Design Validation of *ZCSP* with SPIN

Differents Models and Properties of a Zero-Copy Secured Protocol

Vincent BEAUDENON, Emmanuelle ENCRENAZ, Jean-Lou DESBARBIEUX
UPMC - LIP6 - ASIM
12, rue Cuvier
75005 PARIS - FRANCE
{Vincent.Beaudenon, Emmanuelle.Encrenaz, Jean-Lou.Desbarbieux}@lip6.fr

ABSTRACT

We consider the problem of specifying a model of a Zero Copy Secured Protocol for LTL verification purpose with the Model Checker SPIN. ZCSP is based on Direct Memory Access. Datas are directly read/written in user space memory, decreasing latency and saving processor computing time. We first introduce the ZCSP protocol before analysing differents ways of modelling.

Two main steps were performed : A finite and a none finite sequences model. The first model gave us an overview of the protocol robustness. The second allowed us to test realistics properties.

Unfortunately the size of the system is frequently prohibitive. Thus, we explain all minimization we had to perform. We also describe LTL properties that were checked with the SPIN model checker.

Keywords

Model Checking, SPIN, ProMeLa, ZCSP, LTL

1. INTRODUCTION

In this paper we consider the problem of specifying the model of ZCSP protocol[1] for LTL verification purpose with the model checker SPIN. Readers can find brief description of Spin Model Checker in [4], and of protocols validation using this tool in [2]. SPIN is a generic verification system that supports the design and verification of asynchronous process systems. Process interactions can be specified in rendez-vous primitives , with asynchronous messages passing through typed buffered channels, through access shared variables or any combination of these [4]. SPIN accepts design specifications written in the verification language ProMeLa (a Process Meta Language), and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL)[6].

To perform verification, SPIN takes a correcteness claim that is specified as a temporal logic formula, converts that formula into a Büchi automaton, and computes the *synchronous* product of this claim and the automaton representing the global state space. The result is again a Büchi automaton. A Büchi automaton *accepts* a system execution if and only if that execution forces it to pass infinitely often through one ore more of its accepting states. We call such behaviors *acceptance cycles*. SPIN performs the translations from LTL to Büchi automaton with a tableau-based algorithm geared towards being used for an "on-the-fly" fashion model checking [7]. Then, after specifying a desired behavior in LTL formula, Spin search a counter-example related to an acceptance cycle in the transition-based synchronous product of the system and the Büchi automata correspondig to the negated formula.

We aim to prove some temporal properties of a Zero-Copy Secured Protocol, and describe our modelization experience. In a first part we will describe the Zero-Copy Secured Protocol, then we will expose two main kinds of models, explain all choices and results. Finally we will extract general modelization rules, property assumptions and resource difficulties that will induce future works in formal proof.

2. ZERO-COPY SECURED PROTOCOL

2.1 Utility

Clusters of PCs are a very attracting solution for low cost parallel computing. In these machines, the performances are strongly dependant on the communication protocols. A basic idea to improve network performance is to avoid multiple copies of data. In classical communication protocols such as TCP/IP, the transmitted data are copied several times from memory to memory on both the source node processor and the target node processor. With gigabit/s networks, those multiple copies are not anymore negligible in terms of latency and introduce processor overhead. Zero copy communication protocols are based on direct memory access (DMA) : the sender node's network interface controller (NIC) reads the data directly from user space memory and the receiver node's NIC writes data directly in user space memory, without processor's intervention or copying any byte of data. The zero copy behavior decreases latency and saves processor computing time, hence enabling more computing while communication occurs.

2.2 Behavior

2.2.1 Principle

A packet is the smallest piece of data transmitted atomically through the network. Each packet contains the target node number, the physical address in the remote memory, and the message identifier. In most networks, a corrupted packet is simply discarded by the network. So we make the assumption that any transmission error will simply result in one or several packet losses.

On the sending node, the NIC starts the timer when the last packet of a message is sent. It can then send others messages or acknowledges to the same or other destinations. If the NIC receives the acknowledgement packet, the sending process is notified and the timer is stopped. If the acknowledgement packet is not received after the given time, the timer fires a timeout signal, and the sender node goes into error recovery mode.

2.2.2 *Error Recovery*

In error recovery mode, the NIC tries to send again the whole message, as a "bis" message, and it will not send any other message until the expected acknowledge packet is received. For each transmitted packet to a given receiver node, the sender NIC adds a sequence number. The receiver NIC checks the sequence number and writes the packet into memory only when the packet has the expected sequence number.

If the last packet of a message is accepted and written to memory, then the whole message has been written to memory. Hence the acknowledgement packet can be sent back to the sender, and the message may be signaled to the software on the receiving node. All arriving packets with a wrong sequence number are discarded and the acknowledgement packet is not generated. In these last cases, the sender will timeout and will re send the whole message as a"bis" message. On retransmission of a "bis" message, the sender reinitialize its sequence numbers for transmission, so that the message is transmitted with the same sequence numbers as the first time. Packets that were already received and written into memory will be discarded by the receiver, because they do not have the expected sequence number.

2.3 Examples

We consider the sending of two messages. The first message (A) contains two packets. The second message (B) contains three packets. The current sequence number is 4. The receiver is awaiting packet 4 then 5 etc. Packet 5 is the last packet of message A, and packet 6 is the first of B. In the left-hand side of figure 1 we can see a correct transmission of these two messages. Arrows represents packets transmitted, and in the right column (receiver) we can see *expected* packets. The sender may send B *before* receiving the Acknowledgement of A. We can see this type of transmission in the center of figure 1. If the packet 5 is lost, the receiver will discard all other packets. The Sender will time out and send the whole message A in "bis" mode. After packet "A#4", receiver expects "A#5" packet even if message B is perfectly transmitted. Thus, B will have to be re-sent.

This mechanism also handles the loss of an acknowledgement packet. Each acknowledgement is bounded to a unique message. If network contention causes the acknowledge packet to reach the sender too late, or if the acknowledge packet is lost, the sender node will timeout and send a "bis" message. The receiver will simply drop all packets of the "bis" message, and send again an acknowledge packet when receiving the last data packet. We can see this type of transmission in the right-hand side of the figure 1. The "bis mode" transmission of A is only used to force the receiver to re-emit an acknowledgement. We therefore have to introduce another counter, maintained on each node for each peer node: the Expected Recovery Sequence Number that is used only for "bis" messages.

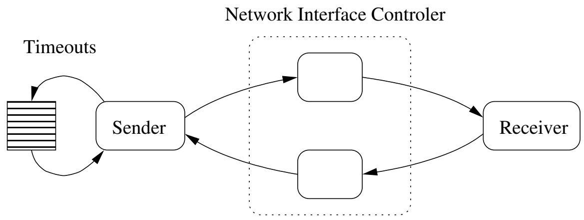

Figure 2: Global Scheme of the finite model

3. THE FINITE SEQUENCES MODEL

The first question we wanted to ask is "For a given set of messages, are we sure that all desired acknowledgements will arrive if the network is not broken ?". This is the first step of our global work, it allows main modelization tests before implementing a model close to the real protocol.

3.1 Specifying the system

3.1.1 *Datas*

The set of messages is described in a table. Each place of this table is bounded to a unique element in the finite set of messages. Each message has a given set of packets and a boolean variable indicating if the message was correctly acknowledged. Each packet contains these informations :

- Its sequence number : `Sqn`.
- The number of the related message : `Mess`.
- A bit meaning if it is a "bis" packet : `Bis`.
- A bit meaning if it is the last packet of the message : `Last`.

An Acknowledgement contains the number of the related message. The sender uses a table wich contains, the state of each messages (not sended, sended and acknowledged).

3.1.2 *Specification Scheme*

We have three differents components :

- the Sender,
- the Receiver and
- the Network.

To model the timeouts' signal order, we record them in a FIFO named `Timeouts`. The global scheme can be resume as in figure 2. The Network Interface Controler is described in section 3.1.3.

3.1.3 *Communications*

Each communication channel is split into three parts : a buffer with a given size, a process wich reads in this buffer and writes in a channel for a rendez-vous communication. In each of these processes, we can insert flags which can inform the LTL property manager about the quality of all communications as we will see in section 3.3.1.

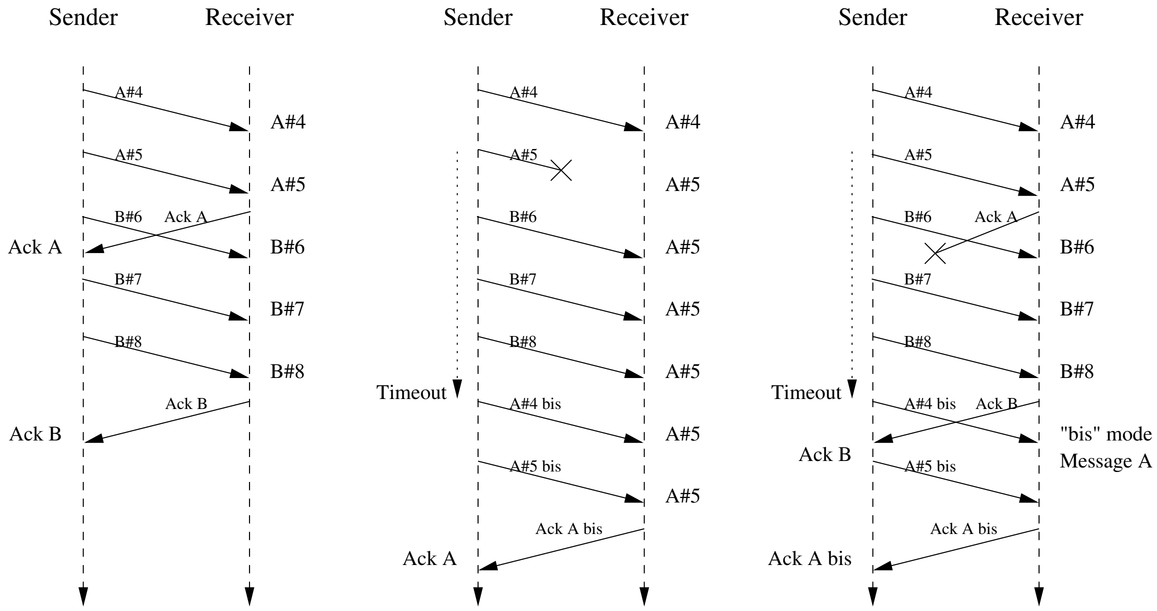

Figure 1: Three executions of ZCSP

3.2 Processes

3.2.1 Receiver

The receiver's behavior is described in figure 3. We can see a difference between this model and the real implementation in the case of "bis messages". In the real system, all values are limited. Thus, to avoid any confusion between two different messages, an acknowledgment is discarded if it doesn't have the same re-emission number of the bounded message. In our model, we suppose that this methods works and the receiver re-send an acknowledgement iff it receieves the last packet of a "bis" message allready acknowledged. This choice was made under the assumption that receiver always recognize the real nature of any message.

3.2.2 Sender

The sender's behavior is described in figure 4. The most important part in this model is select state. The choice between each branch means a different *scenario* and doesn't perform any priority of execution.

- the "ack" guard means that an acknowledgement may be read.
- the "mess" guard means that new message may be sent respecting the size of emission window.
- the "timeout" guard means that a timeout may be extracted from the FIFO. This may happen even if no message is waiting acknowledgement. The SOS step reads the "timeout" and the "valid" test verifies that it corresponds to a not allready acknowledged message. In such a case, the process enter in "bis" mode (bottom-right part in the scheme).

When sending message in "bis mode",timeouts on others messages are not considered (The process doesn't read the FIFO), but only acknowledgements. This means that no timeout may arrive *before* the corresponding acknowledgement. This will be corrected in section 4.

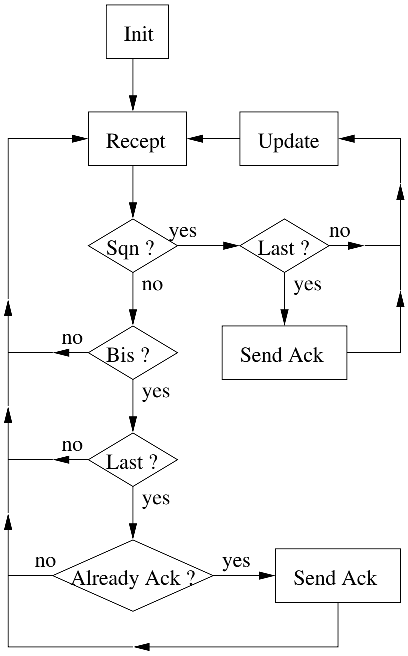

Figure 3: the finite sequences receiver's model

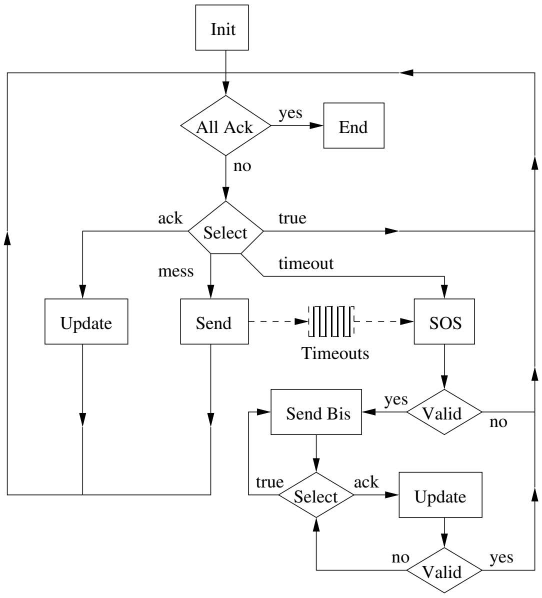

Figure 4: the finite sequences sender's model

3.3 Validation

3.3.1 Assumptions

The SPIN model-checker can construct all possible executions of the specified system. Obviously, there is a class of scenarii that performs unwanted behaviors. There is a set of trivial counter-examples (for example, considering that each packet or each acknowledgement is loss) wich doesn't perform any realistic execution. Then we must construct assumptions related to a good behavior of the network. We introduce a new property on messages' progression. We are sure that receiver will progress if it receives an incremental suite of sequence number from the first to the last packet. For a set of four sequence numbers like $\{0, 1, 2, 3\}$, the suite $0, 1, 2, 3$ obviously works, but it works too with $0, 1, 1, 3, 2, 3$, but the second occurence of "1" and the first occurence of "3" will be discarded. For this set of sequence number, any suite works if it can be written like this :

$$S_0, 0, S_1, 1, S_2, 2, S_3, 3, S_4$$

where each S_i is a finite suite of $\{0, 1, 2, 3\}^*$. Let `ExpectedN` the expected sequential number. For each packet `p`, if `p.Sqn = ExpectedN` then `ExpectedN <- ExpectedN+1`. From the beginning, `ExpectedN = 0` when `ExpectedN = N` we are sure that `p.Sqn` made a progressing suite from `0` to `N-1`. We introduce an analog value for acknowledgements and then, we have a good sight on system's progression. These progression flags are implemented in the network processes. In this section we just assume that these variables reaches an expected value. In this finite sequence model, the progression of emissions induces fairness between processes.

3.3.2 LTL formula

The LTL formula that has been tested for this system is $progress \rightarrow \mathbf{F}(AllAck)$. The Büchi automaton for this first formula is showed in figure 5. In this figure, p means "sending progression" as described in 3.3.1 is verified (`ExpectedN=N`, e means that "End State" is reached . Each state of the synchronised product may be linked to a unique state of the Büchi automaton.

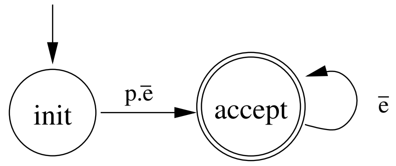

Figure 5: Inverted Büchi automaton for $Progress \rightarrow \mathbf{F}(AllAck)$

3.3.3 Results

The Spin Model Checker doesn't find any counter-exemple, thus, the property is true.

But, as we can see in figure 4, the sender has a state for the begining and a state for the end of any execution that respects assumptions in section 3.3.1. The validation tool checked about 20 millions states. This very small value is due to the progress assumption. And twenty minutes were sufficient to reach the end of the validation. Save the correct transmission of a finite set of messages and a first impression on ZCSP's functionning priciple, no conclusion on the protocol behavior can be made. In section 4 we will see how to deal with a none finite system.

4. THE INFINITE SEQUENCES MODEL

4.1 A more realistic model

In this section we will avoid any ending state. We have to model an automaton that constructs all possible scenarii of infinite sequences. In the real implementation, each variable has a finite domain. But we can't use the same ecoding for our model : The size of the set of states would be prohibitive. Considering the finite domain of each variable, each sequence is *ω-regular* (a finite prefix followed by a cyclic regular expression). In section 4.3 we will see how to avoid any conflict between to "different" scenarii.

4.2 Specification Scheme

4.2.1 The System

The global scheme of the system is shown in figure 6. We use a message-entries table and variables that may be directly linked to the real implementation.

- Eld_P : Index pointing the elder awaiting acknowledgement (or timeout) message.
- FF : Index pointing the "First Free" box in the table.
- FT : Index pointing the index of the first timeout received. This informations allow us to take care of timeouts received when sending a "bis message", in these cases the boolean "ToBis" is set to True.

All question marks on the figure represents non-deterministic behaviors. The "network processes" in section 3 are directly implemented in the Sender and Receiver. We just replace the action of sending with a two-guards select. Each branch may always be visited. One of them receives/sends a packet. The other throwit down. There is no difference for the global

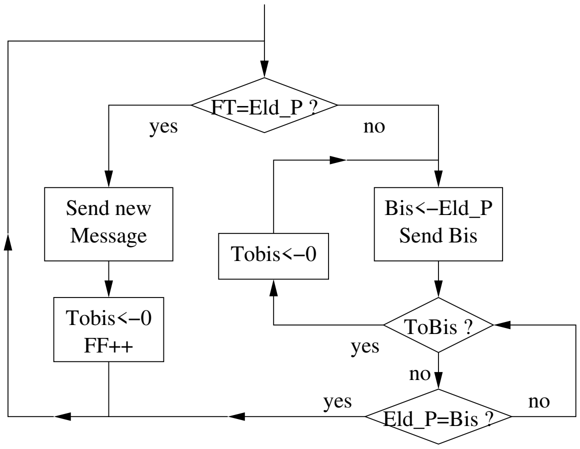

Figure 7: The infinite sequences model's Sender

behavior but it reduce unexpected interleaving. The necessity of this kind of reduction shows the limitations of Partial Order Methods [5, 8] as we will see in section 4.4.3.

The non-deterministic behavior of the "update" process is more decisive as we will see in section 4.2.4. In the figure 6 we can see a reachable state of the message-entries'table : The "first free" column is empty. There are three messages awaiting acknowledgement (in order 3,0,1), remember that messages 0 was sended *after*message 3. A timeout occured for message 0 (FT = 0) during the first re-emission of message 3 (R[3]=1) , and message 1 was acknowledged (Ack[1]=1).

4.2.2 Sender

Despite of the possibility to throw down a packet, this process has now a deterministic behavior. All choices are made on global variables. the sender's automaton is described in figure 7.

Any message emission (in "Send new Message" and "Send Bis") is performed in an atomic sequence.

4.2.3 Receiver

This new process uses reemission numbers and is described in figure 8. State "AckR" has no incidence on this system. In state "Recept" the receiver reads a unique packet from the Sender in the "Packets" channel. In State "AckE" the receiver emits an acknowledgment (or throw it down) to the update process.

4.2.4 Update

The most specific process of the entire system. Its behavior combined with the Sender's automaton modelso the real implementation. The Sender reacts on any scenario constructed by Update. Update process is described in figure 9.

The unique significant "Select" in the system resides in this process. It constructs all scenarii that we can encounter, by creating ordered timeouts signals or readings acknowledgements. A "Good Acknowledgement" contains the same re-emission number as in message-entries table. In other case "Good Ack" guard is not considered. Precisely, a "bad" acknowledgement is simply discarded and the process returns to select state. If $ToBis = 1$, $FF = FT$ and $FT \neq Eld\_P$, there is a cycle that may accept a lot of unexpected

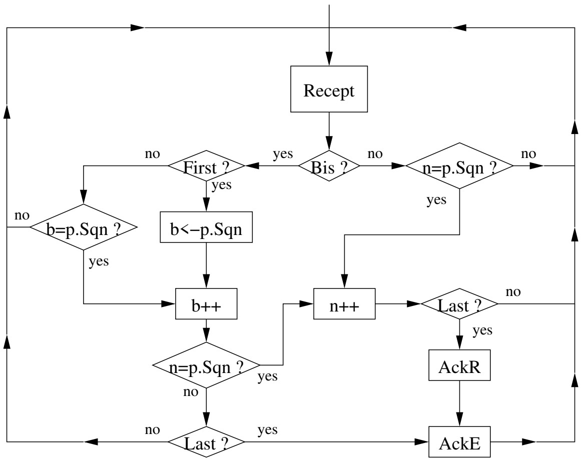

Figure 8: The infinite sequences model's Receiver

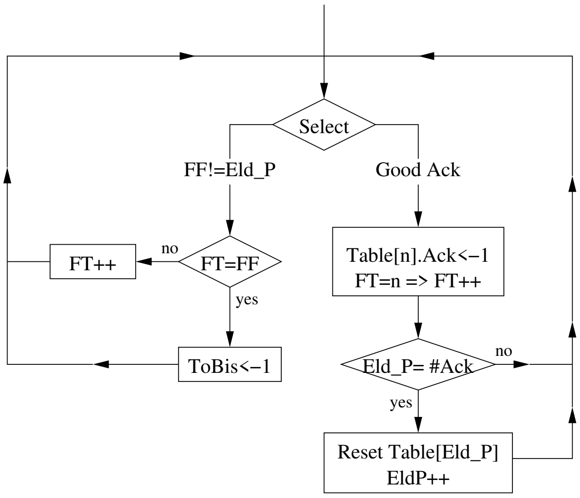

Figure 9: The infinite sequences model's Update process

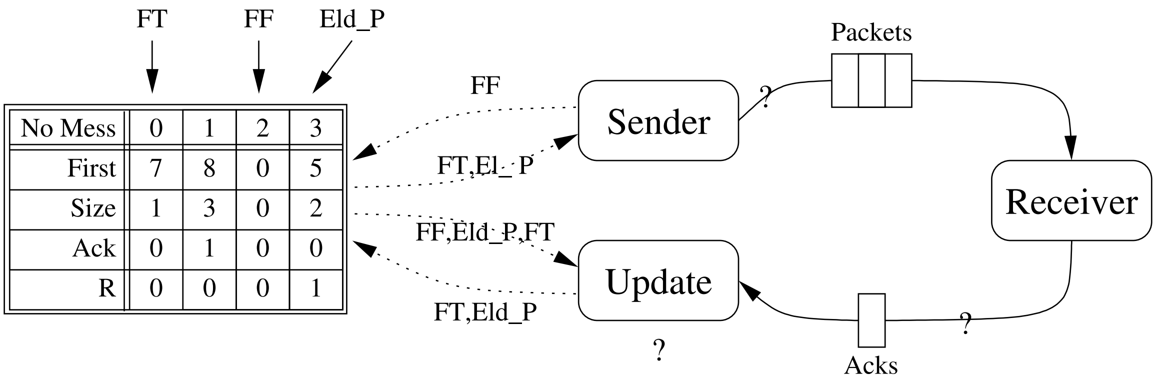

Figure 6: The infinite sequences model's global scheme

properties. We will see in section 4.4.1 how to avoid this kind of phenomenons.

4.3 System's minimization

4.3.1 Table

We reduce the table size three entries. There are two places for awaiting acknowledgement messages and an empty place like in the real model. Thus, Eld_P, FF and FT are defined on $\mathbb{Z}/3\mathbb{Z}$.

4.3.2 Messages

There are three kinds of packets : The first packet of the message, the last, and an intermediary one. Thus there are three kind of messages :

- One-packet messages, this packet is both the first and the last packet.
- Two-packets messages, the first and the last packet.
- Three-packets messages, the first, the last and another packet between them.

4.3.3 Packets

There are two awaiting messages, each of them may contain three packets. Therefore, at least six differents sequential numbers must be used. As in message-entries table, where "FF" leads to a non-existent message, we wished to separate the first sequential number of a new message from the first of the last acknowledged message. In the worst case (two messages of three packets), six differents sequential numbers wouldn't be sufficient. Thus, all Sequence Numbers are defined on $\mathbb{Z}/(2 \times 3 + 1)\mathbb{Z}$.

4.3.4 Channels

The channel between the sender and the receiver is buffered with a size of three packets, to allow atomic emission of a three-packets message. The channel between the sender and the update process is buffered with a size of one acknowledgement to avoid any rendez-vous communication.

4.4 Validation

4.4.1 Fairness : Assumptions or injonctions ?

There are two problems related to exhaustive search on the complete scenarii set.

The first difficulty is to avoid a non-interesting scenario : When a data is systematicaly throw-down during transmission. In section 3 we saw how to build a valid suite of sequence numbers or acknowledgements. But in this non-finite model, we only must consider the progression of the variable "ExpectedN" and the reemission number may reach values that induce a prohibitive size of the search domain. We introduced a ceil of reemission, because scenarii that performs more than one "bis emission" for a given message are not significant. Thus we can force a good emission when this ceil is reached.

The other difficulty is due to potential cycles in processes. As we saw in section 4.2.4, if update is the only working process, a cycle is found but it doesn't represent any real behavior of ZCSP. But if we forces the system to let others processes to work infinitely often, this cycle will be simply avoided.

We saw here two fairness problems. The first one may be solved in two ways :

- Forcing good emmission when a ceil is reached.
- Ignoring emissions that overtakes this ceil.

In very smalls systems, we saw an increasing of the search size by 20% when using the second way; and this value may be greater in taller problems. One the one hand, a fairness assumption in LTL formulae causes new states in Büchi automaton. On the other hand, even if a sequence is not considered as an *acceptance cycle*, all its states must be checked and this induces useless operations. Consequently, fairness must be expressed in expected behavior only if it can't be expressed in the model.

For the other problem, we impose the Sender to work infinitely often (This is the only process that can force Update to get out of the cycle). This property must be expressed independently from other processes' behaviors. Thus we had to add a fairness property (f) in our LTL formula.

4.4.2 LTL properties and generated Büchi automatas

We propose to solve two questions :

- Is it possible to send new messages infinitely often (Liveness property) ?
- If no new message is sended, will the table inevitably become empty (Ending property)?

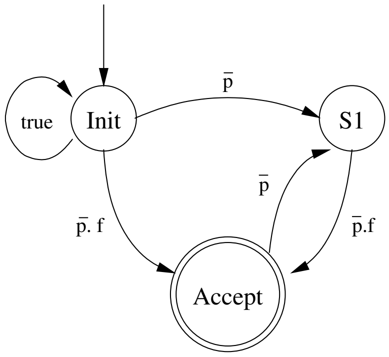

Figure 10: Inverted Büchi automaton for Liveness property

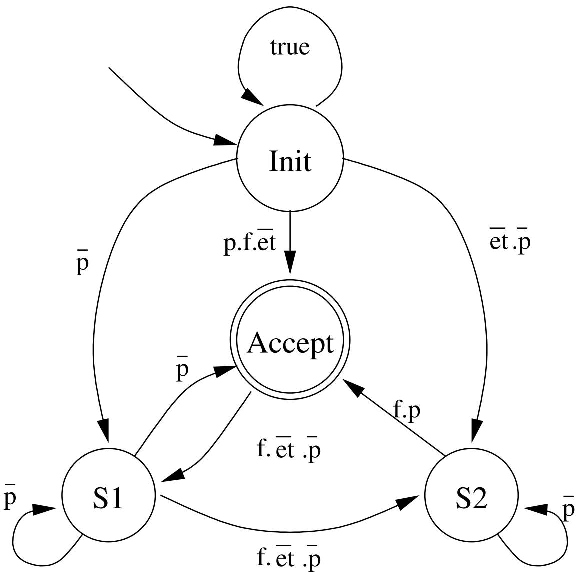

Figure 11: Inverted Büchi automaton for Ending property

The first question is translated in the LTL formula

$$\mathbf{GF}(f) \rightarrow \mathbf{GF}(p)$$

where f means "fairness between processes" and p means "sending progress". We already use the same notations as in section 3 to construct the related Büchi automaton in fig10.

The second question is translated in the LTL formula

$$(\mathbf{FG}not(p)\,and\,\mathbf{GF}(f)) \rightarrow \mathbf{FG}(et)$$

where et means "empty table".

According to the size of this automaton, the reader may be surprised to know that the set of transitions find in the validation search is increased by "only" 83%. But as we can see, there is only one condition to get out of the unique accepting state in the Büchi automata (and eventualy return in later). And this condition is very strict, all values that are used in the LTL formula are implicated. Thus, a great majority of accepting states in the synchronous product are no longer considered.

Property	Liveness	Liveness	Ending
PO Reduction	**no**	**yes**	yes
Transitions(10^6)	1928	455	835
Depth Reached	160265	68948	68948
Memory Needed (Mo)	39521	18392	27589
Memory Used (Mo)	**115**	232	243
Compression %	0,29%	1,26%	0,88%
CPU time	35h09	**12h55**	19h25
Validation Result	**Valid**	**Valid**	**Valid**

Table 1: Stats on Spin validations

4.4.3 Results

The most recommanded option in Spin is "partial order reduction" . The reader may find informations in [8, 5]. We can show the efficiency of these methods with experimental results, in our problem the number of states stored and matched was divided by four ! This option reduces interleaving but as we could see in section 4.2 this is not sufficient and, for example, replacing network processes by two selects in the sender and the receiver allowed us to reach the end of validation by removing useless internal variables.

Bitstate Hashing option can be used to find an error but, in case of "valid" simulation, the results can't be exploited.

To reduce memory requirements, we have used a state compression proposed by Spin. Spin performs Recursive Indexing, run-time penalty is 280% but compared with no compression method, memory needed is decreased to 18,3% only. The default compression mode in Spin is Byte Masking (188% for run-time and 83,6% for memory). Reader may find more informations in [3].

Main results are shown in table 1. All values are brought by the SPIN model checker after validation. In all these cases, we used system's minimizations described in section 4.3. If one of these is discarded, for example, setting the size of the table to four, or increasing the size of a FIFO, the validation tool leads to an "out of memory".

Reader may note some interesting values, comparing performances with and without Partial Order Reduction. In each cases needed memory were prohibitive, thus we used state compression but its interactions with other options leads to differents results. Partial Order Reduction and compression are two recommended options to *reduce* memory usage. The first, decreases the set of matched states. The second decreases the physical memory to store it. But we can see that memory used without Partial Order Reduction is the smallest. The set of matched states is greater but the compression ratio is most impressive. CPU times' line shows the importance of combining these two options, but time processing is not a crippling parameter when you must run your computer during the night... Thus model checkers'users must have to deal with main options separately, according to the machine's performances (CPU and memory), particularly when memory used is too close (or greater than) of available memory.

5. CONCLUSION

We saw that differents properties may induce differents verification models and assumptions. We have encountered the main difficulty of modelling : Finding the minimal sufficient system may be harder than specifying the *real* system itself ! Futhermore, the temporal logic formula may be reduced by translating fairness properties directly in the model. Fortunatly, one of them may be expressed as a realistic property : The end of the process after too many network errors. A direct translation from the real implementation would have lead to a prohibitive memory cost, and we needed a sharp knowledge about the protocol to model it in ProMeLa. Thus, we have to help model-checkers' users to build available models for verification purpose or we must find other ways to express and check temporal properties.

6. REFERENCES

[1] A. Greiner, E. Dreyfus, J.L. Desbarbieux, F. Wajsburt. A Secured protocol for high speed interconnexion networks providing a remote DMA communication primitive. *Internal report of LIP6.*

[2] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[3] G. Holzmann. State Compression in Spin. Twente University, The Netherlands, April 1997.

[4] G. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[5] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. Sixth Int l Conf. Computer Aided Verification (CAV94)*, pages 377–390, 1994.

[6] A. Pnueli. The Temporal Logic of Programs. In *18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[7] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV95)*, pages 3–18. Chapmann & Hall, 1995.

[8] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *CONCUR'93*, August 1993.