Electronic Journal of SADIO

http://www.dc.uba.ar/sadio/ejs/

vol. 1, no. 1, pp. 52-75 (1998)

Some Issues in Using Formal Methods for the Development of Reactive Systems

Pablo Argón[1] *Olivier Roux*[1,2]

[1] IRCyN (UMR 6597), 1 rue de la noë, 44321 NANTES (FRANCE).
{argon|roux}@lan.ec-nantes.fr
[2] Institut Universitaire de France.

Abstract

For the development of safety-critical reactive systems, proving correctness is unavoidable. Here we describe some research issues on using and combining formal methods. Using the Electre reactive language we illustrate a technique to the design of a sound compiler with the Coq theorem prover. Based on the same source language semantic model, we present the outlines of a method to verify correctness claims with the SPIN model checker.

Keywords: Compiler design, Coq theorem prover, Electre reactive language, program proof, program extraction, model checking, Spin model checker.

1 Introduction

Nowadays, for the development of safety-critical reactive systems[3], establishing *correctness* is a crucial task, if it is not even required by security standards. Nevertheless, correctness is a relative concept, and we must distinguish at least two different aspects: the *specification correctness*, and the *construction correctness* of the system from its specification. The first aspect cannot be entirely formalized, it will always remain a "gap" between the problem itself, and the problem specification (this point is out of the scope of this paper). On the other hand, strong results and techniques are now available to ensure the correctness of the construction (or the derivation) of a system from its formalized specification. We can mention: research on semantics of programming languages, automata theory, development of *ad hoc* logics, model checking, and recently, implementation of powerful proof assistants and automatic theorem provers through constructive higher-order logic. It seems clear that, current research challenge is to combine these techniques in order to tackle the complexity of real concurrent systems.

In this paper we present some results and issues on the formalization of the Electre [CR95, Arg95] language semantic model using the Coq [CCF$^+$95] logical meta-language, and an application to the verification of temporal claims using SPIN [Hol97]. We claim that development of such formalized framework is a suitable basis for the formalization of most of the development stages of reactive systems providing rigorous foundations: from specification language design through specified system properties verification.

Organization of the paper. We start with a brief recall of the Electre reactive language in Sect.2. In Sect.3 the Coq theorem prover is introduced and we sketch the Electre compiler design and certification with Coq. In Sect.4 we give the outlines of the implementation mechanism of Electre programs into PROMELA/SPIN [Hol91, Hol97]. Sect.5 presents some research issues on using this model to combine the Coq theorem prover engine and the Spin [Hol91] model checker, to verify properties on unbounded Electre programs. Then, in Sect. 6 we show a Spin implementation of the readers-writer problem modeled in Electre, in which we verify safety and liveness properties. Conclusions and further work are presented Sect.7.

[3]Belong to this category; embedded systems in avionics and trains, some industrials controllers, etc.

Remark: The aim of this paper is to give a survey on the utilisation of formal methods to the desing and validation of a reactive system. The reader interested on a particular stage of this development may refer to the papers cited in each section.

2 Overview of the Electre language

Electre is a language for describing the scheduling of *tasks* with regard to *events* in reactive applications programming. Roughly, a reaction of an Electre program to the environment (the history of the events that have occurred up until now) is to "rewrite" itself in the part of the program which is left to be executed. This arises at each event occurrence.

Then, this remainder of a program is the feature of one of the states in the derived *labeled transition system* which is the execution model of the initial program. In this way, Electre is a *reactive* language since the program rewriting implies changes on the tasks status: some are to be run, others suspended or stopped, according to the event occurrences. These changes are the output of a reaction which is identified with a transition in the automaton.

The formal semantics of the language is such that it has been possible to prove that each program can be compiled into a unique transition system [CR95].

Basic components. *Tasks* stand for actions whose duration is finite but not null. They refer to sequences of executable code.

Indeed, we are not concerned with the action itself but rather with the three main featuring transitions around a *lengthing action* : start (or resumption), preemption, completion.

Moreover, it becomes easy to add properties to the tasks so as to give some particular behavior specifications, e.g.: "the task would never be preempted" or else "restart the task at the beginning rather than resume it at the point where it was preempted".

Event is the second type of structured entity Electre handles. It is linked with software or hardware signals in order to deal with their multiple occurrences. It acts as a clock ticking at unpredictable instants.

As it was already mentioned, the event is the distinctive trigger of one transition. Nevertheless, a specific issue about events is that, since the instants of occurrences are unpredictable, it may happen that some events occur when they must not be treated immediately, but they have to be memorized in order to be taken into account later. For that reason, the model we use is accurately a FIFO labeled transition system.

As a matter of fact, the occurrence of an event leads either to its memorization, or to the immediate launching of an action.

Obviously, properties may also be assigned to events, e.g.: "no more than one occurrence of the event has to be memorized" or else "no occurrence of the event has to be memorized" (which means the event is fleeting).

Putting together tasks and events. Events "schedule" the *preemption* and *activation* of tasks. These are the main operators of the language and they make it possible to compose intricated behavior structures. For example, consider a very simple program `p` as: `A await e:B` (where `A` and `B` stand for tasks and `e` for an event). Then, as soon as `e` occurs, the program rewrites in `B`, and the reaction is <`preempt A`, `launch B`>. This is expressed by the following rules:

$$e \vdash \text{`}\texttt{A await e:B}\text{'} \overset{\text{<preempt A, launch B>}}{\rhd} \text{`}\texttt{B}\text{'}.$$

And the next reaction is:

$$end_B \vdash \text{`}\texttt{B}\text{'} \overset{\text{<end B>}}{\rhd} \texttt{nil}$$

where `nil` means that the program is finished.

The semantics of the language is made up of these rules which are applied structurally on a program in order to compile it. Actually, the rules are conditional since several things may happen in a given state. For example, according to the individual behaviors of their components (programs `p1` and `p2`), there are four possible rules to apply when dealing with the parallel program `p= p1 || p2`. Two of these rules are given below (the other two rules being the symetric ones with respect to `p2`):

$$\frac{e \vdash \texttt{p1} \overset{\langle x_1 \rangle}{\rhd} \texttt{nil} \wedge e \vdash \texttt{p2} \overset{\langle x_2 \rangle}{\rhd} \texttt{p'2}}{e \vdash \texttt{p1 || p2} \overset{\langle x_1 \cup x_2 \rangle}{\rhd} \texttt{p'2}} \tag{1}$$

$$\frac{e \vdash \texttt{p1} \overset{\langle x_1 \rangle}{\rhd} \texttt{p'1} \wedge e \vdash \texttt{p2} \overset{\langle x_2 \rangle}{\rhd} \texttt{p'2}}{e \vdash \texttt{p1 || p2} \overset{\langle x_1 \cup x_2 \rangle}{\rhd} \texttt{p'1 || p'2}} \tag{2}$$

All the operators of the Electre language are not listed in this paper but the reader may refer to [CR95] to know them.

3 Certified Compiler Construction with Coq

The Coq theorem prover. The Coq system is a proof assistant for higher-order logic, based on an extension of the Calculus of Constructions with inductive types. We distinguish three main aspects of the system:

- *the logical language*, in which we write axioms and specifications in order to build *theories*,
- *the proof assistant*, which allows to develop proofs of specifications interactively applying *tactics*, and
- *the program extractor*, which synthesizes CAML-programs from the constructive proof of the specification.

This environment can be used both as a logical framework for developing machine checked mathematical proofs, and as a programming environment for developing certified programs. For our work, we stress the latter aspect. The next paragraph details program certification process, by introducing the Coq devoted tactic.

In theory, we are able to synthesize a functional program from the constructive part of a specification proof, in accordance with the Curry-Howard isomorphism.

The work of C.Parent [Par95] defines the notion of *weaker extraction function* and proves that this function is invertible under certain constraints. The point is to give a specification and a functional term "realizing" this specification. Then, in accordance with the inversibility of the extraction mechanism, it is possible to take advantage of the information encoded into the functional assertion, to achieve the specification proof. Enriched with logical annotations the asserted function is viewed as a pre-built proof. Annotation introduces in a functional term a logical assertion that will be used for the proof. By this mechanism, annotations add, to the function, the logical part which is needed for the proof and which cannot be automatically retrieved. The specialized Coq tactic, named `Program_all`, is based on these theoretical results.

Language Semantic Model. We start with a *deep embedding* [Mel96] of the Electre language into the Coq meta-language (i.e. a Coq model of the Electre syntax and semantics). The semantics (given as SOS rules) define the allowed transition steps of any program: given any program and input event, application of the semantic rules determines the output events generated by the program reaction and a new program to handle subsequent input events.

Then, the idea is to encode SOS rules into a Coq predicate. The rules are encoded through an inductively defined set EleSem, each rule being transformed into a set constructor. A transition $\mathtt{e} \vdash \mathtt{p} \rhd \mathtt{p'}, \langle em \rangle$ is valid, if and only if the Coq term (EleSem e p p' $\langle em \rangle$) can be proven correctly constructed. Note that in Coq context, as in any intuitionistic logical framework, "true" means "effectively constructed".

Correctness criterion. The predicate EleSem, becomes our correctness criterion in the following sense: if $\mathcal{T}$ is the constructed automaton (by compilation) of program p, all transitions of $\mathcal{T}$ must make true the predicate EleSem, and each tuple (e, p, p', $\langle em \rangle$) making true the predicate is in $\mathcal{T}$. Both conditions are needed to ensure respectively *soundness* and *completeness* of the constructed automaton. The reader interested in the details of this proof, may refer to the paper [AMR96].

Certification. The above criterion is translated into a Coq theorem (named Compiler) which specifies, for any Electre program the equivalent automaton. In order to prove this theorem, we associate to this specification, an *annotated functional term.* This functional term computes the automaton which complies with the stated specification.

While proving the Compiler theorem, the Program_all tactic generated several logical subgoals, which were easily solved by the user.

Compiler Extraction. In addition, Coq offers the possibility to realize all proven functions in a ML-like functional programming language, CAML [MM92]. Hence, all the structures we use are synthesized in their corresponding CAML type definitions, and the functions are expanded; this "kernel" can be enhanced into an effective Electre compiler.

4 Implementing Electre with Spin/Promela

The Spin model checker and its specification language: Promela. SPIN is a generic verification system that supports the design and verification of asynchronous process systems [Hol97], developed by Gerard Holzmann at Bell Laboratories.

SPIN tool provides:

- a specification language named PROMELA (PROcess MEta LAnguage),
- a concise notation for expressing correctness requirements (assertions, progress-labels, etc.); including temporal claims written in standard LTL,
- an optimized stand-alone model checker generator for establishing correctness claims.

It also provides a graphical front-end named XSPIN which includes an interactive simulator. Simulation can be randomic, steered by the user, or steered by a counterexample trace generated while verifying a correctness claim.

A PROMELA specification consists of one or more process templates defined using constructs based on the guarded command language CSP with a C-like syntax. Process templates can be dynamically instantiated and can communicate through message channels. Communications can be defined to be synchronous (by rendez-vous) or asynchronous (buffered message passing). Each process is translated into a finite automaton, and the resulting global system is obtained by computing an asynchronous interleaving product of automata.

To perform verification, SPIN computes on-the-fly the Büchi automaton resulting from the synchronous product of the claim[4] and the global system: then if the language accepted by this automaton is empty, this means that the original claim is not satisfied for the given system.

All these features makes SPIN adapted to the model and the formal verification of temporal properties of software systems. In the next section we present the outlines of the PROMELA/SPIN implementation of Electre programs.

Implementing Electre. Electre programs are implemented in PROMELA by associating a generic process `module` to each module, a process `Queue` to the FIFO-list, a non-deterministic process `Environment` generating the events, and synthesizing a process `Controller` from the source program, as explained below. Fig.1 shows the communication diagram between processes. The two side arrow means a double-way communication channel. Only the channels between the controller and the modules are synchronous, because the controller must know the exact internal state of each module.

[4] Any LTL formula can be translated into a Büchi automaton.

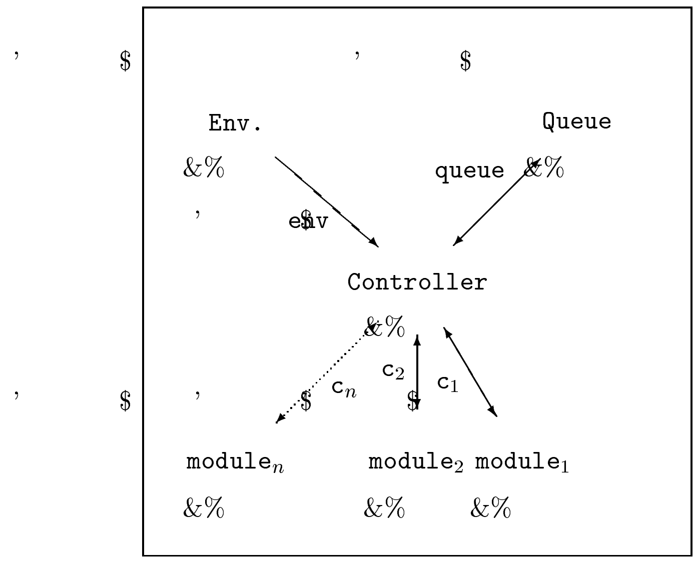


Figure 1: Communication diagram

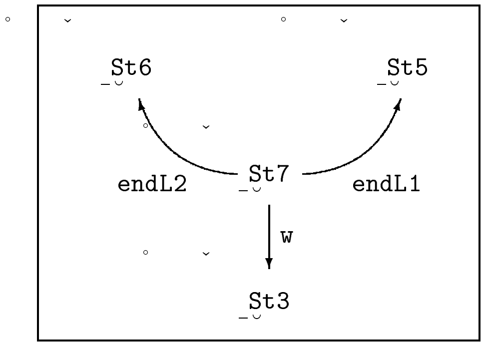
The generic process module models the three possible states in which each module can be: *Terminated, Activated* (or running), or *Interrupted.*

The synthesis of process `Controller` is based on the compilation of Electre programs into finite automata (see Sect.2). Every state of the finite automata resulting from the compilation of an Electre program, appears as a "state label" in the `Controller` process. The structure of a controller state is a list of (guarded) reception commands.

The Fig.2 presents the transitions from state 7 of the automata corresponding to the example program of Sect.6, and an excerpt of the resulting code of the `Controller`. For an explanation of the notation, and the whole PROMELA program listing, see appendix A.

Process `Queue` obeys to the same synthesis principles; the standard semantics of a FIFO-list is implemented as a 3 state automata. Initially, the queue is waiting for an event ("pushing" action), or waiting for a requesting ("poping" action) from the controller. The queue itself is a vector, its actual size being parametrized by the user.

As a conclusion, we notice that our implementation is very close to the execution model.




```
St7:
do
::queue!CTR(REQ);
  queue?CTR(x);
  if
   ::(x==EMY)-> break
   ::(x==wff)-> queue!CTR(YES);
                c_L1!Int;
                c_L2!Int;
                c_W!Act;
                goto St3
   ::else    -> skip
  fi
od;
do
::c_L1?end -> goto St5
::c_L2?end -> goto St6
::env?w    -> queue!CTR(SND);
              queue!NORM(l2ff);
              queue!NORM(l1ff);
              queue!CTR(ESD);
              c_L1!Int;
              c_L2!Int;
              c_W!Act;
              goto St3
od
```

Figure 2: Example of `Controller` synthesis

5 Combining Theorem Proving and Model Checking

On the one hand, model checking techniques are limited to finite systems[5] (in practice limited to "not too big" finite systems). Its main advantage is to be fully automatic, with minimal user interaction. A model checker answers "yes"or "no"; with sometimes a counterexample. On the other hand, theorem provers can deal with infinite structures and prove properties about them using structural induction, but requires user interaction at each proof step. Even if automatic high-level tactics exist, the user must intimately know underlying logical mechanisms to actually use them.

In this section we show how the Electre semantic model used in Sect.3 can be used to combine the Coq theorem prover with a model checker (Spin for example), in order to prove *safety* or *liveness* properties using a linear time temporal logic (LTL).

We recall that the Electre execution model is FIFO-Automata (cf. Sect1). Consequently, the main Electre verification problem comes from the possible unbounded growth of FIFO-list (where events are stored while waiting to be taken into account by the scheduler).

Dealing with a potentially infinite Fifo-list. The idea is to *abstract* infinite objects, i.e. to map infinite structures into finite ones [CGL94]. Even if the abstraction may depend on the property that we want to prove, some classes of abstractions concerning the FIFO-list may be automatically chosen.

An example of such a *morphism* mapping is given in Fig.3; an unbounded queue of events e is mapped into a three elements set. Using such an abstraction, we successfully prove liveness claims in several examples.

Soundness of verification is ensured by the fact that whenever a formula holds in the abstract system, it also holds in the concrete system. More formally, if we note $\mathcal{C}$ the concrete system, $\mathcal{A}$ the abstract one, and φ property to be proved; abstraction mapping must ensure the following relation:

$$\mathcal{A} \models \varphi \Rightarrow \mathcal{C} \models \varphi$$

Obviously, the converse is false: although the property holds in the concrete system, there might not be an abstraction such that the property holds in the abstract system (or the user may not find it).

[5] Using symbolic representation for the space of states, this technique can be extended to a certain class of infinite systems [HNSY92].

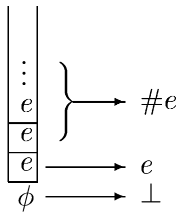


Figure 3: Example of morphism mapping

Stating mapping soundness. Since LTL formulas are interpreted over all possible automata behaviors, it suffices to show that every concrete behavior has a corresponding abstract one. The mapping plays the role of "one half" Milner's bisimulation [Mil89]. The above property can be proved by structural induction over the formula.

We can depict the verification process of a property φ for the concrete system $\mathcal{C}$,on the following steps:

1. proving that the mapping is an morphism (by structural induction over the formula), with the theorem prover (Coq),
2. proving φ for the abstract system (in the general case, assuming some extra hypothesis) using the model checker (SPIN),
3. and proving that assumptions on the abstract system (used in previous point) holds on the concrete system too.

Last condition generally has to be checked with a theorem prover.

We can formally prove that the above conditions allows us to state that the concrete system verifies the property.

6 Spin verification example

In this section we introduce our Spin implementation of Electre by means of the well-known example of the readers-writers. We can model this example with the following Electre program:

```
loop
  await
     { l1:L1 || l2:L2 }
  or
     w:W
end loop
```

This program models the variant in which writer (noted `W`) has a priority over readers (noted `L1` and `L2`). Lower case labels; `l1, l2` and `w` model requests for reading by reader 1 or 2, and for writing respectively.

Using our Electre to SPIN compiler we produce the PROMELA program implementation presented appendix A. As explained in Sect.4, the FIFO-list is implemented as independent process (noted `Queue`), and the process `Environment` produces undeterministically events `l1, l2` or `w`. In our example, each instantiation has its own name and its own message channel to communicate with the `Controller` process.

6.1 Checking safety and liveness properties

The Spin model checker is designed to deal with LTL formulae compatible with partial order reductions.

Safety. The main safety property we want to check is: *never a writer will write while one or two readers are reading.*

In our model, we translate this sentence into the formula:

$$\begin{aligned}\Box\neg\quad &(((state\_L1 == Act) \vee (\texttt{state\_L2} == \texttt{Act}))\\ &\wedge(\texttt{state\_W} == \texttt{Act}))\end{aligned}$$

This property is successfully checked by Spin.

Liveness. As in the general case, checking liveness properties is a bit harder. An interesting liveness property to check is: *every request for reading is eventually satisfied.*

In LTL *is* written:

$$\Box((\texttt{var\_l1} == \texttt{true}) \rightarrow \Diamond(\texttt{state\_L1} == \texttt{Act}))$$

To understand the formula we must note that the boolean variable `var_l1` is false if an event `l1` has not arisen or if it arose, it was "consumed". In the Electre context, we say that an event is consumed if the module activated by this event is terminated (see in appendix A, the code of module `L1`: in state `Act` when it terminates the variable `var_l1` is reset to false). Naturally, process `Environment` sets variable `var_l1` to true when generating a `l1` event.

When we first tried to check the above formula, we failed and Spin shows us a path in which the event `l1` is followed by the event `w`. The former activates module `L1`, then the latter interrupts it and launches the writer

`W`. Then, a cycle violating the property is shown where the environment produces event `l1` and the writer never ends! This simple counter-example shown us that our implementation do not respect the Electre hypothesis: All modules must have a finite execution. To solve this problem we added a "progress" labels, to force the checker to eliminate non-progress cycles, and the property was eventually proved.

The reader can find another explanation of the correctness proof with SPIN of the reader-writers example, using a morphism abstraction, in the paper [Arg98].

7 Conclusion

We have shown in Sect.3 how to exploit the semantic model of the source and target language to ensure compilation correctness. We think that this is an interesting application of theorem provers based on constructive logic. Moreover, when the language is being enriched with a new operator, together with its semantic rules, the new compiler can be derived from the previous one, according to some identified transformations. This system has been achieved and experimented with. Indeed, it provides a convenient tool to create new constructions, or to create custom-sized compilers for the language.

Furthermore, Sect.5 sketches a general methodology for the application of the semantic model to the proof of *safety* or *liveness* properties with LTL. The goal is to give techniques to deal with unbounded systems by means of abstractions. The whole theory formalization and its practical application are still in progress.

References

[AMR96] Pablo Argón, John Mullins, and Olivier Roux. Correct compiler construction using coq. In *CADE-13 Workshop on Proof Search in Type Theoretic Languages*, New Brunswick (USA), July 1996. Rutgers University.

[Arg95] Pablo Argón. Sémantique opérationnelle d'electre par réécritures : la version 4 du compilateur. Technical Report 95-11, Laboratoire d'Automatique de Nantes, LAN, 1 rue de la Noé, 44072 Nantes cédex 03, (France), October 1995.

[Arg98] Pablo Argón. Implémentation et vérification de programmes réactifs avec spin. In *Renpar–98 : 10-ièmes Rencontres Francophones du Parallélisme*, Strasbourg (FR), Juin 1998. Submitted.

[CCF+95] C. Cornes, J. Courant, J.C. Filiâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq proof assistant, reference manual. Technical Report 0177, INRIA Sophia Antipolis, July 1995. Available at `ftp://ftp.inria.fr/INRIA/coq/V5.10/doc/`.

[CGL94] E. M. Clark, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CR95] F. Cassez and O. Roux. Compilation of the electre reactive language into finite transition systems. *Theoretical Computer Science*, 146, July 1995.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *Proc. IEEE 7th Symp. Logic in Computer Science, LNCS*, 1992.

[Hol91] Gerard Holzmann. *Design and validation of computer protocols.* Prentice-Hall, 1991.

[Hol97] Gerard Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[Mel96] Thomas Melhan. Some research issues in higher order logic theorem proving. In *Abstract of Lectures presented at the BRICS Autumn School on Verification.* BRICS, November 1996.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MM92] V. Menissier-Morain. The caml reference manual. Technical Report 141, INRIA, Sophia-Antipolis (France), july 1992.

[Par95] C. Parent. *Synthèse de preuves de programmes dans le calcul des constructions inductives.* PhD thesis, École Normale Supérieure de Lyon (France), 1995.

A Spin implementation example

In this appendix we give the complete PROMELA program correspondig to the example used in Sect.6. First, we explain some of the notations. For the PROMELA syntax, refer to [Hol91].

Notation and remarks. The defined type (noted `mtype`) encodes the message constants used for the communication between the `Controller` and the `Queue`. Messages have the structure: *command(argument)*, see the definition of the channel `queue`, line 17.
Commands are:

- `CRT` means that the argument is a control command,
- `REQ` request control command, means that the controller asks for an event (poping);
- `SND` send control command, means that the controller will send one or more events (pushing);
- `EMY` empty control command, informs that no (more) event is in the queue;
- `ESD` end-of-send, means that the controller has complete the sending;
- `YES` acknowledgment command, informs the queue that the controller consumed the given event;
- `MULT` multiple command, informs the queue that the argument is an multiple memorized event,
- `NORM` normal command, informs the queue that the argument is an "normally" memorized event, i.e. at least one time.

The remaining codes are the alias for the events `l1`, `l2` and `w`. Notice that the process `Queue` has three main states, it can be "waiting" (line 54) for a command (request or send); treating a request command (line 60); or treating a send command (line 76). The events are keeped into the vector `fifo`.

From the line 106 to the line 180, we have the instanciations of the generic process module to `L1`, `L2`, and `W`, respectively.

Promela listing.

```
/**     ELECTRE to SPIN version0.7 - Oct. 1997 - IRCyN         **
 **                    (U.M.R. 6597)                           **
 **         1, rue de la noe NANTES CEDEX 03 FRANCE            **
 **               electre@lan.ec-nantes.fr                     **/
/*

Source Electre program:
[[1/{l1:L1~l1|||l2:L2~l2}]^{w:W~w}]*
*/
#define MAX        3  /* Fifo-list length  */
#define PRESENT        1

#define true        1
#define false        0
mtype = {CTR, REQ, EMY, SND, ESD, YES,  MULT, NORM,
         l1ff, l2ff, wff};
chan queue = [0] of {mtype,mtype};

#define NoAct     96
#define Start     97
#define Act       98
#define Int       99
#define end       100
#define w         101
#define l2        102
#define l1        103

chan env= [1] of {byte};

/* Module Channel & State var definitions */
chan c_W= [0] of { byte };
byte state_W= NoAct;
chan c_L2= [0] of { byte };
byte state_L2= NoAct;
chan c_L1= [0] of { byte };
byte state_L1= NoAct;

/*========> Process Definitions <==============*/
init {
run L1(c_L1);
run L2(c_L2);
run W(c_W);
}

/* Process Fifo */

```

```
active proctype Queue()
{
mtype fifo[MAX];
byte nb = 0;
byte cy ; bool t;
mtype x,y;

waiting:
if
 :: queue?CTR(REQ) -> goto requesting
 :: queue?CTR(SND) -> goto reception
 :: timeout -> goto error
fi;
requesting:
cy=0;
do
 ::(nb==0) -> queue!CTR(EMY) ; goto waiting
 ::(cy<nb) -> queue!CTR(fifo[cy]);
              queue?CTR(x)-> atomic{
                              if
                               ::(x==YES)->
                                  do
                                    ::(cy<nb)->fifo[cy]=fifo[cy+1];cy++
                                    ::else   ->nb--;goto waiting
                                  od
                               ::else -> cy++
                              fi}
 ::else -> queue!CTR(EMY) ;goto waiting
od;
reception:
queue?y(x) -> atomic{
                if

                 ::(x==ESD) -> goto waiting
                 ::else -> if
                            ::(y==MULT)-> fifo[nb]=x;nb++; goto reception
                            ::else ->cy=0;t=0;
                                     do
                                      ::(fifo[cy]==x)-> goto reception
                                      ::(cy>nb)->fifo[nb]=x;nb++;
                                                 goto reception
                                      ::else->cy++
                                     od;
                           fi;
                fi
                };
error: skip
```

```
}
active proctype Environment()
{
do
   :: env!l1 -> var_l1=true
   :: env!l2
   :: env!w
   :: timeout -> break
od;
end:skip
}

proctype L1(chan q)
{
state_L1 = NoAct;
if
  :: atomic{q?Start ->  state_L1 = Act; goto Actif}
  :: atomic{q?Act   ->  state_L1 = Act; goto Actif}
fi;
Actif:
progress_L1:
       do
  :: atomic{ q?Act ->  state_L1 = Act; goto Actif}
  :: atomic{ q?Int ->  state_L1 = Int; goto Stopped}
  :: atomic{ q!end ->  state_L1 = end ; var_l1=false; goto End }
       od;
Stopped:
         if
  :: atomic{ q?Act   -> state_L1 = Act; goto Actif}
  :: atomic{ q?Start -> state_L1 = Act; goto Actif}
         fi;
End:
      if
  :: atomic{ q?Act   ->  state_L1 = Act; goto Actif}
  :: atomic{ q?Start ->  state_L1 = Act; goto Actif}
      fi
}
proctype L2(chan q)
{
state_L2 = NoAct;
if
  :: atomic{q?Start ->  state_L2 = Act; goto Actif}
  :: atomic{q?Act   ->  state_L2 = Act; goto Actif}
fi;
Actif:
progress_L2:
       do
```

```
  :: atomic{ q?Act ->  state_L2 = Act; goto Actif}
  :: atomic{ q?Int ->  state_L2 = Int; goto Stopped}
  :: atomic{ q!end ->  state_L2 = end ; goto End }
       od;
Stopped:
         if
  :: atomic{ q?Act   -> state_L2 = Act; goto Actif}
  :: atomic{ q?Start -> state_L2 = Act; goto Actif}
         fi;
End:
      if
  :: atomic{ q?Act   ->  state_L2 = Act; goto Actif}
  :: atomic{ q?Start ->  state_L2 = Act; goto Actif}
      fi
}
proctype W(chan q)
{
state_W = NoAct;
if
  :: atomic{q?Start ->  state_W = Act; goto Actif}
  :: atomic{q?Act   ->  state_W = Act; goto Actif}
fi;
Actif:
progress_W:
       do
  :: atomic{ q?Act ->  state_W = Act; goto Actif}
  :: atomic{ q?Int ->  state_W = Int; goto Stopped}
  :: atomic{ q!end ->  state_W = end ; goto End }
       od;
Stopped:
         if
  :: atomic{ q?Act   -> state_W = Act; goto Actif}
  :: atomic{ q?Start -> state_W = Act; goto Actif}
         fi;
End:
      if
  :: atomic{ q?Act   ->  state_W = Act; goto Actif}
  :: atomic{ q?Start ->  state_W = Act; goto Actif}
      fi
}
active proctype Controller()
{
 mtype x;
/* No process to Start up ! */

St0:
do
```

```
::queue!CTR(REQ);
  queue?CTR(x);
  if
   ::(x==EMY)->  break
   ::(x==l1ff) -> queue!CTR(YES);  c_L1!Act; goto St1
   ::(x==l2ff) -> queue!CTR(YES);  c_L2!Act; goto St2
   ::(x==wff) -> queue!CTR(YES);  c_W!Act; goto St3
   ::else    -> skip
  fi
od;
do
:: env?l1 -> c_L1!Act; goto St1
:: env?l2 -> c_L2!Act; goto St2
:: env?w -> c_W!Act; goto St3
od;

St3:
do
::env?l1  -> queue!CTR(SND);
             queue!NORM(l1ff);
             queue!CTR(ESD)
::env?l2  -> queue!CTR(SND);
             queue!NORM(l2ff);
             queue!CTR(ESD)
::c_W?end -> goto St0
od;

St2:
do
::queue!CTR(REQ);
  queue?CTR(x);
  if
   ::(x==EMY)->  break
   ::(x==l1ff) -> queue!CTR(YES);  c_L1!Act; goto St4
   ::(x==wff) -> queue!CTR(YES);  c_L2!Int; c_W!Act; goto St3
   ::else    -> skip
  fi
od;
do
::c_L2?end -> goto St0
:: env?l1 -> c_L1!Act; goto St4
:: env?w -> queue!CTR(SND);
            queue!NORM(l2ff);
            queue!CTR(ESD);
            c_L2!Int; c_W!Act; goto St3
od;

```

```
St4:
do
::queue!CTR(REQ);
  queue?CTR(x);
  if
   ::(x==EMY)->  break
   ::(x==wff) -> queue!CTR(YES);  c_L1!Int; c_L2!Int; c_W!Act; goto St3
   ::else    -> skip
  fi
od;

do
::c_L1?end -> goto St5
::c_L2?end -> goto St6
:: env?w -> queue!CTR(SND);
            queue!NORM(l1ff);
            queue!NORM(l2ff);
            queue!CTR(ESD);
            c_L1!Int; c_L2!Int; c_W!Act; goto St3
od;

St6:
do
::queue!CTR(REQ);
  queue?CTR(x);
  if
   ::(x==EMY)->  break
   ::(x==wff) -> queue!CTR(YES);  c_L1!Int; c_W!Act; goto St3
   ::else    -> skip
  fi
od;
do
::env?l2  -> queue!CTR(SND);
             queue!NORM(l2ff);
             queue!CTR(ESD)
::c_L1?end -> goto St0
:: env?w -> queue!CTR(SND);
            queue!NORM(l1ff);
            queue!CTR(ESD);
            c_L1!Int; c_W!Act; goto St3
od;

St5:
do
::queue!CTR(REQ);
  queue?CTR(x);
  if
```

```
   ::(x==EMY)->  break
   ::(x==wff) -> queue!CTR(YES);  c_L2!Int; c_W!Act; goto St3
   ::else    -> skip
  fi
od;
do
::env?l1  -> queue!CTR(SND);
             queue!NORM(l1ff);
             queue!CTR(ESD)
::c_L2?end -> goto St0
:: env?w -> queue!CTR(SND);
            queue!NORM(l2ff);
            queue!CTR(ESD);
            c_L2!Int; c_W!Act; goto St3
od;

St1:
do
::queue!CTR(REQ);
  queue?CTR(x);
  if
   ::(x==EMY)->  break
   ::(x==l2ff) -> queue!CTR(YES);  c_L2!Act; goto St7
   ::(x==wff) -> queue!CTR(YES);  c_L1!Int; c_W!Act; goto St3
   ::else    -> skip
  fi
od;
do
::c_L1?end -> goto St0
:: env?l2 -> c_L2!Act; goto St7
:: env?w -> queue!CTR(SND);
            queue!NORM(l1ff);
            queue!CTR(ESD);
            c_L1!Int; c_W!Act; goto St3
od;

St7:
do
::queue!CTR(REQ);
  queue?CTR(x);
  if
   ::(x==EMY)->  break
   ::(x==wff) -> queue!CTR(YES);  c_L1!Int; c_L2!Int; c_W!Act; goto St3
   ::else    -> skip
  fi
od;
do
```

```
::c_L1?end -> goto St5
::c_L2?end -> goto St6
:: env?w -> queue!CTR(SND);
            queue!NORM(l2ff);
            queue!NORM(l1ff);
            queue!CTR(ESD);
            c_L1!Int;
            c_L2!Int;
            c_W!Act;
            goto St3
od;
end: skip
}
```