton Minimal:
state0: if e then over; end; goto state0;
end of automaton Minimal
```

To get a way to minimize automata, reaching thus minimal automata, would certainly be a good point for reducing automata code size. This could be obtained by interfacing Junior automata with validation tools in which minimisation is implemented; this point is left for future work.

3.6 Modularity

Compiling parallel code into sequential code is non modular because choices made during compilation may be obstacles to reusability. To show this phenomenon, consider the following instruction P:

```
Jr.Par(
     Jr.Seq(Jr.Await("l1"),Jr.Generate("l2")),
     Jr.Seq(Jr.Await("r1"),Jr.Generate("r2")));
```

[2] One also says that they are *bisimilar*.

P terminates when put in parallel with L:

```
Jr.Seq(Jr.Generate("l1"),Jr.Seq(Jr.Await("l2"),Jr.Generate("r1")))
```

or with R:

```
Jr.Seq(Jr.Generate("r1"),Jr.Seq(Jr.Await("r2"),Jr.Generate("l1")))
```

Situation is symetric because, in both cases, execution can fire the parallel branch of P corresponding to the first generated event, leading to generation of the other event.

Symetry is lost if sequential code is used instead of parallel code. Indeed, in order to compile P into sequential code, one must choose one event, out of `l1` and `r1`, that is first tested for presence. Let us suppose that `l1` is chosen. Then, as sequential code is considered, status of `l1` must be determined at the very first step. Putting the sequential code in parallel with L would be ok; but execution will block with R, as there would be no way to progress. The point is that the same problem would appear if `r1` is chosen, instead of `l1`, to be first tested for presence. Actually, for each possible choice, there exists a context in which execution blocks.

This discussion shows that modularity is, in a way, the price to pay for compiling parallel code into sequential code. This has important consequences when one wants to produce executable code from automata; it will be discussed in section 4.1.

4 Printers

Basic printers of class `BasicPrinter` have been defined in section 2.3. Two other printers for automata are presented in this section.

4.1 Java Printer

Java printers are producing reactive machines from automata. Produced machines implements interface `Machine` but dynamic adding of new programs is not possible (a warning message is printed when method `add` is called). Actually, one uses the REPLACE implementation of Junior and produced machines extends class `BasicContext` defined in the package `junior.core` of this implementation (see [2] for details).

Java printers are instances of class `JavaPrinter` with methods:

```
JavaPrinter(Automaton automaton)
void print(String name)
```

First method is the class constructor, and parameter is the automaton to be printed. Second method prints automaton, giving it the name in parameter.

For example, let us consider the program:

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

The machine produced by:

```
   Automaton stepProducer = new Rcompiler(p).compile();
   new JavaPrinter(stepProducer).print("stepProducer")
```

is:

```
class stepProducer extends junior.core.BasicContext
{
public stepProducer(){ super(Jr.Nothing()); }
protected int state = 0;
public void add(Program program){ System.out.println("Warning: no add allowed");}
boolean test(String s){ return env.isGenerated(Jr.StringIdentifier(s)); }
void gen(String s){ env.generate(Jr.StringIdentifier(s)); }
public boolean react(){ // activation method
boolean res = false;
switch (state){
case 0:  gen("step");
   if (test("suspend")){ state = 1; res = false; break; }
   state = 2; res = false; break;
case 1:  if (test("resume")){
   gen("step");
   if (test("suspend")){ state = 1; res = false; break; }
   state = 2; res = false; break;
}  state = 3; res = false; break;
case 2:  gen("step");
   if (test("suspend")){ state = 1; res = false; break; }
   state = 2; res = false; break;
case 3:  if (test("resume")){
   gen("step");
   if (test("suspend")){ state = 1; res = false; break; }
   state = 2; res = false; break;
}  state = 3; res = false; break;
default: res = true;
}
env.newInstant();
return res;
}
} //end of class stepProducer
```

In this code:

- `add` method is redefined in order to forbid dynamic addings; moreover, the program passed to the super class is never used (it is, thus, simply set to `Nothing`).

- `react` method is implemented as a switch statement on variable `state` which contains the actual state number.
- Event generations and tests are performed on the machine environment (using the two auxiliary methods `test` and `gen`).

Note that automaton is not minimal: states 0 and 2 are equivalent, as are states 1 and 3.

As explained in section 3.6 in discussion about modularity, it would not be possible to produce a program, or a reactive instruction, in place of a machine; indeed, a produced program could be put in parallel with others programs, and modularity problems could then appear. This is why automata produced from Junior programs are implemented as reactive machines and not as reactive instructions: a reactive machine can only be used as top level executable code, and reusability is restricted to production of new instances of it.

4.2 Evaluators

Evaluators are extracting three characteristics from automata: the number of states, the global number of elementary actions (generation, tests, Java actions, and gotos) present in the automaton, and the size of the longest transition (number of elementary actions of the longest path). Evaluators are instances of class `Evaluator`. For example, the call:

```
new Evaluator(stepProducer).print("stepProducer");
```

produces output:

```
//evaluation of stepProducer - states: 4, actions: 20, longest path: 4
```

5 Partial Automata

A partial compiler is a machine that performs the compiling process during program execution. In a partial compiler, automaton is built on demand, when needed by execution; it is called a *partial automaton* because it can be run despite the fact that some states are left unanalyzed. The point is that states that are not reachable by execution (often called *unreachable states*) are not analyzed and will not appear in the constructed automaton. This is thus a way to reduce automata size.

Partial compilers have two running modes:

- In the *unlimited mode* (which is the default mode), there is no limit to the size of the automaton built. Each reaction executes the transition associated to the current state if it exists (that is, if it has been previously analyzed); otherwise, reaction starts by analyzing the state, that is computes the associated transition before executing it. In this mode, each reaction executes a transition.

- In the *limited mode*, the number of analyzed states is limited. When limit is reached, then program is run if the machine has to execute an unanalyzed state (a state without any transition associated to it). Thus, in limited mode, when automaton cannot be run, program is executed instead, as with a standard machine.

Partial compilers are instances of class `RpartialCompiler` and have the following methods:

```
RpartialCompiler(Program program)
int howManyAnalyzedStates()
void limit(int max)
boolean limited()
void noMoreStates()
```

- First method is the constructor; parameter is the program to be run; it cannot be changed: calls to the `add` method have no effect.
- Method `howManyAnalyzedStates` returns the number of states currently analyzed (that is, for which an associated transition exists).
- Method `limit` sets the limit number of analyzed states to the value of its parameter.
- Method `limited` returns true if the compiler is in limited mode, false otherwise.
- Method `noMoreStates` puts the compiler in limited mode and sets the limit of analyzed states to the actual number of them; thus, after the call, no new states can be analyzed (or even created).

5.1 Unlimited Mode

One considers a small program which cyclically prints a message; this program is put in parallel with an instruction that waits for an event and then falls in an instruction producing a huge automaton:

```
Program go =
   Jr.Loop(Jr.Seq(Jr.Repeat(3,Jr.Stop()),Jr.Atom(ForAut.Print("Go"))));
Program prog = Jr.Par(Jr.Seq(Jr.Await("e"),huge),go);
```

Then, one defines a partial compiler `partial`, and let it react several times:

```
for(int i = 0; i<50; i++){ partial.react(); }
```

Finally, one gets the automaton built and prints it:

```
Automaton aut = partial.automaton();
new BasicPrinter(aut).print("UNLIMITED");
```

The point is that `huge` is never reached because event `e` is never generated; thus, the produced automaton has a small size:

```
automaton UNLIMITED:
state0: if e then goto state1; end; goto state2;
state1:
state2: if e then goto state3; end; goto state4;
state3:
state4: if e then goto state5; end; goto state6;
state5:
state6: if e then System.out.print("Go"); goto state1; end;
        System.out.print("Go"); goto state2;
end of automaton UNLIMITED
```

Only even states are reached (odd states correspond to `huge`), and execution always stays in this set of states as long as `e` is absent. Using a partial compiler, it thus becomes possible, in specific situations (here, absence of `e`), to run a program as a finite states machine, but without having to produce the global automaton.

5.2 Limited Mode

Here is a small example of an executable program run by a partial compiler in limited mode. The considered program is a finite loop producing a 10 states automaton, enclosed in an infinite loop. The partial compiler is allowed to analyze at most 5 states. After several instants, automaton stored in the partial compiler is printed:

```
public class Partial
{
  public static void main(String[] argv){
     RpartialCompiler partial =
        new RpartialCompiler(Jr.Loop(Jr.Repeat(10,Jr.Stop())));
     partial.limit(5);

     for(int i = 0; i<100; i++){
         System.out.print("***** instant "+i+": ");
         partial.react();
         System.out.println("");
     }

     Automaton aut = partial.compiler().automaton;
     new BasicPrinter(aut).print("LIMITED");
  }
}
```

The partial automaton which is finally printed is:

```
automaton LIMITED:
state0: goto state1;
state1: goto state2;
state2: goto state3;
state3: goto state4;
state4: goto state5;
state5:
end of automaton LIMITED
```

Thus, 6 states have been created, out of which 5 are analyzed. Output is (only the 16 first instants are printed):

```
***** instant 0: runing state 0...
***** instant 1: runing state 1...
***** instant 2: runing state 2...
***** instant 3: runing state 3...
***** instant 4: runing state 4...
***** instant 5: running the program...
***** instant 6: running the program...
***** instant 7: running the program...
***** instant 8: running the program...
***** instant 9: running the program...
***** instant 10: running the program...
***** instant 11: runing state 1...
***** instant 12: runing state 2...
***** instant 13: runing state 3...
***** instant 14: runing state 4...
***** instant 15: running the program...
```

This output shows how execution cyclically switches from states to program, and conversely from program to states.

5.3 Possible Variants

To end this section, one can think of several possible variants of partial compilers and automata:

- The state table of a limited compiler can be considered as a *cache table*; a state which is not run during a certain delay would be removed from the table, leaving place for a new one.
- Only states with limited size transitions are stored; thus, automaton size would be controlled not only in the number of analyzed states, but also in the number of elementary actions it contains.
- After a certain time, one extracts the automaton of an unlimited compiler (for example, in order to analyze it); after extraction, an error is produced if execution run out of the analyzed states.

6 Related Work

6.1 JIT Compilers

The approach of partial automata is actually close to "just-in-time" (JIT) optimizing technics notably used by Java compilers[6]; in both cases, execution is a compromise between interpretation and compilation, and this compomise can vary along time. Main difference is

that partial automata put the focus on concurrency, and not on sequential code optimization as JIT technics do; note however that the two approaches are compatible and could be used jointly.

6.2 Synchronous Languages

Automata have been used as targets when compiling synchronous languages[3], specially in the v3 implementation of Esterel[1] (let's called it Esterel v3, for short). In Esterel v3, automata states are program states, and transitions are statements coding for program reactions. A program state is actually completely determined by the sets of `halt` instructions on which control is stuck. Basically, the Esterel v3 compiling process symbolically executes the program, building two sets:

- the set of states (called *halt-sets*) corresponding to program states;
- the set of transitions associated to states. A transition is a tree made of boolean tests (signal tests, and `if` statements) and of actions (variable assignments, procedure calls, etc.).

Actually, the compiling process for producing automata from Junior programs is more or less the one of Esterel v3. However, there are several differences:

- Instantaneous loops are not detected in Junior[4]; running an instantaneous loop can thus lead to a non convergent situation.
- At the basis of Junior is the rejection of immediate reaction to absence, which is one of the major difference with synchronous formalisms. A consequence is that causality cycles, which are a major problem in Esterel, do not exist in Junior. Thus, there is no need for causality cycles detection during automata construction.
- Recusively defined programs can also lead to non convergent situations; this is impossible in synchronous languages, where recursivity is forbidden.

In current version of Esterel (v5), automata have been rejected to avoid states combinatorial explosion problems. Automata are replaced by equations sets; however, software execution of equations is slower than execution of automata; this is the usual tradeoff between efficiency and code size. Moreover, equations are closer to harware circuits, for which Esterel is targeted, than automata are.

6.3 Mealy Machines

Mealy machines are finite states machines in which transitions are linking states and are holding conditions and output produced when condition holds; several transitions can be associated to same state, and runing the automaton means to choose a transition from current state that have a condition which is true. For example, figure 3 shows the Mealy machine associated to:

```
Jr.Par(Jr.Await("e"),Jr.Await("f"))
```

On this drawing, conditions are put over transitions which are simple arrows (one does not consider outputs). Presence of an event x is represented by x, and absence by $\bar{x}$. Conditions are simply concatenations of event presences or absences; for example $e\bar{f}$ means e present and f absent.

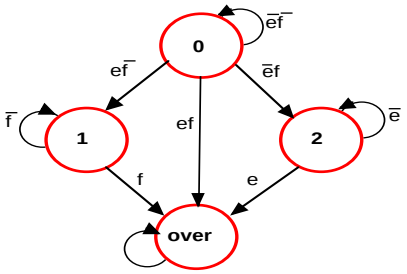

Figure 3: Mealy Machine

Mealy machines are well suited for program verification systems because transitions are very simple; moreover, each state directly exhibits all possible program behaviors from it. However, Mealy machines are less adapted to be directly run, because before executing a transition there is always the overhead of determining it.

Mealy machines are important in the context Junior automata because Mealy machines are standard input for automata-based verification systems; first step for verification of Junior automata would be, then, to translate them in Mealy machines.

6.4 Mode Automata

Mode automata[5] have been recently introduced in the context of dataflow synchronous languages[3] to deal with reactive systems which have running modes. Actually, in the mode automata model, states are labelled by dataflow programs, and are representing system modes. Thus, mode automata are a mix between dataflow and imperative programming. They can also be seen as mixing concurrent code (dataflow programs) and sequential code (transitions).

Mode automata are a way to avoid the combinatorial state explosion when compiling dataflow synchronous programs because the number of states depends on the semantics of the program modes; thus, one can hope that programmers may limit the number of produced states, using appropriate syntax constructs.

7 Conclusion

A way to produce automata from Junior programs has been described. Basically, the compiling process transforms parallelism into sequential code, when possible.

There are two intrinsic problems with automata: modularity, that is reusability, and code size. Proposition to deal with these problems is twofold:

- Automata are implemented by reactive machines to avoid modularity problem that would occur if they were implemented as reactive instructions. Of course, it is a very limited solution as reusability is thus restricted to production of new program instances.
- Partial compilers give possibility to limit the number of states produced; a partial compiler can run the automaton states or the program, according to the number of analyzed states. Thus, partial compilers give a way to deal with programs whose automata have large numbers of states.

A specific problem of the compiling process is that non minimal automata are produced. To minimize automata is left for future work.

Automata seem useful in several contexts:

- Program analyzis and verification of program properties. Model checking is the natural verification technics with automata; it would be helpful to be able to print automata in some verification system input format. A possible use would be to extract some required properties from automata; non-regression tests could then verify that properties are still valid for new program versions.
- Production of automata for security concerns: the simple fact that an automaton has been produced from a program shows, for example, that it will never fall in a recursive loop where new parallel components are created for ever, forbidding the rest of the system to work properly. This could be useful for migrating agents; indeed, agents are generally small programs, from which automata can thus be produced; one can thus imagine some kind of "automaton carying agent" which can exhibit the automaton produced out of it, in order to enter into a remote site.
- Evaluation, as shown in section 4.2, of the longest transition of an automaton. This could be used for *Quality of Services* (QoS) purposes: for example, one could think to restrict the language of elementary actions, for being able to evaluate execution time for them; in this context, it would be possible to compute bounds for execution time, which is an important matter for QoS, and more generally for real-time programming.

The Junior implementation and processors presented in the paper are freely available on the Web[7].

References

[1] G. Berry, G. Gonthier, *The Esterel Synchronous Language: Design, Semantics, Implementation*, Science of Computer Programming, 19(2), 87-152, 1992.

[2] F. Boussinot, L. Hazard, J-F. Susini, *Programming with Junior*, available at `http://www.inria.fr/mimosa/rp/Junior`, July 2000.

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

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

[5] F. Maraninchi, Y. Remond, *Mode-Automata: About Modes and States for Reactive Systems*, Proc. European Symposium on Programming, Lisbon, Portugal, 1998.

[6] see information on Sun site at `http://java.sun.com`

[7] at `http://www.inria.fr/mimosa/rp/junior`

Contents

1 Introduction **3**

2 Automata **4**
2.1 States . . . 4
2.2 Transitions . . . 5
2.2.1 Action . . . 5
2.2.2 Event Test . . . 5
2.2.3 Condition Test . . . 6
2.2.4 Generation . . . 6
2.2.5 Goto . . . 6
2.2.6 Over . . . 6
2.2.7 Sequence . . . 6
2.2.8 Link . . . 6
2.2.9 Empty Transition . . . 6
2.3 Basic Printer . . . 7

3 Reactive Compilers **8**
3.1 Compiling Process . . . 8
3.2 Parallelism . . . 9
3.3 Events . . . 9
3.4 Finite Loops . . . 10
3.4.1 Expanded form . . . 10
3.4.2 Compact form . . . 10
3.4.3 Comparison of the two forms . . . 11
3.5 Minimality . . . 12
3.6 Modularity . . . 12

4 Printers **13**
4.1 Java Printer . . . 13
4.2 Evaluators . . . 15

5 Partial Automata **15**
5.1 Unlimited Mode . . . 16
5.2 Limited Mode . . . 17
5.3 Possible Variants . . . 18

6 Related Work **18**
6.1 JIT Compilers . . . 18
6.2 Synchronous Languages . . . 19
6.3 Mealy Machines . . . 19
6.4 Mode Automata . . . 20

7 Conclusion **21**

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399