Protocol Verification with Reactive Promela/RSPIN

Elie NAJM*, Frank OLSEN*/**

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

(**) CNET
Centre National d'Etudes des Télécommunications
38-40, rue du Général Leclerc
92131 Issy-Les-Moulineaux
France
E-mail: olsen@issy.cnet.fr

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 demonstrate the language and tool by the specification, translation, simulation and verification of the LAP–B data link protocol. This protocol is quite complex, and benefits from decomposition.

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. Furthermore, 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*.

We illustrate the language and the tool with the specification and validation of LAP–B protocol. We will show that errors in the specification can be detected by using the SPIN tool to perform exhaustive validation.

1.1 Organisation of the paper

The paper starts with an overview of Holzmann's *Promela* language and SPIN tool in section 2. It is followed, in section 3, by an informal description of *Reactive Promela*, our proposed extension to *Promela* and its associated tool RSPIN. In section 4 we show how we have used *Reactive Promela* to specify and verify the LAP–B protocol. We have also included the formal semantics of *Reactive Promela* and of a subset of *Promela* in appendix A. For completeness the full *Reactive Promela* code for the LAP–B protocol is given in appendix B.

2 Overview of Promela/SPIN

In this section we give a brief overview of the *Promela* language and the associated SPIN tool. More information can be found in [Hol91].

Promela is a language used to model communication protocols, and other kinds of distributed systems, at an abstract level. A program in *Promela* consists of processes that communicate either asynchronously over FIFO channels or by binary rendez-vous between processes. The processes are extended Finite State Machines (EFSMs). Only simple data–types are available: integers of various ranges and record types (similar to typedefs in the C language). The reason is that code–generation is outside the scope of *Promela*: it is a pure *validation language*.

Promela does not have a formal semantics, although some authors have given the semantics of a subset of *Promela*: one was in our paper at the TACAS'96 conference [NO96]; another, also presented at TACAS'96, was given by S. Tripakis and C. Courcoubetis in [TC96].

Associated with the language is a tool called SPIN. It allows *Promela* systems to be simulated, either step-by-step or randomly. Furthermore it is possible to perform an exhaustive simulation by generating the complete state-space of the system. Properties like deadlock and livelock can be detected automatically by checking the generated state-space. Detection of these properties rely on special labels (*end–state-*, *acceptance-*, and *progress* labels) that are placed in the *Promela* code by the user. Even more interesting is that other more specific properties can be expressed in linear time temporal logic, and be verified by exhaustive simulation. Unfortunately only one temporal logic formula can be verified at a time which is not convenient. All properties are verified *on–the–fly*, i.e. it is not necessary to generate the whole state space to detect errors, (although to verify that the specification is error–free it is required to generate the whole state–space).

The verification part of SPIN uses several different state–space generation algorithms, including normal depth–first search and Holzmann's *Supertrace* bitspace algorithm. More recent versions of SPIN uses partial–order methods [God94] to perform exhaustive simulation on a reduced state-space. The aim of this is to combat the state-explosion problem which limits the size of systems that can be verified.

3 Overview of Reactive Promela/RSPIN

We now go on to describe our *Reactive Promela*, our extension to *Promela*. It aims at enhancing *Promela* in the following areas:

- **decomposition:** *Promela* gives the user a single level of decomposition: the process level. We add a *reactive process* which can be decomposed into a set of communicating *automata*[1].
- **reactive composition:** *Promela* processes can be composed in parallel via asynchronous message channels or by binary rendez–vous. The automata that make up a reactive process communicate using a *synchronous reactive* style of communication[2]
- **state–space reduction:** the automata that make up a reactive process are combined into a single *Promela* process using the RSPIN tool. If these automata had been modelled as *Promela* processes the SPIN tool would have created the full interleaving of their actions with all other processes on the specification. Our combination algorithm prevents this interleaving by forbidding external inputs during a reaction.
- **atomic reactions:** when a reactive process accepts an input it reacts atomically by changing its state and by generating a set of outputs. This is simpler than using *Promela*'s atomic sequence statement, since the RSPIN tool automatically encapsulates reactions with this construct. Hence, the user does not need to decide in each case whether to use atomic or not.

This section is organised as follows: we first give the concrete syntax for *Reactive Promela* in section 3.1, before we give an informal overview of the language, expanding on the above introduction, in section 3.2.1–3.2.6. Then in section 3.3 we discuss the accompanying RSPIN tool.

3.1 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}*
```

[1]We also consider providing arbitrary levels of decomposition.
[2]Note that neither reactive processes nor *Promela* processes communicate using the synchronous reactive style, only automata do.

```
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.

3.2 The Reactive Promela language

In this section we describe *Reactive Promela* informally. For the interested reader we have included the formal semantics in appendix A.

3.2.1 Reactive Promela specifications

A *Reactive Promela* specification consists of two separate parts:

- a *Reactive Promela* part consisting of a set of *rproctypes.*
- a *Promela* part containing any valid *Promela* code. This is the *pro-active* part of the specification.

3.2.2 Ports and channels

The ports in *Reactive Promela* are *typed* and *directional.* A port is either an *inport* or an *outport.* Ports are declared in the interfaces of rproctypes and automata. As for *Promela* channels, the types of the message-parameters are declared within braces. The ports declared in the rproctype interface are only references to *Promela channels*, declared globally.

3.2.3 Links

The link statement is used to create connections between outports and inports. A *valid link* connects exactly one outport to one or more inports. A link statement contains a list of links (separated by a semicolon). Refering to the rule for `link` in the grammar, we see that each link connects an outport (to the left of the `=>` separator) to one or more inports. The port name is qualified with the name of automaton that it belongs to.

3.2.4 Rproctypes

An rproctype allows a collection of automata to be encapsulated as a single unit. It consists of:

- a set of *synchronously communicating automata*
- definition of *links* between the automata.
- definition of an *interface* (the ports over which the rproctype can communicate with its environment). We can think of this as an *external interface* since it is a subset of interfaces of the automata contained in the rproctype. These ports are only redeclared in the interface—they must be defined as *Promela channels* before they can be used in the rproctype.

3.2.5 Automata

The *Reactive Promela* construct used to specify dynamic behavior is the *automaton*. An automaton definition consists of two parts:

- definition of an *interface* (the ports over which the automaton can communicate with its environment). Contains the name of each inport and outport together with a list of the types of the arguments that each port can take. Ports that are also declared in the rproctype interface are qualified as `external` ports. For these only the name is declared not the list of argument types.
- definition of *dynamic behavior*. Given by a directed graph whose nodes are stable- or transitory states and whose edges are transitions labeled by atomic actions. The actions (currently) allowed are those listed under `abody` in the grammar (see section 3.1).

Stable states are the states in which an automaton is waiting for an input on one of its inports. When it receives a message it goes through a reaction phase before ending up in another (possibly the same) stable state. Each automaton has an *initial state* which is stable.

The participation of an automaton to a reaction starts with the execution of the receive statement. The automaton then executes a series of elementary actions (assignments, conditional statements, send statements, ...). External input actions are not allowed during the reaction. The states separating the actions of the reaction are called *transitory states*. The result of such a behavior is a set of outputs to the automaton's environment, as well as an update of the state.

3.2.6 Reactions

Now let us explain how the behavior of the automata contained in an rproctype make up the behavior of the rproctype itself. The initial state of an rproctype is a *state vector* containing the initial states of each automaton it encapsulates (with all automata in stable states). When there is a message in one of the channels referred to in the rproctype's interface, the rproctype can go through a *reaction phase* which takes it from one stable state to another (possibly the same).

The reaction starts when one of the encapsulated automata executes an input action to consume a message at the head of a channel. In the sequence of actions that follows one possible action is to send a message to another automata within the same rproctype. This causes the reaction to spread to some or all the automata. An important property of reactions is that all the encapsulated automata are involved in it and that, notably, no external events are taken into account until the reaction terminates. Termination of the reaction is when all the automata again find themselves in a stable state.

Seen from the exterior, the rproctype reacts to an input by changing the state of one or more of the automata it encapsulates, by changing the values of variables local to the automata and by producing a set of messages appended to channels in its interface.

3.3 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.

The translation from a reactive process to a *Promela* process is based on an automata combination algorithm which uses breadth–first search to create the state–space of the automata contained.

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. A property of an rproctype is that its reaction to an input from the environment is atomic. This means that many interleavings of actions from the reactive process with actions from other processes need not be calculated during verification. This means that only reactive process actions occuring at stable states are interleaved. In the mapping from *Reactive Promela* to *Promela* we take advantage of *Promela*'s atomic sequence construct (`atomic { ... }`) to provide a way to implement this property. This also simplifies the use of atomic statements: instead of the user selecting manually which part of a specification to encapsulate in an atomic sequence RSPIN does this automatically.

One difficulty in preserving the semantics of *Reactive Promela* in the translation is that *Promela* channels are finite. This means that we cannot be certain that reactions are really atomic. We emphasise that this is not a problem specific to *Reactive Promela*: anyone using the atomic sequence in a *Promela* specification might experience blocking in the middle of a supposedly atomic sequence. One solution is to use SPIN's *"loose messages to full queues"* simulation option. In the RSPIN tool we have also made generation of atomic encapsulation optional to give the user some more flexibility. This option is notably to be used if zero–lenght channels are used to communicate with a reactive process (i.e. rendez–vous communication).

4 Specifying and validating the LAP–B protocol.

In this section we use the specification of the LAP–B protocol[3] to demonstrate the *Reactive Promela* language on a nontrivial example. For more information on LAP–B we refer to most textbooks on communications protocols, e.g. [Sch88]. In section 4.1 we first show how the protocol is modelled as a *Reactive Promela* process through decomposition. Then, in section 4.2 we discuss the translation to *Promela*. Finally, in section 4.3 we give the results from verifying the protocol using a suitable test system.

4.1 Decomposing the LAP–B protocol

To model the LAP–B protocol in *Reactive Promela* we decompose it into the five automata[4], playing the following roles:

1. ***Transmitter:*** receives messages from a higher protocol layer and encapsulates each one in a frame to be sent over a physical medium. It sends the frames to the *Retransmitter* process (which will handle possible retransmissions) and the sequence number to the *Window*, and then waits for new messages.

2. ***Window:*** handles the sequence numbers and the sliding window. Once the *Window* is saturated it notifies the *Transmitter* and the upper protocol layer.

[3] The example is taken from a course in protocol specification given by Elie Najm at ENST.

[4] This description is not complete, notably with respect to the use of timers; for more information we refer to appendix B which is a complete listing of the *Reactive Promela* code for LAP–B.

3. ***Retransmitter:*** responsible for keeping a buffer of the frames sent but not yet acknowledged. When it receives an acknowledgment number from the *Receiver* it purges all acknowledged frames (in case of reject), and demands the *Acknowledger* process to retransmit the other, non-acknowledged frames.

4. ***Receiver:*** receives frames from the physical medium. Depending on the frame type it decides whether the message is a new one which can be exteacted and sent to the upper protocol layer, whether immediate retransmission is required or not, and so on.

5. ***Acknowledger:*** its role is to (re)transmit one or more frames as indicated by the *Transmitter* or the *Retransmitter*.

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 the protocol as a whole. This is because RSPIN translates it into a single *Promela* proctype.

This decomposition arose from an exercise that started with the simplest possible protocol: just a sender and a receiver communicating over a simplex channel. When extending this to handle duplex communication, finally ending up with the LAP–B protocol, we thought that it might be a good idea to reuse the specifications of the sender and the receiver and combine them into one process. However, combining automata manually is not an easy exercise. If we keep the two processes separate we also noticed that there may be a need for some coordination. An example is that the sender and receiver need to coordinate their actions to decide whether an acknowledgement should be retransmitted as a separate message or to be piggybacked. If both the sender and receiver may receive messages at the same time, coordinating their actions may be difficult. By using the synchronous reactive style of communication we avoid this problem altogether by forbidding other external inputs while one inputs is treated[5]

4.2 Translating LAP–B to *Promela*

Once we have modelled the LAP–B protocol in *Reactive Promela* we must use RSPIN translate it to *Promela* before we can perform any simulation and verification. We do not show the complete *Promela* proctype produced, since it is far to big, but we will make a few remarks about it.

We note first that the *Promela* proctype resulting from running RSPIN on the file containing the *Reactive Promela* specification of the LAP–B protocol is quite big. The original *Reactive Promela* specification is contained in a file of approx. 5 kB (approx. 350 lines of code), whereas the resulting *Promela* proctype is contained in a file of about 200 kB (approx. 5000 lines of code). The resulting proctype has 16 global stable states, which is what we would expect. The full crossproduct of the five automata would give 32 states, but the *Transmitter* and the *Window* have a shared state, corresponding to the case of a saturated window. This shows very clearly that decomposition is essential for modelling complex protocols: we have tried to combine automata manually but gave up after combining the three smallest ones.

Internal communication between automata is reduced to assignment to variables. An example (see appendix B) is that the send statement `F!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

[5]Note that in many of the other syncronous languages, e.g. Esterel [BG91] or SL [BdS95], processes react to a collection of inputs. We consider making this possible in *Reactive Promela* as well.

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 external communication actions are kept as is in the combined automaton.

4.3 Simulating and verifying the LAP–B protocol

To check that the specification of the LAP–B presented above is correct we use SPIN to simulate and verify the translated protocol. Note that in this section we only talk about normal *Promela* processes.

We use a standard system to test the protocol: one process models a communication medium which non-deterministically looses, corrupts or transfers correctly a frame; on each side of the link is an instance of the LAP–B process. Each LAP–B instance communicates with a simple user process (which sends a finite number of messages using the protocol), and with two timer processes—one for retransmissions if no acknowledgement arrives in time[6] and one for keeping track of how long to wait before acknowledging a message with a separate frame instead of waiting for a message to piggyback the acknowledgement on.

We can now use SPIN to analyse the protocol. By random or step-by-step simulation we have checked that the test system in the basic cases behaves as expected. However, when we used the exhaustive simulation (verification) mode, we detected an error in *our specification of* the LAP–B protocol. The problem occurs when the *Windows* on both sides of the communication link are saturated. Saturation means that acknowledgements will not be piggybacked. If a frame is received correctly by the saturated side and it sends an acknowledgement which is lost, it will not retransmit the acknowledgement when the other side retransmits the frame. The other side retransmits after a timeout period, but the acknowledgements are not retransmitted (unless they can be piggybacked).

It would of course be interesting to compare the size of the graph created during verification of our system with the graph for a corresponding system where everything is modelled as *Promela* processes. Unfortunately we do not have a machine with enough memory available to achieve this.

In addition to the LAP–B protocol we have also modelled the **Go–Back–N** and the **Selective Reject** protocols. So far we have only simulated them, but we will soon try to verify these protocols as well.

5 Conclusion

In this paper we have presented the *Reactive Promela* language and its associated tool RSPIN through the specification of the data link protocol LAP–B. 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.

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 saw that by translating a *Reactive Promela* specification to a *Promela* specifications using RSPIN, we could simulate and verify it using SPIN.

References

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

[6] Of course, *Promela* doesn't model time—a timeout can only occur if no other action is possible—which makes it difficult to model this accurately. Therefore proposed timed extensions of *Promela* [TC96] are very interesting.

[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.

[NO96] Elie Najm and Frank Olsen. Reactive efsms – reactive promela/rspin. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: second international workshop ; proceedings / TACAS'96, Passau, Germany, March 27–29*, volume 1055 of *Lecture Notes in Computer Science*, pages 349–368, Berlin, 1996. Springer-Verlag.

[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.

[Sch88] Mischa Schwarz. *Telecommunications Networks: Protocols, Modeling and Analysis*. Addison–Wesley Series in Electrical and Computer Engineering. Addison-Wesley, Reading, MA, USA, 1988.

[TC96] Stavris Tripakis and Costas Courcoubetis. Extending promela and spin for real time. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: second international workshop ; proceedings / TACAS'96, Passau, Germany, March 27–29*, volume 1055 of *Lecture Notes in Computer Science*, pages 329–348, Berlin, 1996. Springer-Verlag.

[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.

A Semantics of Reactive Promela

In order to reason about our extension and about the translation from *Reactive Promela* to *Promela* we formalise an essential subset of *Promela* in section A.1, in terms of *Promela State Machines (PSM)*. In section A.2 we then formalise the reactive extension with the *Reactive State Machine (RSM)* model[7].

A.1 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.

A.1.1 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.

A.1.2 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: $(PSM \times \Gamma \times PSM)$[8]. 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:

[7]Note that this section is taken from our paper presented at TACAS'96 [NO96]. It is included here for completeness and in order to resolve any ambiguities in our informal, but more intuitive description in section 3.

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

TEST

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

ASSIGNMENT

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

SEND

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

RECEIVE

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

A.1.3 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[9] (with typical element Q) as 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.

A.1.4 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

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

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

ACTIVATED–ATOMIC–SEQUENCE

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

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).

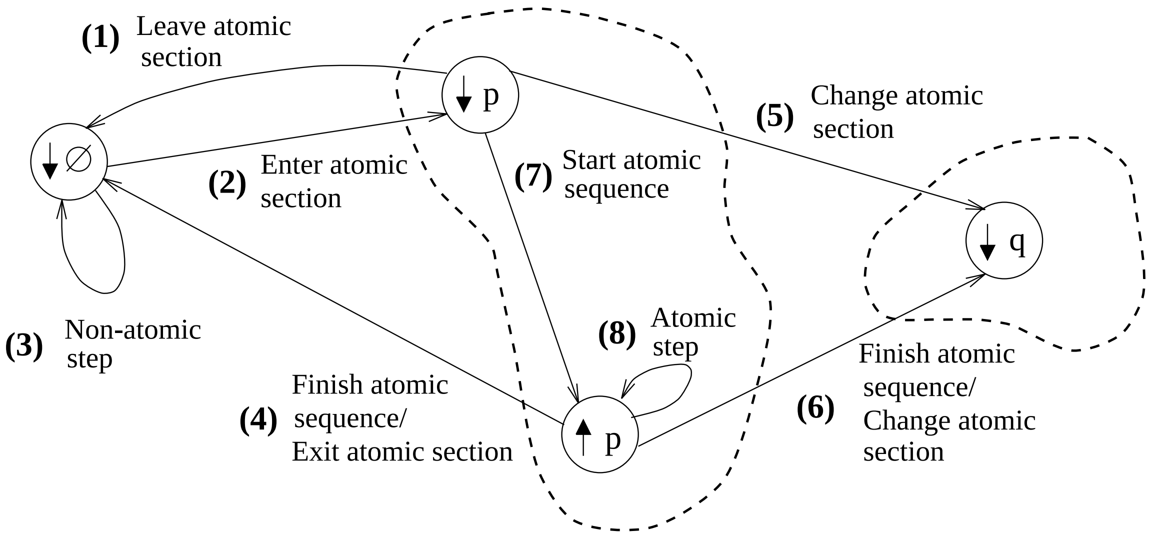
(1) Leave atomic section
↓p
↓∅
(2) Enter atomic section
(5) Change atomic section
(7) Start atomic sequence
↓q
(3) Non-atomic step
(8) Atomic step
(4) Finish atomic sequence/ Exit atomic section
↑p
(6) Finish atomic sequence/ Change atomic section

Figure 1: Allowable atomic transitions.

A.1.5 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.

A.1.6 Semantics of PSM specifications

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

SPEC–ASYNCH–SEND

$$\boxed{\begin{array}{c}\dfrac{Q_i \xrightarrow{c!v} Q_i' \quad \forall j \neq i : Int(Q_j) \quad \#c < l(c) \quad l(c) \neq 0}{(Chans, Procs) \longrightarrow (Chans', Procs')} \\ \text{where } \begin{array}{l} Procs' = (Procs - \{Procs'\}) \cup Q_i' \\ Chans' = Chans - \{c(v_1, \ldots, v_n)\} \cup \{c(v_1, \ldots, v_n, v)\}\end{array}\end{array}}$$

SPEC–ASYNCH–RECEIVE

$$\boxed{\begin{array}{c}\dfrac{Q_i \xrightarrow{c?v} Q_i' \quad \forall j \neq i : Int(Q_j) \quad \#c > 0 \quad head(c) = v}{(Chans, Procs) \longrightarrow (Chans', Procs')} \\ \text{where } \begin{array}{l} Procs' = (Procs - \{Procs'\}) \cup Q_i' \\ Chans' = Chans - \{c(v_1, \ldots, v_n, v)\} \cup \{c(v_1, \ldots, v_n)\}\end{array}\end{array}}$$

SPEC–RENDEZ-VOUS

$$\boxed{\begin{array}{c}\dfrac{Q_i \xrightarrow{c!v} Q_i' \quad Q_j \xrightarrow{c?v} Q_j' \quad \forall k \neq i : Int(Q_k) \quad l(c) = 0}{(Chans, Procs) \longrightarrow (Chans, Procs')} \\ \text{where } Procs' = (Procs - \{Procs'\}) \cup Q_i' \cup Q_j'\end{array}}$$

SPEC–INTERNAL

$$\boxed{\begin{array}{c}\dfrac{Q_i \xrightarrow{\epsilon} Q_i' \quad \forall j \neq i : Int(Q_j)}{(Chans, Procs) \longrightarrow (Chans, Procs')} \\ \text{where } Procs' = (Procs - \{Procs'\}) \cup Q_i'\end{array}}$$

A.2 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 A.1.

A.2.1 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[10]:
  - $SS \subseteq S$ 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.*
- $spred : s \rightarrow \{True, False\}$ is a function to determine if a give state is stable 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*[11] 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 $c \in P_{out}$.

A.2.2 Semantics of RSM automata

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

A.2.3 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 is an *interface*[12] 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:

[10]where $S = SS \cup ST$ and $SS \cap ST = \emptyset$
[11]where $P_{in} \cap P_{out} = \emptyset$.
[12]where $C_{in} \cap C_{out} = \emptyset$

  - 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 $\exists s_i : spred_i(s_i) = False$.

A.2.4 Semantics of RSM processes

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

RSM–PROC–EXT-SEND

$$\frac{a_i \xrightarrow{g!val} a_i' \quad ((a_i, g), (\bullet, c)) \in L_{out} \quad c \in C_{out} \quad l(c) < \#c \quad \#c \neq 0}{(A, I, L) \xrightarrow{c!val} (A', I, L)}$$

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

RSM–PROC–EXT-RECEIVE

$$\frac{a_i \xrightarrow{g?v} a_i' \quad ((\bullet, c), (a_i, g)) \in L_{in} \quad c \in C_{in} \quad l(c) \neq 0 \quad \#c \neq o \quad stable((A, I, L))}{(A, I, L) \xrightarrow{c?v} (A', I, L)}$$

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

RSM–PROC–SYNCH

$$\frac{a_i \xrightarrow{g!val} a_i' \quad a_j \xrightarrow{h?v} a_j' \quad ((a_i, h), (a_j, g)) \in L_{int} \quad \#c = 0}{(A, I, L) \xrightarrow{\epsilon} (A', I, L)}$$

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

$$\boxed{\begin{array}{l} \dfrac{a_i \xrightarrow{\epsilon} a'_i}{(A, I, L) \xrightarrow{\epsilon} (A', I, L)} \\ \boxed{\text{where} \quad \begin{array}{l} A = a_1, \ldots, a_i, \ldots, a_n \\ A' = a_1, \ldots, a'_i, \ldots, a_n \end{array}} \end{array}}$$

A.2.5 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.

A.2.6 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 $Int(R) = stable(R)$.

B Reactive Promela code for LAP–B protocol

Below follows a listing of the *Reactive Promela* specification of the LAP–B data link protocol. We do not show the code generated by RSPIN as it is far to big (approx. 5000 lines, as opposed to approx. 350 for the *Reactive Promela* specification).

```
#define QSIZE          10
#define Xoff           0
#define Xon            1
#define armer          1
#define desarmer       0
#define sonnerie       2
#define N              0
#define U              1
#define I              2
#define RR             3
#define REJ            4
#define cor            5
#define f              2
#define W              4

typedef Frame {
  byte type;
  byte D;
  byte NS;
  byte NR
}

/* for the inports */
chan UE[2]  = [QSIZE] of { int };
chan T1r[2] = [QSIZE] of { byte };
chan T2r[2] = [QSIZE] of { byte };
chan LR[2]  = [QSIZE] of { byte,byte,byte,byte };
/* for the outports */
chan UR[2]  = [QSIZE] of { int };
chan CF[2]  = [QSIZE] of { byte };
chan T1d[2] = [QSIZE] of { byte };
chan T2d[2] = [QSIZE] of { byte };
chan LE[2]  = [QSIZE] of { byte,byte,byte,byte };

rproctype HDLC
  (inport  UE, T1r, T2r, LR;
   outport UR, CF, T1d, T2d, LE)
  (bit no)
{

  automaton Tra
    (external inport UE;
     inport   CF={bool};
     outport  RA={byte,byte,byte,byte},
              F={byte})
    (/* no fpars */)
  {
    byte VS=1, M;
    Frame F;

  tra0:
    if
    :: UE[no]?M ->
       F.type=I;
       F.D=M;
       F.NS=VS;
       VS=(VS+1)%W;
       RA!F.type,F.D,F.NS,F.NR;
       F!VS;
       goto tra0
    :: CF[no]?Xoff ->
       goto tra1
    fi;

tra1:
  CF[no]?Xon ->
  goto tra0
}

automaton Fen
  (external outport CF={bool};
   inport   T={byte}, C={byte})
  (/* no fpars */)
{
  byte VS, VA, va;
  bool x, y;

fen0:
  if
  :: T?VS ->
     x = ((VS-VA > 0 && VS-VA <= f) || \
          (VS-VA+W > 0 && VS-VA+W <= f));
     if
     :: (!x) ->
        CF[no]!Xoff;
        goto fen1
     :: (x) ->
        goto fen0
     fi
  :: C?va ->
     y = ((va-VA>0 && va-VA<=f) || \
          (va-VA+W<=f && va-VA+W>0));
     if
     :: (!y) ->
        goto fen0
     :: (y)  ->
        VA=va;
        goto fen0
     fi
  fi;

fen1:
  C?va ->
  y = ((va-VA>0 && va-VA<=f) || \
       (va-VA+W<=f && va-VA+W>0));
  if
  :: (!y) ->
     goto fen1
  :: (y)  ->
     VA=va;
     CF[no]!Xon;
     goto fen0
  fi
}

automaton Ret
  (external inport  T1r;
   external outport T1d;
   inport   C={byte,byte},
            T={byte,byte,byte,byte};
   outport  A={byte,byte,byte,byte})
  (/* no fpars */)
{
  byte RT, ni, n, nf, i, j, b;
  Frame Buf[f];

ret0:
  if
  :: T?Buf[ni].type,Buf[ni].D, \
       Buf[ni].NS,Buf[ni].NR ->
     n=1;
     T1d[no]!armer;
     goto ret1
  :: C?b,RT ->
     goto ret0
  fi;

ret1:
  if
  :: T?Buf[(ni+n)%f].type,Buf[(ni+n)%f].D, \
       Buf[(ni+n)%f].NS,Buf[(ni+n)%f].NR ->
     nf=(ni+n)%f;
     n++;
     goto ret1
  :: C?b,RT ->
     i=0;
     j=n;
     do
     :: (i == j) ->
        n=j;
        break
     :: (i != j) ->
        if
        :: (Buf[(ni+i)%f].NS == RT) ->
           ni=(ni+i+1)%f;
           n--;
           break
        :: (Buf[(ni+i)%f].NS != RT) ->
           i++;
           n--
        fi
     od;
     if
     :: (n == 0) ->
        T1d[no]!desarmer;
        goto ret0
     :: (n != 0) ->
        if
        :: b == N ->
           goto ret1
```

```
        :: b == U ->
           i=0;
           do
           :: (i != n) ->
              A!Buf[(ni+i)%f].type,Buf[(ni+i)%f].D, \
                Buf[(ni+i)%f].NS,Buf[(ni+i)%f].NR;
              i++
           :: (i == n) ->
               T1d[no]!armer;
              goto ret1
           od
        fi
     fi
  :: T1r[no]?sonnerie ->
     i=0;
     do
     :: (i != n) ->
        A!Buf[(ni+i)%f].type,Buf[(ni+i)%f].D, \
          Buf[(ni+i)%f].NS,Buf[(ni+i)%f].NR;
        i++
     :: (i == n) ->
        T1d[no]!armer;
        goto ret1
     od
  fi
}

automaton Rec
  (external inport  LR;
   external outport UR;
   outport  A={byte,byte},
            R={byte,byte},
            F={byte})
  (/* no fpars */)
{
  Frame F;
  byte VR=1, i;
  bool x;

rec0:
  LR[no]?F.type,F.D,F.NS,F.NR ->
  if
  :: (F.type == cor) ->
     goto rec0
  :: (F.type != cor) ->
     F!F.NR;
     if
     :: (F.type == RR) ->
        R!N,F.NR;
        goto rec0
     :: (F.type == REJ) ->
        R!U,F.NR;
        goto rec0
     :: (F.type == I) ->
        R!N,F.NR;
        x = ((F.NS-VR > 0 && F.NS-VR < f) || \
             (F.NS-VR+W < f && F.NS-VR+W > 0));
        if
        :: (!x && F.NS != VR) ->
           i++;
           if
           :: (i != f) ->
```

```
               goto rec0
            :: (i == f) ->
               A!U,(VR+W-1)%W;
               i=0;
               goto rec0
            fi
         :: (F.NS == VR) ->
            UR[no]!F.D;
            A!N,VR;
            VR=(VR+1)%W;
            i=0;
            goto rec0
         :: (x) ->
            A!U,VR;
            goto rec1
         fi
      fi
  fi;

rec1:
  LR[no]?F.type,F.D,F.NS,F.NR ->
    if
    :: (F.type == cor) ->
       goto rec1
    :: (F.type != cor) ->
       F!F.NR;
       if
       :: (F.type == RR) ->
          R!N,F.NR;
          goto rec1
       :: (F.type == REJ) ->
          R!U,F.NR;
          goto rec1
       :: (F.type == I) ->
          R!N,F.NR;
          if
          :: (F.NS != VR) ->
             goto rec1
          :: (F.NS == VR) ->
             UR[no]!F.D;
             A!N,VR;
             VR=(VR+1)%W;
             i=0;
             goto rec0
          fi
       fi
    fi
}

automaton Ack
  (external inport  T2r;
   external outport T2d, LE;
   inport   C={byte,byte},
            T={byte,byte,byte,byte},
            R={byte,byte,byte,byte})
  (/* no fpars */)
{
  Frame F;
  byte b,NA;

ack0:
  if
  :: T?F.type,F.D,F.NS,F.NR ->
     F.NR=NA;
     LE[no]!F.type,F.D,F.NS,F.NR;
     goto ack0
  :: R?F.type,F.D,F.NS,F.NR ->
     F.NR=NA;
     LE[no]!F.type,F.D,F.NS,F.NR;
     goto ack0
  :: C?b,NA ->
     if
     :: (b == N) ->
        T2d[no]!armer;
        goto ack1
     :: (b == U) ->
        F.type=REJ;
        F.NR=NA;
        LE[no]!F.type,F.D,F.NS,F.NR;
        goto ack0
     fi
  fi;

ack1:
  if
  :: T?F.type,F.D,F.NS,F.NR ->
     T2d[no]!desarmer;
     F.NR=NA;
     LE[no]!F.type,F.D,F.NS,F.NR;
     goto ack0
  :: R?F.type,F.D,F.NS,F.NR ->
     T2d[no]!desarmer;
     F.NR=NA;
     LE[no]!F.type,F.D,F.NS,F.NR;
     goto ack0
  :: C?b,NA ->
     if
     :: (b == N) ->
        goto ack1
     :: (b == U) ->
        T2d[no]!desarmer;
        F.type=REJ;
        F.NR=NA;
        LE[no]!F.type,F.D,F.NS,F.NR;
        goto ack0
     fi
  :: T2r[no]?sonnerie ->
     F.type=RR;
     F.NR=NA;
     LE[no]!F.type,F.D,F.NS,F.NR;
     goto ack0
  fi
}

link {
  A  in Ret => R  in Ack;
  F  in Tra => T  in Fen;
  RA in Tra => T  in Ret, T in Ack;
  CF in Fen => CF in Tra;
  F  in Rec => C  in Fen;
  R  in Rec => C  in Ret;
  A  in Rec => C  in Ack
}
}
```