# Design Validation of ZCSP with SPIN

Vincent BEAUDENON, Emmanuelle ENCRENAZ, Jean-Lou DESBARBIEUX
UPMC - LIP6 - ASIM
12, rue Cuvier,
75252 PARIS Cedex 05 - FRANCE
email: {Vincent.Beaudenon, Emmanuelle.Encrenaz, Jean-Lou.Desbarbieux}@lip6.fr

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

**Two main steps were performed: A finite and a non-finite sequences model. The first model gave us an overview of the protocol robustness. The second allowed us to test realistic properties. We also describe LTL properties that were checked with the SPIN model checker.**

**Unfortunately the size of the system was frequently prohibitive. Thus, we explain all minimization steps we had to perform: Variables' domains restriction, interleaving reduction, realistic environnement representation by fairness constraints.**

## I. Introduction

In this paper we consider the problem of specifying the model of ZCSP protocol [1] in order to apply LTL verification with the SPIN model checker. Readers can find a brief description of SPIN in [2], and of protocol validation using this tool in [3]. SPIN is a generic verification system that supports the design and verification of systems represented as a collection of asynchronous processes. Interaction between processes can be specified with rendezvous primitives, with asynchronous messages passing through typed buffered channels, through shared variables or any combination of these [2]. 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) [4].

To perform verification, SPIN takes a correcteness claim that is specified as a temporal logic formula, converts this 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 the automaton to pass infinitely often through one or 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 "on-the-fly" model checking [5]. Once a desired behavior has been specified in LTL, SPIN searchs for a counterexample related to an acceptance cycle in the transition-based synchronous product of the system and the Büchi automaton corresponding to the negated formula.

We take aim at proving some temporal properties of a Zero-Copy Secured Protocol, and describe our modeling experience. The paper is organized as follows: In the first part we shall describe the Zero-Copy Secured Protocol, then we shall expose two main kinds of models, explain all choices and results. Finally, we propose modeling rules and property assumptions, and we will discuss on resource difficulties that we have encountered.

## II. Zero-Copy Secured Protocol

### A. Utility

Clusters of PCs are a very attracting solution for low cost parallel computing. In these machines, the performance strongly depends 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 is copied several times from memory to memory on both the source processor node and the target processor node. With gigabit/s (and faster) 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: 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 the processor's intervention or copying any byte of data.

The zero copy behavior decreases latency and saves processor computing time, thus, enabling more computing while communication occurs.

### B. Behavior

The messages are the communication units between processes of an application. At our level each message is split into packets which are the elementary transfer units between an emitter and a receiver. The good reception of a message by a receiver is signaled to the emitter by sending back a special packet named acknowledgement.

*1) Principle:* A packet is the smallest piece of data transmitted atomically through the network. In addition, 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. The last packet of a message has to be acknowledged. The packets must be received in the sequential order they were sent while the message acknowledgements may be received in another order. On the sending node, the NIC starts a timer when the last packet of a message is sent. It can then transmit other messages or acknowledgements to the same or other destinations. When 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) Error Recovery:* In error recovery mode, the NIC tries to send the whole message again, as a “bis” message, and it will not send any further message until the expected acknowledgement 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 this last case, the sender will timeout and will re-send the whole message as a “bis” message. On retransmission of a “bis” message, the sender reinitializes its sequence number 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.

### C. Examples

An example is given in Figure 1. We consider the sending of two messages between an emitter and a receiver. 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”, the receiver expects “A#5” packet even if message B was perfectly transmitted. Thus, B will have to be re-sent later.

This mechanism also handles the loss of an acknowledgement packet. Each acknowledgement is bound 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 acknowledgement packet when receiving the last data packet. We can see this type of transmission in the right-hand side of 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.

### D. Remark

Reader may find similarities between ZCSP and Sliding Window protocol [6]. The main difference resides in the order of acknowledgements. In the right hand of Figure 1 we can see that message “B” is acknowledged *before* message “A”, while in the Sliding Window protocol acknowledgements must be received in the order the messages were sent. For this reason, we had to use a table to describe awaiting-acknowledgement messages, contrary to Sliding Window protocol formal specification for the purpose of verification [2], [7].

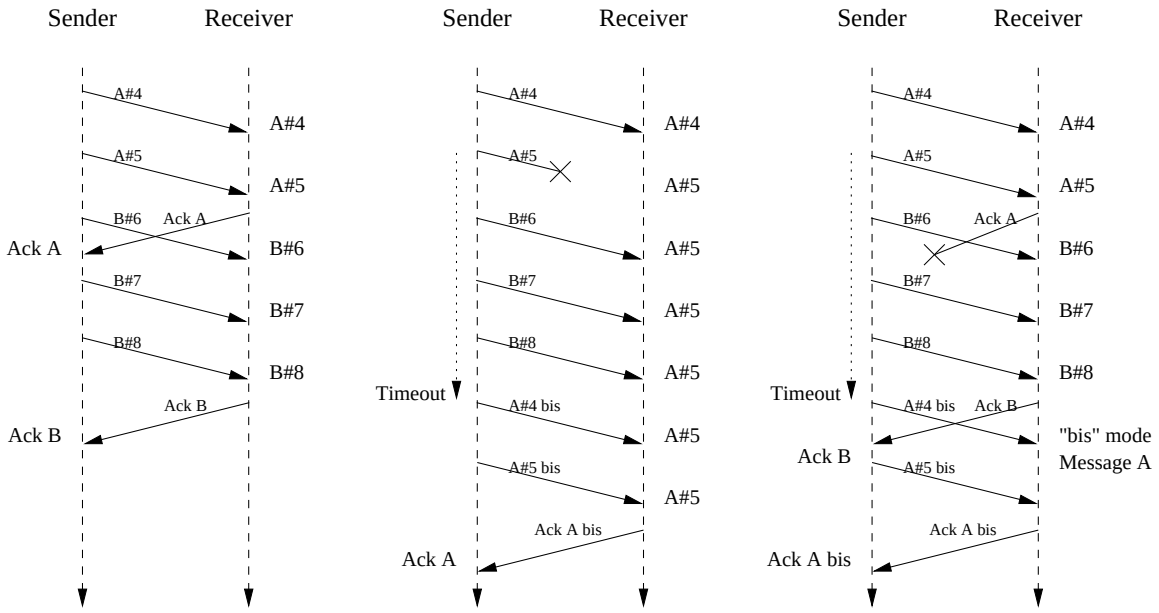


Fig. 1. Three executions of ZCSP

## III. 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?". The meaning of "not broken" will be explained in section III-C.1 This question is answered by a first model that does not include all implementation details of ZCSP. A more realistic model will be presented in section IV.

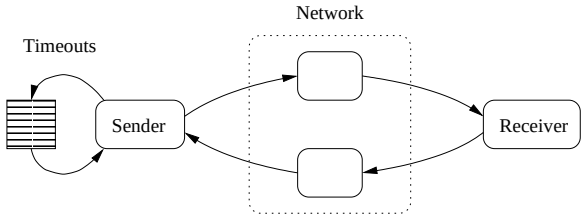


Fig. 2. Global Scheme of the finite model

### A. Specifying the system

*1) Data:* The finite set of messages to be sent is described in a table. Each message has a given set of packets and a boolean variable indicating if the message was correctly acknowledged. Each packet contains the following information:

- Its sequence number: `Sqnr`.
- The number of the related message: `Msg`.
- A bit indicating wether 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 which contains the state of each message (*not sent*, *sent* and *acknowledged*).

*2) Specification Scheme:* We have three differents components: The sender, the receiver and the network.To model the timeout signals' order, we record them in a FIFO. The global scheme can be resume as in figure 2. The Network Interface Controler is described in section III-A.3.

*3) Communications:* Each communication channel is split into three parts: a buffer with a given size, a process which reads in this buffer and writes in an output 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 III-C.1. Note that the LTL property manager is a part of the model checker SPIN, and it doesn't require any modeling, save the checked LTL formula.

### B. Processes

*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, the domain of each variable is finite. Thus, to avoid any confusion between two different messages, an acknowledgment is discarded if it doesn't

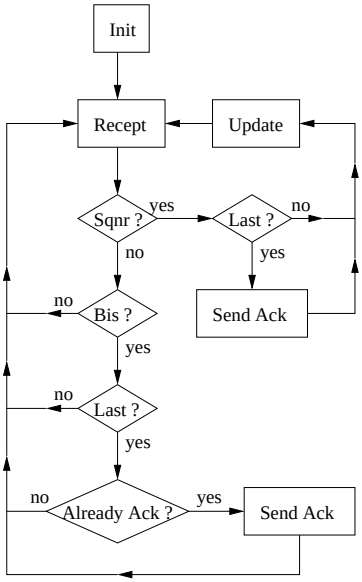


Fig. 3. the finite sequences receiver's model

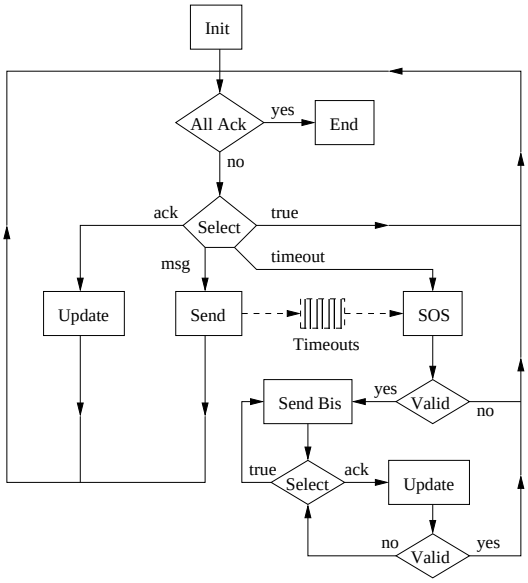


Fig. 4. the finite sequences sender's model

have the same re-emission number as the one of the linked message. In our model, we suppose that this method works and the receiver re-send an acknowledgement iff it receives the last packet of a "bis" message allready acknowledged. This choice was made under the assumption that the receiver always recognizes the real nature of any message.

*2) Sender:* The sender's behavior is described in figure 4. The most important part in this model is the "select" state. The choice between each branch means a different *scenario* and is non-deterministic.

- The "ack" guard means that an acknowledgement may be read.
- The "msg" guard means that a 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 for an 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 enters in "bis" mode (bottom-right part in the scheme).

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

### C. Validation

*1) Assumptions:* The SPIN model-checker can construct all possible executions of the specified system. Obviously, there is a class of scenarii that performs uninteresting behaviors. There is a set of trivial counter-examples (for example, considering that each packet or each acknowledgement is lost) which 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 the receiver will progress if it receives an incremental suite of sequence numbers 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 the suite $0, 1, 1, 3, 2, 3$ works also; in this case 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 $\{0, 1, 2, 3\}^*$. This is modeled by a counter named `ExpectedN`. Let `ExpectedN` be the expected sequential number. For each packet `p`, if `p.Sqn = ExpectedN` then `ExpectedN <- ExpectedN+1`. Initialy, `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

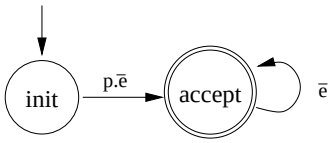


Fig. 5. Inverted Büchi automaton for $Progress \Rightarrow \mathbf{F}(AllAck)$

processes. In this section we just assume that these variables reachs an expected value. In this finite sequence model, the progression of emissions induces fairness between processes.

*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 III-C.1 is verified, $e$ means that "End State" is reached in the sender. Each state of the synchronised product may be linked to a unique state of the Büchi automaton.

*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 III-C.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 on a 1GHz *intel Pentium III* processor with 1 GB RAM.

Although these results are encouraging, conclusions on a finite model are not sufficient. Moreover, we noted a crippling defect in section III-B.2 about considering acknowledgements in "bis mode".

In section IV we will see how to deal with a non-finite sequences system.

## IV. The Infinite Sequences model

### A. A more realistic model

In this section, we presents a more realistic model of ZCSP. The data structures described in ProMeLa and the algorithms are those of the real implementation of ZCSP. As ZCSP has to transmit an undefined number of messages, the verification must be performed on a model representing these non-finite execution sequences. The real implementation uses modulo arithmetic. Considering the finite domain of each variable, each infinite sequence is $\omega$*-regular* (a finite prefix followed by a cyclic regular expression) and the LTL verification will terminate.

In section IV-C we will reduce each variables' domain to the smallest size that recreates all pertinent scenarii.

### B. Specification Scheme

*1) The System:* The global scheme of the system is shown on figure 6. As we saw in section II-D, there is no assumption on acknowledgement ordering. Thus, we use a message-entries table and variables to memorize the set of sent-but-not-yet acknowledged messages. These variables are:

- Eld_P: Index pointing the elder awaiting acknowledgement (or timeout) message.
- FF: Index pointing the "First Free" entry in the table.
- FT: Index pointing the index of the first timeout received. This information allows 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 in figure 6 represent non-deterministic behaviors. The "network processes" previously described in section III are directly implemented in the Sender and Receiver. We just replaced the action of sending with a two-guards select. Each branch may always be visited. One of them receives/sends a packet. The other throws it down. There is no difference for the global behavior but it reduces unexpected interleaving. The necessity of this kind of reduction shows the limitations of Partial Order Methods [8], [9] as we will see in section IV-D.3.

The non-deterministic behavior of the "update" process is more decisive as we will see in section IV-B.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 their 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).

*2) Sender:* Despite 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 "end new Message" and "Send Bis") is performed in an atomic sequence.

*3) Receiver:* This new process uses reemission numbers and is described in figure 8. 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) Update:* This is the most specific process of the entire system. Its behavior combined with the Sender's

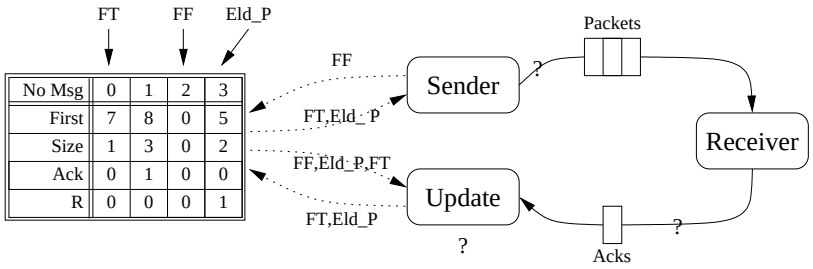


Fig. 6. The infinite sequences model's global scheme

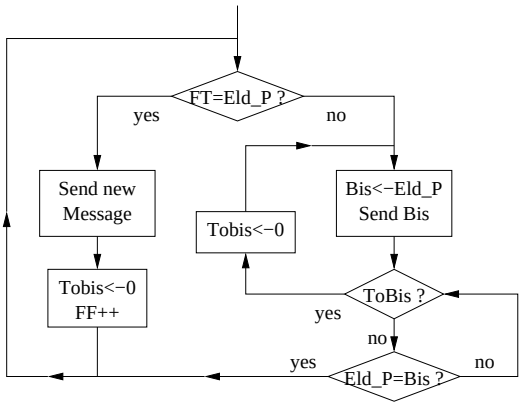


Fig. 7. The infinite sequences model's Sender

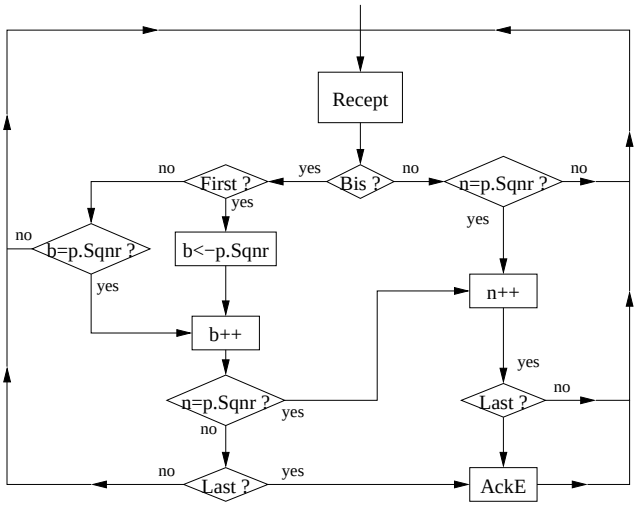


Fig. 8. The infinite sequences model's Receiver

automaton models 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 reading acknowledgements. A "good acknowledgement" contains the same re-emission number as in message-entries

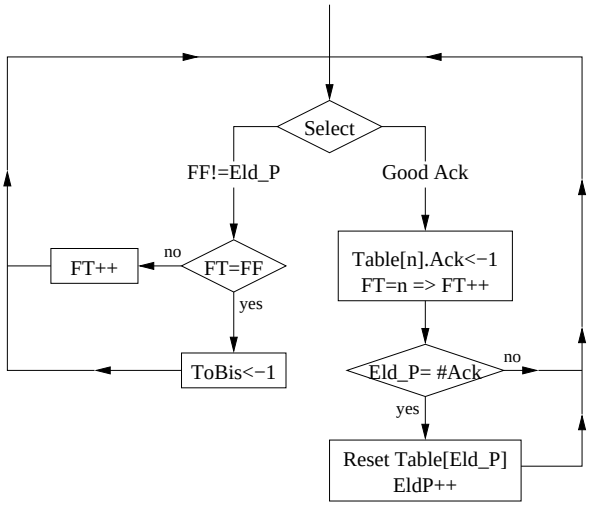


Fig. 9. The infinite sequences model's Update process

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 uninteresting properties. We will see in section IV-D.1 how to avoid this kind of phenomenon.

### C. System's minimization

Even if the real implementation uses modulo arithmetic, the promela model cannot have the same variables domains as the real implementation since the generated state space would be too big to achieve verification. We have to find the smallest definition domain for each variable that preserves the set of pertinent scenarii.

*1) Table:* We reduce the table size at three entries. There are two places for awaiting acknowledgement messages plus an empty one. In the real model this table memorizes 31 messages but has 32 entries. Thus, Eld_P, FF and FT are defined on $\mathbb{Z}/3\mathbb{Z}$.

*2) Messages:* There are three kinds of packet: The first packet of the message, the last one, and an intermediary one. Thus there are three kinds of message:

- One-packet message, this packet is both the first and the last packet.
- Two-packets message, the first packet and the last one.
- Three-packets message, the first packet, the intermediary one and the last one.

The "Size" field of empty entries is set to zero. Thus, all size fields are defined on $\mathbb{Z}/(3+1)\mathbb{Z}$.

*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 different sequential numbers wouldn't be sufficient.

Analog phenomenon is noted in formal verification of the Sliding Window protocol [7].

Thus, all Sequence Numbers are defined on $\mathbb{Z}/(2\times 3+1)\mathbb{Z}$.

*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 synchronization imposed by rendez-vous communication.

*5) Remark:* For a better understanding, figure 6 shows a table of four messages. But the validation finishes only if all the variables are defined as above. When increasing the size of only one variable's domain, SPIN runs systematically to an "out of memory" error. Thus, we had to avoid exceeding the model checker's limits.

### D. Validation

*1) Fairness: Assumptions or injonctions?:* There are two problems related to exhaustive search on the complete scenarii set.

The first difficulty is to avoid any uninteresting scenario: When a data is systematicaly thrown-down during transmission. In section III 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 sequence number and the reemission number may reach values that induce a prohibitive size of the search domain. As scenarii that perform more than one "bis emission" for a given message are not significant (since they aren't linked to a new scenario), we introduced a threshold of reemission to force a good emission to occur when this threshold is reached.

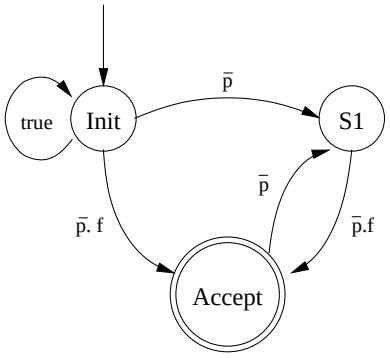


Fig. 10. Inverted Büchi automaton for Liveness property

This fairness problem may be solved in two ways:

- Forcing good emission when a threshold is reached.
- Ignoring emissions that overtakes this threshold.

In very small 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.

Another difficulty is due to potential cycles in processes. As we saw in section IV-B.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 force the system to let other processes to work infinitely often, this cycle will be simply avoided. We impose the sender to work infinitely often (This is the only process that can force the update process 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.

*2) LTL properties and generated Büchi automata:* We propose to answer 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)?

*tout mettre dans les eq* The first question is translated into the LTL formula

$$\mathbf{GF}(fairness\ between\ processes)$$

$$\Rightarrow \mathbf{GF}(sending\ progress)$$

The second question is translated in the LTL formula

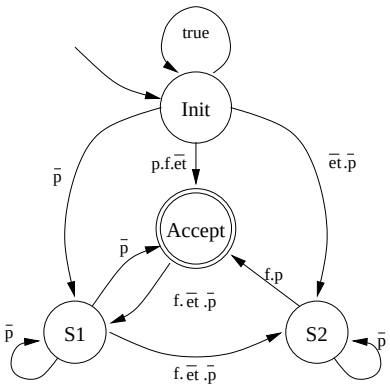


Fig. 11. Inverted Büchi automaton for Ending property

$$(\mathbf{FG}\ not(sending\ progress)$$
$$and\ \mathbf{GF}(fairness\ between\ processes))$$
$$\Rightarrow \mathbf{FG}(empty\ table)$$

The related Büchi automaton is given in Figure 11.

According to the size of this automaton, the reader may be surprised to know that the set of transitions found 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 automaton (and eventualy return into it 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.

*3) Verification Results:* We present the results obtained for the verification of the liveness of ending properties in table I. All values are brought by the SPIN model checker after validation. We used a 1Ghz intel pentium 3 CPU with 1Go RAM. We experimented some of the SPIN options and report the results in this table. Concerning the liveness property, the first column presents the results when no partial order reduction is used contrary to the second one. For the ending property, only the results with partial order reduction are shown. For each column, "Transitions" represents the sum of stored and matched states during all the verification process. It also represents the total number of fired transitions during the traversal of the *synchronous product* of the system and the Büchi automaton. The depth reached is the longest path even explored in the *synchronous product*. The memory needed represent the number of MB that SPIN would have needed if the compression option were not used (This number is extrapolated by SPIN from the results obtain when the compression is set). In our case, the compression option was mandatory.

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

TABLE I
STATISTICS ON SPIN VALIDATIONS

The memory used entry represents the effectively used memory when compression is set. The compression entry represents the ration between these two last lines. CPU times shows the elapsed time for each verification and the last line shows that both properties are verified.

The most recommanded option in SPIN is partial order reduction. Reader may find informations in [8], [9]. We can see the efficiency of this method with experimental results: In our problem the number of states stored and matched was divided by four! This option reduces interleaving but as we saw in section IV-B, it is not sufficient: "Network processes" described in section III-A.3 led to useless transitions and internal variables which increased the size of the system. Nevertheless, partial order reduction aims at avoiding this kind of useless transitions.

Bitstate Hashing option can be used to find an error but, in case of "valid" simulation, the results can't be exploited: Hash-conflict may occur between an already checked state (out of any acceptance cycle) and a never checked state which could lead to an acceptance cycle.

Reader may note some interesting values, comparing performances with and without Partial Order Reduction.

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. 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 of (or greater than) available memory.

## V. CONCLUSION

The paper describes a verification experiment of a communication protocol close to the well known Sliding Window protocol. The description level presented here is closer to the real implementation as the related works

concerning the verification of the SW and the main differences concerns the lack of ordering constraints for the acknowledgement messages; this imposes us to use a message-entries table that complicates the protocol.

We have encountered the major difficulty of modeling a real system for verification purpose that fit in the model checker. We investigated different solutions to reduce the size of our model but steel capture all the pertinent behaviors: Variables' domains reduction, interleaving reduction, fairness constraints added into the model or into the LTL formula.

The properties were verified and we have a good confidence level in our model since it is realistic and the implementation choices are represented but it was not so easy and it could have been impossible to perform the verification for a bigger system unless being an expert in both the system to be verified and the verification tool used. We are convinced that a close collaboration between these actors is mandatory to achieve the formal verification of complex systems even when the verification method used is quite easy to apprehend.

## 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*, 2002.

[2] G. J. Holzmann, "The Model Checker Spin," *IEEE Trans. on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997, special issue on Formal Methods in Software Practice.

[3] ——, *Design and Validation of Computer Protocols*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[4] A. Pnueli, "The Temporal Logic of Programs," in *18th IEEE Symp. Foundations of Computer Science*, 1977, pp. 46–57.

[5] 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)*. Chapmann & Hall, 1995, pp. 3–18.

[6] N. Stenning, "A data transfer protocol," in *Computer Networks*, vol. 1(2), 1976, pp. 99–110.

[7] D. Chkliaev, J. Hooman, and E. de Vink, "Formal verification of an improved sliding window protocol," in *Proc. of the 3d PROGRESS Workshop on Embedded Systems*, 2002.

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

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