Reactive EFSMs – Reactive Promela/RSPIN

Elie NAJM*
Frank OLSEN*†

ABSTRACT *Reactive Promela*/RSPIN is an extension to the protocol validator *Promela*/SPIN. It enhances the simulation and verification capabilities of SPIN by allowing modular specifications to be analysed while alleviating the state-space explosion problem. *Reactive Promela* is a simple reactive language. The tool RSPIN is a preprocessor for SPIN which translates a *Reactive Promela* specification into a corresponding *Promela* specification. The main function performed by RSPIN is to combine *configurations of Reactive Promela automata* into *Promela proctypes*. The translated specification can then be simulated and verified using SPIN. We present our ideas first in a formal setting then we discuss their implementation in *Reactive Promela* and RSPIN concrete syntax and tool.

1 Introduction

When considering the problem of specification, (de)composition is a central issue. *Promela* provides for two styles of composition of automata: loosely coupled (communication is by FIFO queues) and tightly coupled (communication is by rendezvous). A third style, the synchronous reactive style, has been widely advocated and used in the literature and in industry.

In the synchronous reactive style, a configuration of automata reacts to external events in a synchronous way: a collection of external events is treated thoroughly by the configuration before another collection of events is taken and processed. In other words, the reactive configuration reacts to input events, and it is only at the end of the reaction that new inputs can be considered and processed. This kind of processing is valid when the speed of a reaction is higher than the delay between two consecutive input events.

The reactive style allows for powerful decomposition of specifications, beyond what is possible with merely rendezvous between automata. Fur-

*Ecole Nationale Supérieure des Télécommunications, 46, rue Barrault, 75013 Paris, France, E-mail: {najm,olsen}@res.enst.fr

†This author is now also affiliated with France Telecom–CNET, PAA/TSA/TLR, Issy-les-Moulineaux, Paris, France, E-mail: olsen@ issy.cnet.fr

thermore, it reduces the state space explosion by constraining parallelism between automata.

Whereas Holzmann in [Hol91] proposes ways of reducing the complexity of systems (by incremental composition, minimization, generalization, atomic sequences, layering and structuring techniques, and so on), this is not a feature of the *Promela* language in itself. Instead it is a guideline for how to use the language for large, complex systems.

This paper describes an extension to *Promela*, whereby reactive processes can be defined and instantiated. A reactive process is a configuration of synchronously composed automata. Besides the linguistic extension, this paper also describes a translation mechanism of reactive processes into *Promela* processes. This translation has been implemented in a preprocessor to SPIN, called RSPIN which translates a specification in *Reactive Promela* into an equivalent one in *Promela*.

In order to reason about the extension and about the translation from *Reactive Promela* to *Promela* we use a formalisation of an essential subset of *Promela* (see section 2). In section 3 we formalise the reactive extension model. In section 4 we use these models to formalise the translation algorithms. Section 5 then presents a concrete syntax for *Reactive Promela* with an example. The same section also introduces the RSPIN tool through an example which demonstrates the mapping from *Reactive Promela* to *Promela*.

2 Promela State Machines

The subset of *Promela* that consider can be formally defined using the following settings. We consider first a set L of elementary *Promela* instructions (with typical elements l):

$$L ::= [pred] \mid v = Exp \mid c!Exp \mid c?v$$

These instructions correspond to test, assignment, send, and receive instructions. The send and receive instructions are to bounded channels c. We consider simple channels containing FIFO sequences of simple values.

2.1 PSM processes

We consider PSM processes (with generic element P) as follows: P is a tuple (S, s, T, E) where:

- S is a set of *states*;
- s is either the *initial state* or the *current state*;
- $T \subset S \times L \times S$ is a *transition relation*;

- E is the *environment* of variables in P. It consists of a set of (untyped) variables V (ranging over values in the set VAL) and a mapping of each variable $v \in V$ to a value $val \in VAL$.

We use a dot-notation to access elements of the tuple representing P. $P.T$ is the transition relation T of PSM process P and $P.E$ is the environment of P.

2.2 Semantics of PSM processes

We give the semantics of a PSM process P by a translation function from PSM into a *Labeled Transition System (LTS)* defined by: $(PSM \times \Gamma \times PSM)$[1]. A PSM transition (P, α, P'), also written $P \xrightarrow{\alpha} P'$, means that P can perform action α to become P'.

The following notations are used in the rules:

- $E \vdash Exp \rightarrow val$ denotes that the expression Exp evaluates to val in the environment E.

- $E \oplus (v : val)$ denotes an environment E obtained from E by updating variable v to val.

The translation from PSM to LTS is given by the following set of SOS rules:

TEST

$$\frac{(s, [pred], s') \in T \quad E \vdash pred \rightarrow True}{(S, s, T, E) \xrightarrow{\epsilon} (S, s', T, E)}$$

ASSIGNMENT

$$\frac{(s, v = Exp, s') \in T \quad E \vdash Exp \rightarrow val}{(S, s, T, E) \xrightarrow{\epsilon} (S, s', T, E \oplus (v : val))}$$

SEND

$$\frac{(s, c!Exp, s') \in T \quad E \vdash Exp \rightarrow val}{(S, s, T, E) \xrightarrow{c!val} (S, s', T, E)}$$

[1]where $\Gamma ::= \epsilon \mid c!val \mid c?val$, with a typical element α.

RECEIVE

$$\frac{(s, c?v, s') \in T}{(S, s, T, E) \xrightarrow{c?val} (S, s', T, E \oplus (v : val))}$$

2.3 Atomic PSM processes

We now consider a larger subset of *Promela* containing the *atomic*{...} construct. Consequently, our model is extended to reflect this construct. We define atomic PSM processes[2] (with typical element Q is a triple (P, Π, σ) where:

- P is a PSM process.
- Π is a *partioning* of $P.S$ into a set of disjoint non-empty sets of states, i.e. $\forall p_1, p_2 \in \Pi : p_1 \cap p_2 = \emptyset$ where $p_1 \in P.S, p_2 \in P.S$. Note that there may be some states in $P.S$ that are not in any partition $p \in \Pi$.
- $\sigma = \downarrow \emptyset \mid \downarrow p \mid \uparrow p$ is the current *atomic section* of P. $\downarrow \emptyset$ denotes that P is not in an atomic section, $\downarrow p$ denotes that P has entered atomic section p but is not yet active, and $\uparrow p$ denotes that P is active (executing a sequence of atomic steps) in p.

2.4 Semantics of atomic PSM processes

The semantics of atomic PSM processes is given by two rules. We use the function $\Pi(s)$ defined by $\Pi(s) = p \in \Pi$ (for $s \in p$) and $\Pi(s) = \emptyset$ (for $s \notin \bigcup_{p \in \Pi} p$).

DEACTIVATED–ATOMIC–SEQUENCE

$$\frac{P \xrightarrow{\alpha} P' \quad (\Pi(P'.s) \neq p \vee \Pi(P'.s) = \emptyset)}{\begin{array}{c}(P, \Pi, \downarrow p) \xrightarrow{\alpha} (P', \Pi, \downarrow \Pi(P'.s)) \\ (P, \Pi, \uparrow p) \xrightarrow{\alpha} (P', \Pi, \downarrow \Pi(P'.s))\end{array}}$$

[2] we do allow for non-atomic PSM processes as a special case where $Q = (P, \emptyset, \downarrow \emptyset)$.

ACTIVATED–ATOMIC–SEQUENCE

$$\frac{P \xrightarrow{\alpha} P' \quad p \neq \emptyset \quad \Pi(P'.s) = p}{\begin{array}{c}(P, \Pi, \downarrow p) \xrightarrow{\alpha} (P', \Pi, \uparrow p) \\ (P, \Pi, \uparrow p) \xrightarrow{\alpha} (P', \Pi, \uparrow p)\end{array}}$$

Figure 1 illustrates the rules and shows the different cases for activation/deactivation of atomic sequences. The transitions numbered (1) to (6) correespond to the first rule (DEACTIVATED–ATOMIC–SEQUENCE), whereas transitions (7) and (8) correspond to the second rule (ACTIVATED–ATOMIC–SEQUENCE).

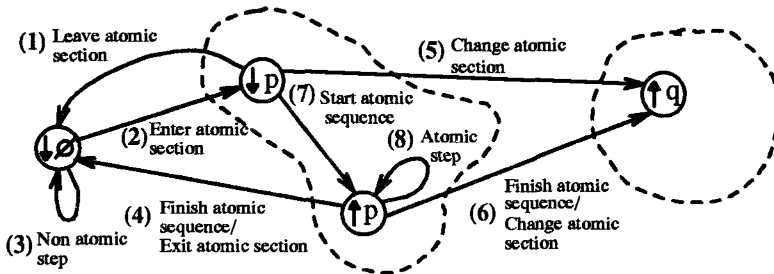

FIGURE 1. Allowable atomic transitions.

2.5 *PSM specifications*

We now turn to complete PSMs. A PSM specification is a pair $(Procs, Chans)$ where:

- *Procs* is a set of atomic PSM processes;
- *Chans* is a set of bounded FIFO channels. If the length of the channel is zero then communication is by rendez–vous, else it is by asynchronous message passing.

2.6 *Semantics of PSM specifications*

We use the following notations in the rules: $Int(Q)$ means that Q is *interruptible* (i.e. for $Q = (P, \Pi, \sigma) : \sigma = \downarrow \emptyset$), $l(c)$ gives the current number of values (messages) in channel c, $\#c$ gives the length of channel c, and $head(c)$ gives the value at the head of the channel c.

SPEC–ASYNCH–SEND

$$\frac{Q_i \xrightarrow{c!v} Q_i' \quad \forall j \neq i : Int(Q_j) \quad l(c) < \#c \quad \#c \neq 0}{(Chans, Procs) \longrightarrow (Chans', Procs')}$$

where $Procs' = (Procs - \{Procs'\}) \cup Q_i'$
$Chans' = Chans - \{c(v_1, \ldots, v_n)\} \cup \{c(v_1, \ldots, v_n, v)\}$

SPEC–ASYNCH–RECEIVE

$$\frac{Q_i \xrightarrow{c?v} Q_i' \quad \forall j \neq i : Int(Q_j) \quad l(c) > 0 \quad head(c) = v}{(Chans, Procs) \longrightarrow (Chans', Procs')}$$

where $Procs' = (Procs - \{Procs'\}) \cup Q_i'$
$Chans' = Chans - \{c(v_1, \ldots, v_n, v)\} \cup \{c(v_1, \ldots, v_n)\}$

SPEC–RENDEZ-VOUS

$$\frac{Q_i \xrightarrow{c!v} Q_i' \quad Q_j \xrightarrow{c?v} Q_j' \quad \forall k \neq i : Int(Q_k) \quad \#c = 0}{(Chans, Procs) \longrightarrow (Chans, Procs')}$$

where $Procs' = (Procs - \{Procs'\}) \cup Q_i' \cup Q_j'$

The condition that even process Q_j is interruptible is a very strict interpretation of atomic instructions. It prevents rendez–vous communication if the two processes involved are both in an atomic sequence. The rule does permit Q_i to leave, exit or change atomic section while Q_j starts an atomic section.

SPEC–INTERNAL

$$\frac{Q_i \xrightarrow{\epsilon} Q_i' \quad \forall j \neq i : Int(Q_j)}{(Chans, Procs) \longrightarrow (Chans, Procs')}$$

where $Procs' = (Procs - \{Procs'\}) \cup Q_i'$

3 Reactive State Machines

Here we present the *Reactive State Machine (RSM)*, a formalisation of our proposed extension to *Promela*.

Like we did for PSMs, we give the syntax and semantics in an incremental fashion, starting with *RSM automata*, then *RSM processes*, and finally we give the semantics of complete *RSM specifications*. The set of instructions L is the same as the set used in section 2.

3.1 RSM automata

In the RSM model a process can be decomposed into a configuration of *RSM automata*. An RSM automaton, A, is a tuple $(S, s, spred, T, E, I)$ where:

- S is a set of states partioned into two disjoint subsets[3]:
  - $SS \subseteq S$ a set of *stable states*
  - $TS \subset S$ a set of *transitory states*
- s is either the *initial state* $\in SS$ or the *current state*.
- $spred : s \longrightarrow \{True, False\}$ is a function to determine if a give state is stabel or transitory. $spred(s) = True$ if $s \in SS$ and $spred(s) = False$ if $s \in TS$.
- $T \subset S \times L \times S$ is a *transition relation*
- E is the *environment* of variables in A.
- I is an *interface*[4] consisting of:
  - P_{in} a set of *inports*
  - P_{out} a set of *outports*

A *well-formed* RSM automaton has the following restrictions:

- if $(s, l, s') \in T$ and $s \in SS$ then l is an input action.
- if $(s, l, s') \in T$ and $l = c?x$ then $c \in P_{in}$.
- if $(s, l, s') \in T$ and $l = c!val$ then $c \in P_{out}$.

[3]where $S = SS \cup ST$ and $SS \cap ST = \emptyset$

[4]where $P_{in} \cap P_{out} = \emptyset$.

3.2 *Semantics of RSM automata*

The semantics of an RSM automaton is given by the same rules as for a PSM process.

3.3 *RSM processes*

An RSM process R is a tuple (A, I, L) where:

- A is a set of RSM automata $\{a_1, \ldots, a_n\}$
- I an *interface*[5] consisting of:
  - C_{in} a set of *input channels*
  - C_{out} a set of *output channels*
- L is a set of *links* taking the following three forms:
  - L_{int} a set of *internal links* represented by the tuple $((a_i, p_i), (a_j, p_j))$, where $p_i \in a_i.P_{out}$ and $p_j \in a_j.P_{in}$.
  - L_{out} a set of *external output links* represented by the tuple $((a_i, p_i), (\bullet, p_j))$, where $p_i \in a_i.P_{out}$ and $p_j \in C_{out}$.
  - L_{in} a set of *external input links* represented by the tuple $((\bullet, p_i), (a_j, p_j))$, where $p_i \in C_{in}$ and $p_j \in\in a_j.P_{in}$.

Some other definitions:

- the *state* of an RSM process R is the tuple $R.s = (a_1.s, \ldots, a_n.s)$, where:
  - the *initial state* is defined as the tuple $(a_1.s_0, \ldots, a_n.s_0)$
  - a state $s = (s_1, \ldots, s_n)$ is a *stable state* iff $\forall s_i : spred_i(s_i) = True$.
  - a state $s = (s_1, \ldots, s_n)$ is a *transitory state* iff $\exists s_i : spred_i(s_i) = False$.

3.4 *Semantics of RSM processes*

We add the predicate $stable(R) = R.s$ is a stable state. and give the SOS rules for RSM processes:

---

[5] where $C_{in} \cap C_{out} = \emptyset$

RSM–PROC–EXT-SEND

$$\frac{a_i \xrightarrow{g!val} a_i' \quad ((a_i, g), (\bullet, c)) \in L_{out} \quad c \in C_{out}}{(A, I, L) \xrightarrow{c!val} (A', I, L)}$$

where $A = a_1, \ldots, a_i, \ldots, a_n$
$A' = a_1, \ldots, a_i', \ldots, a_n$

RSM–PROC–EXT-RECEIVE

$$\frac{a_i \xrightarrow{g?v} a_i' \quad ((\bullet, c), (a_i, g)) \in L_{in} \quad c \in C_{in} \quad stable((A, I, L))}{(A, I, L) \xrightarrow{c?v} (A', I, L)}$$

where $A = a_1, \ldots, a_i, \ldots, a_n$
$A' = a_1, \ldots, a_i', \ldots, a_n$

RSM–PROC–SYNCH

$$\frac{a_i \xrightarrow{g!val} a_i' \quad a_j \xrightarrow{h?v} a_j' \quad ((a_i, h), (a_j, g)) \in L_{int}}{(A, I, L) \xrightarrow{\epsilon} (A', I, L)}$$

where $A = a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n$
$A' = a_1, \ldots, a_i', \ldots, a_j', \ldots, a_n$

A synchronisation between automata a_i and a_j is possible only if a_I can send a message on outport p, a_I can receive a message on inport q, and there is a link between (a_i, p) and (a_j, q).

RSM–PROC–INTERNAL–ACTION

$$\frac{a_i \xrightarrow{\epsilon} a_i'}{(A, I, L) \xrightarrow{\epsilon} (A', I, L)}$$

where $A = a_1, \ldots, a_i, \ldots, a_n$
$A' = a_1, \ldots, a_i', \ldots, a_n$

3.5 RSM specifications

An RSM specification is a triple $(RProcs, Procs, Chans)$ where:

- *RProcs* is a set of RSM processes;
- *Procs* is a set of atomic PSM processes;
- *Chans* is a set of channels.

3.6 Semantics of RSM specifications

The semantics of RSM specifications can simply be given using the rules for PSM specifications where the predicate $Int(R)$ is defined on RSM process R by $Int(R) = stable(R)$.

4 Translating RSMs to PSMs

In the translation of an RSM specification to a PSM specification, each RSM process is translated to a PSM process through a technique of *automata combination*. The next section shows how we use these techniques to extend *Promela* with reactive processes (*rproctypes*), and RSPIN a tool which performs the translation of reactive processes to *Promela* processes (*proctypes*).

The algorithm for automata combination uses standard state space search techniques to create the global state graph representing the combined automaton. The only main difference is that the combined automaton is trimmed from all transitions starting from transitory state vectors and having external inputs.

Once the combined automaton have been obtained we use another algorithm to partion it into disjoint atomic sections. The purpose is to make each possible reaction of the combined automaton atomic. We should note here that the intended behaviour of atomic reactions is only valid when communication is by asynchronous FIFO channels[6].

We can now give the partioning algorithm, using the notation $s - \bullet \rightarrow s'$ to denote the fact that there exists a path from state s to transitory state s', traversing only transitory states. The algorithm is then:

$$\text{For every stable state } s, \Pi(s) =_{def} \{s\} \cup \{s' | s - \bullet \rightarrow s'\}$$

We must show that $\Pi(s)$ does not create overlapping partitions. We claim that:

---

[6]In the RSPIN tool there is an option which prevents the partioning from taking place (on a global or a per process basis). This gives the user more flexibility to decide whether to allow rendez-vous communication for reactive processes.

$$\forall s_1, s_2 : \Pi(s_1) \cap \Pi(s_2) = \emptyset$$

The proof is obtained by induction on the length of paths $s_1 - \bullet \rightarrow s$ and $s_2 - \bullet \rightarrow s$ and using the fact that transitions of RSM automata are restricted (syntactically) so that this condition holds:

If $((s_1, l_1, s), (s_2, l_2, s)) \in T \times T$, and if s is a transitory state, then $(s_1, l_1) = (s_2, l_2)$.

5 Reactive Promela and RSPIN

We now present *Reactive Promela*, our proposed extension to *Promela*, and the associated RSPIN tool.

5.1 *Syntax of Reactive Promela*

The syntax of *Reactive Promela* strongly resembles that of *Promela*, since the aim is to make it as easy as possible to use the extension. The only new keywords added in *Reactive Promela* are the following:

```
automaton   in          inport     link
outport     rproctype   external
```

Below we present the parts of the *Reactive Promela* grammar where it extends the *Promela* grammar. First, a few words on the notation. The new keywords are displayed in capitals (`RPROCTYPE`), tokens and *Promela* keywords are enclosed within apostrophes (`':'` and `'goto'`), names (references) are displayed in lowercase letters within `< ... >` (`<rproc_name>`) and non-terminals in lowercase letters (`r_proc`). Also, `{ ... }+` means one or more of the enclosed unit and `{ ... }*` means zero or more units. Units enclosed by `[ ... ]` are optional.

In *Reactive Promela*, the old process definition: `proc ::= PROCTYPE ...` is replaced by: `proc ::= p_proc | r_proc`, where `p_proc` is the usual *Promela* process, and `r_proc` is the *Reactive Promela* process defined by:

```
r_proc       ::= RPROCTYPE <rproc_name>
                 '(' r_interface ')' r_body
r_interface ::= {r_port_decl}*
r_port_decl ::= INPORT <port_name> | OUTPORT <port_name>
```

```
r_body       ::= {automaton}+ links
automaton    ::= AUTOMATON <autom_name>
                 '(' a_interface ')' a_body
a_interface ::= {a_port_decl}*
a_port_decl ::= EXTERNAL INPORT  <port_name>
               | EXTERNAL OUTPORT <port_name>
               | INPORT  <port_name> port_init
               | OUTPORT <port_name> port_init
port_init   ::= '=' '{' type_list '}'
a_body      ::= '{' {one_decl | a_stmnt}+ '}'
a_stmnt     ::= <port_name> '?' {<var_name>|const}+
               | <port_name> '!' {a_expr}+
               | <label_name> ':' a_stmnt
               | 'goto' <label_name>
               | <var_name> '=' aexpr
               | 'if' options 'fi'
               | 'do' options 'od'
links       ::= LINK '{' {link}+ '}'
link        ::= port '=>' {port}+
port        ::= <port_name> IN <autom_name>
```

The body of an automaton is defined as **a_body**, which is the same as **body** in *Promela* except that **a_body** only allows (for the time being) a subset of the rules of *Promela* (listed in the rule for **a_stmnt**). We have not shown the rule for **a_expr** but it allows most of the usual *Promela* expressions, at least for arithmetic and boolean operations.

To introduce the *Reactive Promela* language and the syntax shown above, we give a simple example of a *Reactive Promela* specification. It consists of a single *rproctype* which encapsulates two *automata*. Figure 2 below shows the specification in a graphical notation for *Reactive Promela*.

The graphical syntax makes it very easy to visualize a *Reactive Promela* specification, and it would certainly be interesting for the user to write or view a specification using a graphical interface. The textual syntax repres-nting the diagram of figure 2 is:

```
#define NO 0
#define YES 1

chan i1 = [10] of { int, int };
chan o1 = [10] of { bool }; chan o2 = [10] of { int,int };
chan l1 = [10] of { int }; chan l2 = [10] of { int };
chan l3 = [10] of { int }; chan l4 = [10] of { int };

rproctype A
   /* external interface definition */
  (inport  i1, l3; outport o1, l1, l2)
{
  automaton aut1
```

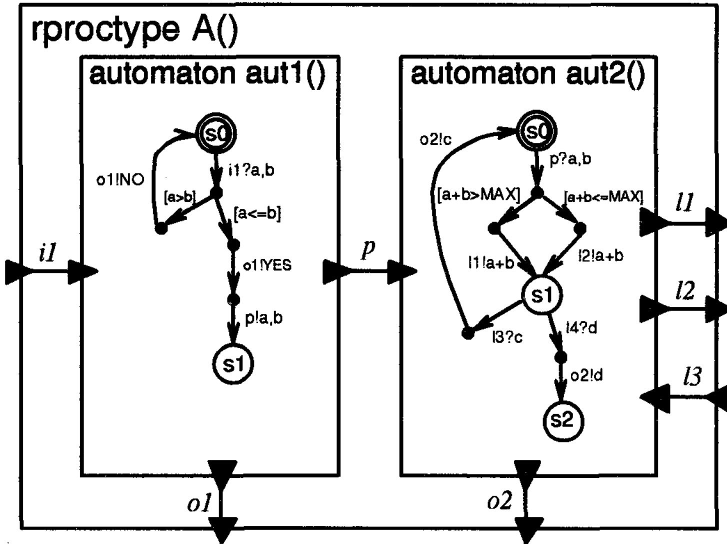

FIGURE 2. A simple rproctype decomposed into two automata.

```
   /* interface definition */
  (external inport i1; external outport o1;
   outport p = { int, int })
{
  int a, b;

 stable0:
  i1?a,b -> if
            :: [a > b]  -> o1!NO; goto stable0
            :: [a <= b] -> o1!YES; p!a,b; goto stable1
            fi;

 stable1: skip
}

automaton aut2
   /* interface definition */
  (external inport  l3, l4; external outport o2, l1, l2;
   inport  p = { int, int })
{
  int a, b, c, d, MAX = 10;

 stable0:
  p?a,b -> if
           :: [a+b > MAX]  -> l1!a+b; goto stable1
           :: [a+b <= MAX] -> l2!a+b; goto stable1
           fi;
```

```
  stable1:
   if
   :: l3?c -> o2!c; goto stable0
   :: l4?d -> o2!d; goto stable2
   fi

  stable2: skip
 }

 link { p in aut1 => p in aut2 }
}

...other rproctypes() and other Promela proctypes()
```

5.2 Reactive SPIN

In order to realistically check the correctness of a specification, tool support is essential. Instead of writing a simulator/verifier from scratch we propose to perform a mapping of *Reactive Promela* constructs into corresponding ones in *Promela*. For this purpose we present RSPIN, a preprocessor tool that translates a *Reactive Promela* specification into an equivalent *Promela* specification.

One of the most important aims we hope to achieve through the *Reactive Promela* extension is to provide a way to reduce the state-space explosion problem. There are two ways in which this is done. We have already seen in section 4 that the state space generated by combining RSM automata is not the full crossproduct of reachable states. The other means of reducing the state space is the use of *Promela*'s `atomic{...}` construct to implement the partitioning function $\Pi(s)$. The effect of this is that the proctypes generated by RSPIN reacts to inputs in an atomic fashion. This means that during simulation or verification, other proctypes are blocked until the reaction is over.

However, it must be noted that the use of atomic reactions is valid only if the channels used are of length greater than zero. In other words, rendez–vous communication is not possible during a reaction. To allow more flexibility we make the encapsulation of reactions as atomic sequences to be optional on a per rproctype basis.

To illustrate how automata are combined we present a part of a *Reactive Promela* specification for the HDLC protocol[7]. To model the protocol in *Reactive Promela* we decompose it into five automata two of which are

[7]The example is taken from a course in protocol specification given by Elie Najm at ENST. Modelisation and validation of this example will be presented in more detail elsewhere.

shown in figure 3 (the *Transmitter* and the *Window*)[8]. This example shows the benefit of decomposition: each automaton that make up the rproctype (i.e. the protocol) have a simple and clearly defined role, but we can still treat and reason about it as a whole.

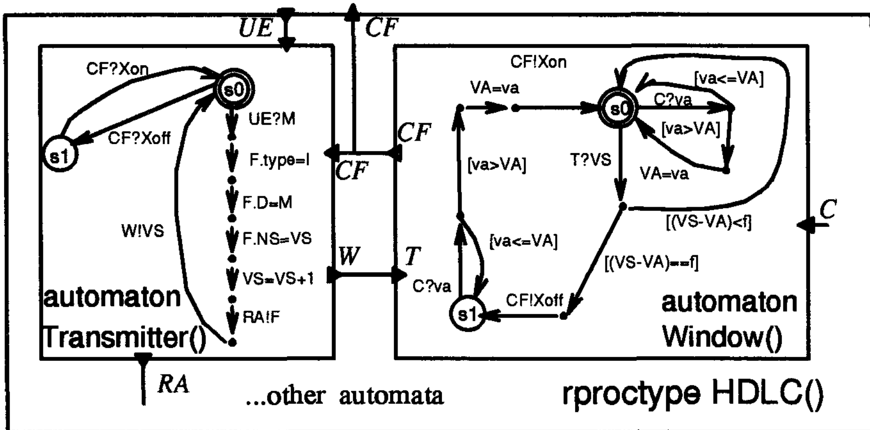

FIGURE 3. The transmitter and the window automata of the HDLC rproctype.

The transmitter can receive a message from the upper protocol (on inport `UE`). From this it creates a frame (the structure `F`) which it sends to the window (on outport `W`) as well as to the retransmitter and the acknowledger (on outport `RA`). The window's responsibility is to manage the sequence numbers. When the window is saturated, it notifies the transmitter and the upper protocol layer (on outport `CF`) that no more messages can be sent, until one is acknowledged. From the receiver (not shown) it receives a message (on inport `C`) that indicate that the last frame acknowledged.

After combining the two automata in the figure above we get the result shown in figure 4. In the textual syntax the combined automaton would be represented as a *Promela* proctype[9]:

```
proctype  combined_Transmitter_Window()
{
  Frame F; /* with fields: type, D, NS */
  int Window_VS, Transmitter_VS, VA, va, f;

 s0:
```

[8]The three automata not shown are the *Retransmitter*, the *Receiver* and the *Acknowledger*.

[9]We ignore the three other automata representing the HDLC protocol.

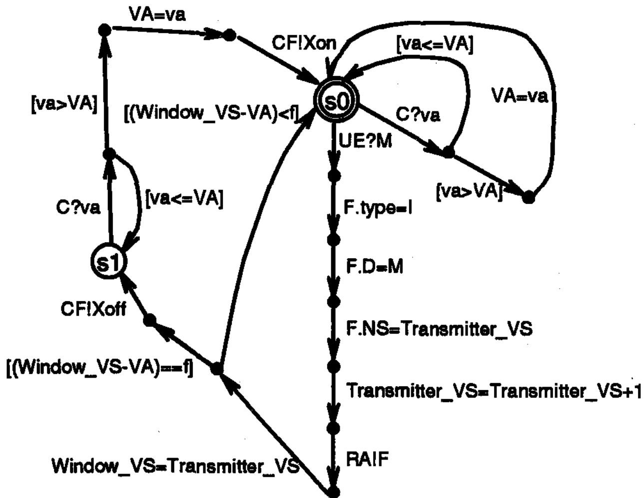

FIGURE 4. The resulting automaton after combining the transmitter and the window.

```
  if
  :: atomic {
       UE?M -> F.type=I; F.D=M; F.NS=Window_VS;
               Transmitter_VS=Transmitter_VS+1;
               RA!F; Window_VS=Transmitter_VS+1;
               if
               :: [(Window_VS-VA)==f] -> CF!Xoff; goto s1
               :: [(Window_VS-VA)<f] -> goto s0
               fi
     }
  :: atomic {
       C?va -> ...
       ...
     }
  fi

 s1:
  atomic {
    C?va -> ...
    ...
  }
}
```

We can make a few observations about this combined automaton. First

we note that the number of states has not increased significantly. This is because the combination process combines transitions and interleaves whole reactions rather than interleaving all transitions.

The stable states of the combined automaton is a subset of the crossproduct between the stable states of the individual automata. In figure 4 we note that the combined automaton has got two stable states. Stable state `s0` corresponds to the case where the window is not saturated and where the transmitter is waiting for a new message from the upper protocol layer. In this state the combined automaton is free to send more messages. The other one, `s1`, is when the window is saturated and transmission blocked.

Communications between the transmitter and the window is reduced to assignment to variables (e.g. the send statement `W!VS` in the transmitter and the corresponding receive statement in the window `T?VS` is reduced to the assignment `Window_VS=Transmitter_VS`). Since the variable `VS` exists in both the transmitter and in the window it is prefixed with the automaton name in the combined automaton. A more interesting case is where the window notifies both the transmitter and the upper protocol layer that it is saturated. This is an example of a communication with more than one receiver. Between the transmitter and the window this is a pure synchronization which takes the combined automaton to stable state `s1` (the saturation state). But we still need to notify the upper layer: therefore the combined automaton keeps the send action `CF!Xoff`. All other communication actions than those sent between the two automata in figure 4 are kept as they were.

To demonstrate more clearly the benefit provided by *Reactive Promela*'s decomposition technique we would have had to show the combined automaton for all the five automata in the HDLC protocol model have been (unfortunately too big to fit conveniently on one page).

6 Conclusion

In this paper we have presented a formalisation of a subset of *Promela* and of a reactive extension. Then we introduced the *Reactive Promela* language and its associated tool RSPIN. The language belongs to the family of synchronous reactive formalisms and allows a system to be decomposed into a reactive part containing configurations of synchronously communicating automata and a pro-active part containing *Promela* proctypes.

In relation to the other languages in the synchronous reactive family, *Reactive Promela* is an *imperative language*, similar to RC [Bou91], Esterel [BG92], [JLRM], and SL [BdS95]. These languages are distinguished by the notions of *state* and *sequences of statements* that lead from one state to

another. In contrast, in the ***data-flow languages*** like Lustre [CPHP85] and Signal [Gue86], the reaction to input events is evaluated as the solution of a set of equations. There are also some attempts to combine the imperative and data-flow approach, e.g. the language ArgoLus as described in [JLRM].

In *Reactive Promela*, like in the latest version of Esterel and in SL, we disallow hypotheses to be made about the presence or absence of a signal during the reaction. This means that a signal can only be taken into account after it has appeared. In *Reactive Promela* dynamic behavior is represented as extended finite automata, while Esterel and SL are special kinds of process algebras. There are two other major differences between Esterel and SL on one side and *Reactive Promela* on the other: The first is that Esterel and SL use broadcast communication while in *Reactive Promela* communication is via point-to-point channels. The second difference is that in *Reactive Promela*, reactions are triggered by one input event, while in Esterel and SL reactions are triggered by an arbitrary set of input events.

The RSPIN tool translates rproctypes into *Promela* proctypes, so that a *Reactive Promela* specification can be simulated and verified with SPIN. No modifications to the SPIN tool are needed to do this.

We also have some ideas for extensions to *Reactive Promela*/RSPIN. The next probable extensions are handle *Promela* code in RSPIN and to make a graphical interface for X-Windows which would allow the *Reactive Promela* user to write and view automata. A very useful tool for the user of RSPIN would be one which allows traces produced by SPIN to be "mapped to" the initial *Reactive Promela* specification. Other extensions might include an emacs mode for editing *Reactive Promela* specifications. More theoretical issues include the possibility for an rproctype to react to collections of inputs instead of to exactly one input. More work is needed on how properties of *Reactive Promela* specifications can be proven, although it seems that most of the SPIN techniques should be applicable (i.e. special labels, assertions and LTL-formulae). We will also look into how the partial-order reduction methods introduced in recent versions of SPIN can be used with *Reactive Promela* specifications, although we can conjecture that *Reactive Promela* provides this one kind of such reductions "for free": as we have seen in previous sections, RSPIN does not consider all the interleavings of transitions. Instead one interleaving is chosen since the set of outputs generated by the reaction will be the same. For more information on partial order reduction methods see [dSdS95, Pel94, GKPP95, Val90, WG93, God94].

7 References

[AF90] C. André and L. Fancelli. A mixed (asynchronous / synchronous) implementation of a real-time system. In *Euromicro 90, Amsterdam*, 1990.

[BB91] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1270–1282, 1991.

[BCGH93] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. Rapport de recherche 2089, INRIA, Unité de recherche INRIA Sophia-Antipolis, France., October 1993.

[BdS95] Frédéric Boussinot and Robert de Simone. The sl synchronous language. Rapport de recherche 2510, INRIA, Unité de recherche INRIA Sophia-Antipolis, France., Mars 1995.

[Ber93a] G. Berry. Communicating reactive processes. In *Proc. 20th ACM Conf. on Principles of Programming Languages, Charleston, Virginia*, 1993.

[Ber93b] G. Berry. The semantics of pure esterel. In *Proc Marktoberdorf Intl. Summer School on Program Design Calculi*, LNCS, to appear. Springer-Verlag, 1993.

[BG91] G. Berry and G. Gonthier. Incremental development of an hdlc entity in Esterel. *Comp. Networks and ISDN Systems*, 22:35–49, 1991.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[BN83] S. Budkowski and E. Najm. Structured finite state automata. a new approach for modelling distributed communication systems. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing and Verification, III*. Elsevier Science Publishers B.V (North-Holland), 1983.

[Bou91] F. Boussinot. Reactive c: An extension of c to program reactive systems. *Software-Practice and Experience*, 21(4):401–428, 1991.

[CPHP85] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre, a declarative language for programming synchronous systems. In *Proceedings of ACM Conference on Principles of Programming Languages*, ACM, 1985.

[dSdS95] Monica Lara de Souza and Robert de Simone. Using po methods for verifying behavioural equivalences. In *Proceedings of FORTE'95*, pages 59–74, October 1995.

[Fer89] Jean-Claude Fernandez. Aldebaran: A tool for verification of communicating processes. Rapport SPECTRE C14, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, September 1989.

[GKPP95] R. Gerth, R. Kuiper, R. Peled, and W. Penczek. A partial order approach to branching time model checking. In *Proceedings of ISTCS*, pages 330–339, 1995.

[God94] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. PhD thesis, UNIVERSITE DE LIEGE, Faculté des Sciences Appliquées, 1994.

[Gue86] P. Le Guernic. Signal, a data-flow oriented language for signal processing. *IEEE Trans. ASSP*, 34(2):362–374, 1986.

[Hal93] N. Halbswachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Press, Netherlands, 1993.

[Hol91] Gerhard Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, N.J., first edition, 1991.

[JLRM] M. Jourdan, F. Lagnier, P. Raymond, and F. Maraninchi. A multiparadigm language for reactive systems.

[Mad92] E. Madelaine. Verification tools from the Concur project. *EATCS Bulletin*, 47, 1992.

[MV89] E. Madelaine and D. Vergamini. Auto: A verification tool for distributed systems using reduction of finite automata networks. In *Proc. FORTE'89 Conference, Vancouver*, 1989.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of CAV'94*, LNCS 818. Springer-Verlag, 1994.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical report, Comput. Sci. Dept., Aarhus Univ., 1981.

[RdS90] V. Roy and R. de Simone. Auto and autograph. In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990. AMS-DIMACS.

[Val90] A. Valmari. A stubborn attack on state explosion. LNCS 531. Springer-Verlag, 1990.

[WG93] P. Wolper and P. Godefroid. Partial order methods for temporal verification. In *Proceedings of Concur'93*, LNCS 715. Springer-Verlag, 1993.