Transforming Coloured Petri Nets to Counter Systems for Parametric Verification: A Stop-and-Wait Protocol Case Study *

Jonathan Billington and Guy Edward Gallasch  
Computer Systems Engineering Centre  
School of Electrical and Information Engineering  
University of South Australia  
Mawson Lakes, SA, 5095, AUSTRALIA  
Jonathan.Billington@unisa.edu.au  
Guy.Gallasch@postgrads.unisa.edu.au  

Laure Petrucci  
LIPN, CNRS UMR 7030, Université Paris XIII  
99, avenue Jean-Baptiste Clément  
F-93430 Villetaneuse, FRANCE  
petrucci@lipn.univ-paris13.fr

Abstract

Protocols may contain parameters that are chosen from a wide range. In some cases we would like our analysis results to apply to an arbitrary upper limit on a parameter value, such as the maximum number of retransmissions. In this case we have an infinite family of finite state systems. This makes their verification difficult. However, techniques and tools are being developed for the verification of parametric and infinite state systems. We explore the use of one such tool, FAST, for verifying several properties of the stop-and-wait class of protocols, where the maximum number of retransmissions and the maximum sequence number are considered parameters. We are also interested in using expressive languages for representing protocols such as Coloured Petri nets (CPNs). FAST’s foundation is counter systems, which are automata whose states are a vector of non-negative integers, with operations limited to Presburger arithmetic. We therefore also present some first steps in transforming CPNs to counter systems in the context of stop-and-wait protocols operating over unbounded FIFO channels.

*This work is supported by the Australian Research Council Linkage International grant LX04544639, and the French-Australian Science and Technology programme FR040062.
1 Introduction

1.1 Background and Motivation

The design and development of computer communication protocols is central to the development of embedded and pervasive computing systems which nearly always involve the co-operation of distributed components. It is important that protocols behave according to their requirements, since their failure can have serious consequences particularly for life critical or financially sensitive applications. Being able to verify that protocols behave correctly is a significant challenge since they usually include a number of parameters (such as a maximum sequence number, flow and congestion control window sizes, and the maximum number of retransmissions) that may be chosen to suit the operating environment, and may vary widely. Thus we would like to consider a class of protocols where the parameters can take any value within their range, and verify their correctness for all values of the parameters. Sometimes the ranges for these parameters are unbounded, giving rise to an infinite family of state spaces, one for each value of the parameter. At other times no limit may be placed on the value of a parameter (e.g. the number of times a packet can be retransmitted) which may result in an infinite state space.

The approach we use to tackle this problem is that of model driven development. The first step is to develop a formal model of our system which we then analyse either using tools or if they fail then by hand or a combination of both. The model is normally at the design level and the proofs are intended to show that the design satisfies the requirements of the system. This is rather important because removing errors at the design stage is very cost effective in the development of systems compared with removing errors in the implementation using testing. The effect is even more pronounced if the errors are discovered after the product has been released. The development of the model and its analysis can also increase the level of understanding of the requirements. Further, if the model is executable it can be used in fast prototyping of system specifications. This also increases the designer’s and customer’s understanding of requirements, which is widely acknowledged as a problematic area in software development.

In previous work [8] we summarised a protocol verification methodology based on Coloured Petri nets [18] and finite state automata. (Coloured Petri Nets (CPNs) are an executable modelling language with a formal semantics, based on Petri nets and the ML functional programming language.) This methodology uses state space methods and has been applied successfully for finite state systems, for small values of parameters. Techniques (such as partial orders, BDDs, and the sweep-line method) for alleviating the state space explosion problem [28] help to extend the method to larger ranges of parameters, but cannot handle large or unbounded values.

In [8], the methodology is illustrated using a stop-and-wait Protocol (SWP) [25, 22] which involves two parameters: the maximum sequence number, MaxSeqNb; and the maximum number of retransmissions, MaxRetrans. From a modelling
point of view, the values of these parameters may be chosen arbitrarily. We would thus like to prove that the SWP class is correct for any values of \( \text{MaxSeqNb} \geq 1 \) and \( \text{MaxRetrans} \geq 0 \). This becomes impossible using finite state techniques, as we need to consider an infinite number of increasingly larger finite state spaces. For FIFO channels (either lossy or lossless), a hand proof is given in [8] that shows that the number of messages in the message channel (and the number of acks in the acknowledgement channel) has a least upper bound of \( 2\text{MaxRetrans} + 1 \), for any positive value of \( \text{MaxSeqNb} \), and any non-negative value of \( \text{MaxRetrans} \). For other properties, such as verifying that the protocol conforms to its service of alternating send and receive events, the standard methodology was used for a range of parameter values \( (0 < \text{MaxSeqNb} < 1024, 0 \leq \text{MaxRetrans} \leq 4) \), but no general result was obtained. This has motivated us to search for methods that will handle unbounded parameters and provide some degree of automation.

This paper addresses the unbounded parameter problem by using a tool called FAST (Fast Acceleration of Symbolic Transition systems) [3], based on counter systems [17]. FAST performs symbolic analysis of infinite state systems by using accelerations (meta-transitions) to encode an arbitrary number of iterations of sequences of actions within the system. Parameters can be input as variables that are not constrained, and hence automated parametric verification of systems may be possible. However, we face two difficulties using this tool. Firstly, FAST’s input language is based on counter systems (CS), whereas we would like to use the much more expressive language of CPNs. CS can model Place/Transition nets augmented with special arc types such as inhibitors [15], but as far as we are aware no attempt has previously been made to translate CPNs to CS. Secondly, FAST provides a semi-algorithm, which is not guaranteed to terminate. Hence we can never be sure the verification will succeed.

The purpose of our work is thus to explore the potential of FAST for the parametric verification of communication protocols which have been previously modelled using CPNs. This paper investigates the class of stop-and-wait protocols. This is because they require parametric verification and are the simplest representative example of the class of protocols which provide flow control and bit error recovery that are used in practice, such as in the data link and transport layers of communication protocol architectures. We slightly revise our CPN model in [8] to make it easier to translate to a counter system. We find that translating CPN places representing states, stored sequence numbers and the retransmission counter is straightforward, but queues are more of a challenge. We are able to use 4 integer variables to represent the FIFO queue, due to the operation of the SWP. The conditions that are required for the queue model to be valid are checked using FAST, as well as the following properties: channel bounds; deadlocks; the stop-and-wait property; in-sequence delivery; and message loss and duplication; for both lossless and lossy FIFO channels.
1.2 Related Work

The simplest SWP restricts its sequence numbers to 0 and 1 and is known as the Alternating Bit protocol (ABP) [5]. The ABP and its extensions (e.g. [12]) have been used extensively in the literature as case studies (e.g. [21]). Often such papers demonstrate in various ways whether the ABP works as expected over (lossy or lossless) FIFO channels [24, 9], investigate performance [23, 20], demonstrate new tools [9], or illustrate verification methodologies [14], the application of formal description techniques [27], new modelling languages or derivatives of existing languages [24, 23, 20]. However, none of these papers address the issue of parametric verification of the ABP (i.e. for arbitrary values of MaxRetrans).

More recently there has been work in the area of symbolic verification of the ABP. Valmari and his co-workers (e.g. [29]) promote a behavioural fixed point method and compositional techniques for the verification of parametric systems. In [29] a variant of the ABP using limited retransmission, i.e. where there is an arbitrary bound (e.g. MaxRetrans) on the number of retransmissions, is verified using Valmari’s CFFD equivalence. There are several differences with our work. Perhaps the most significant is that the channels are limited to holding only one message or acknowledgement at a time, whereas ours are unbounded FIFO queues. Valmari [29] concedes this to be a much more difficult problem. Valmari’s method relies on defining a separate counter process which needs to be synchronised (using parallel composition) with the sender logic, which has 18 states. The counter itself is a recursive parallel composition of counter cells. The receiver is a relatively straightforward 6 state process. The ack channel is given as a 3 state process, but the data channel is more complex and not given explicitly in the paper. To obtain the model, all these processes need to be synchronised with parallel composition. In contrast our CPN model integrates all these aspects in the one model, and extends the model to include unbounded FIFO queues and sequence numbers with an arbitrary maximum sequence number as a parameter. However, our model does not have explicit communication with the users (but relies on the send and (non-duplicate) receive transitions to be considered as synchronised communication with the user) and does not consider reporting errors to the user. We see no problem in extending our model to include these features, however, our aim is to illustrate the use of FAST in analysing parameterised CPN models, rather than a direct comparison with a particular ABP variant.

The ABP and another variant called the Bounded Retransmission Protocol are used in [2] to demonstrate a symbolic verification methodology [1]. TReX (Tool for Reachability Analysis of Complex Systems) [26] was used to implement this methodology in [2]. The content of unbounded lossy FIFO channels is modelled by (a restricted class of) regular expressions thus providing a symbolic representation of the channels. Similar to FAST, an acceleration technique is used. This allows a small symbolic state space to be calculated based on the states of the sender and receiver ABP processes. They verify that the ABP conforms to its service of alternating sends and receives, using the Aldebaran tool [11] for finite state
automata. The maximum number of retransmissions was considered to be unlimited giving rise to a single, infinite-state model. This differs from our approach of modelling MaxRetrans as a parameter and thus having an infinite number of finite-state models, one for each parameter value. As mentioned above, we also model an arbitrary maximum sequence number, rather than being limited to a maximum sequence number of 1.

1.3 Contribution and Organisation

This paper provides two contributions. Firstly, we believe it is the first time that parametric verification of the stop-and-wait protocol class operating over unbounded FIFO channels has been undertaken where MaxRetrans has been modelled as a parameter. We are able to verify the SWP for arbitrary values of MaxRetrans for small values of MaxSeqNb (i.e. 1 to 5), for an extensive range of safety properties. Secondly, we provide some steps towards a method for translating CPNs into counter systems.

The rest of the paper is organised as follows. Section 2 describes the stop-and-wait protocol using a Coloured Petri net model. Counter Systems are introduced in Section 3 which also describes a methodology for translating a CPN model into a CS. This methodology is applied in Section 4 to the SWP CPN of Section 2. The expected properties of the SWP are described in Section 5. After introducing FAST, we analyse the SWP CS in Section 6. Section 7 provides concluding remarks and identifies areas of future work.

2 Stop-and-Wait Class of Protocols: A CPN Model

We explain the class of stop-and-wait protocols by providing a parameterised Coloured Petri Net (CPN) model of it as shown in Figs. 1 and 2, which were created using Design/CPN [13]. Essentially three changes are made to the CPN model presented in [7, 8]:

- in the sender, one place (instead of two) is used to store its states, so that the colour set Sender comprises two states: s_ready and wait_ack;
- one place (receiver.state) is used in the receiver to store its states;
- a new place in the receiver stores its current sequence number; and
- arc inscriptions are revised accordingly.

This makes the CPN diagram more compact and provides a consistent modelling style. Control flow is indicated by bold arcs.

The protocol operates between a sender, shown on the left of Fig. 1 and a receiver shown on the right. The communication medium (Network) is represented by two lossy FIFO queues, one for each direction of message flow. The queues are modelled by using a list type for places mess_channel and ack_channel, adding
messages to the end of the queue (using the operator `^`) and removing messages from the head of the queue (using `::`). Loss is represented by arbitrarily removing the head of the queue.

The protocol is implemented by the sender and receiver procedures. The sender has two states: _s_ready_ and _wait_ack_, with the current state stored in place `sender_state`. When ready, it sends a message (transition `send_message`) and waits for an acknowledgement before sending the next message (hence, stop-and-wait). To overcome the possibility that the message has been lost, the sender sets a timer running on sending a message, and if it expires before receiving the acknowledgement (`receive_ack`), the message is retransmitted (transition `timeout_retrans`) and the timer is set running again. However, the acknowledgement could be lost even though the message had been received. In this case, the receiver needs to detect and discard duplicate messages. To do this, a sequence number is associated with

---

**Figure 1:** CPN of the SWP operating over a lossy FIFO channel.

```plaintext
val MaxRetrans = 1;
val MaxSeqNb = 1;

color Sender = with s_ready | wait_ack;
color Receiver = with r_ready | process;
color Seq = int with 0..MaxSeqNb;
color RetransCounter = int with 0..MaxRetrans;
color Message = Seq;
color MessList = list Message;

var sn, rn : Seq;
var rc : RetransCounter;
var queue : MessList;

fun NextSeq(n) = if(n = MaxSeqNb) then 0 else n+1;
```

**Figure 2:** Global Declarations for the Stop-and-Wait Protocol CPN.
each message, and stored by the sender (place send_seq_no). In the CPN model, messages are represented by their sequence number only, as data is not used in the procedures. The receiver also stores a sequence number (place recv_seq_no) which is used to detect duplicates. The sequence number space is finite, but allows for any range of consecutive integers, starting from zero. In our model we use the parameter MaxSeqNb to represent the maximum value of the sequence number space.

If a new message arrives at the receiver (transition receive_mess with sn=rn), the sequence number is incremented, modulo MaxSeqNb (NextSeq(rn)), whereas if a duplicate is detected (sn≠rn) the sequence number remains the same. The sequence number in the receiver represents the next message to be received, and this is the value that is sent back to the sender, as an acknowledgement. Acknowledgements are returned on receipt of each message, whether or not it is a duplicate. (This is required to recover from loss of acknowledgements.) To model flow control, we have two states in the receiver (r.ready and process), which allow the sending of the acknowledgement to be asynchronous with the receipt of a message. The current state is stored in place receiver_state.

While waiting for an acknowledgement, the sender may continue to retransmit messages, until it reaches a preset limit (MaxRetrans). It then gives up hope of the message getting through and passes control to a management entity for higher level recovery (not modelled). If an acknowledgement arrives before this, indicating that the message has been received (rn=NextSeq(sn)) transition receive_ack increments the send sequence number and returns the sender to ready, allowing the next message to be sent. Duplicate acknowledgements are discarded by the sender (receive_dup_ack) at any time.

3 Mapping the CPN Model to a Counter System

As we aim at obtaining an extensive set of analysis results on the stop-and-wait protocol, parametric analysis is desirable. It can be achieved using tools such as FAST [3]. FAST operates on counter systems, so it is necessary to transform our CPN model into a CS. This is straightforward for Petri nets, even with extended arcs [6, 4] but requires enhancement for CPNs.

Counter systems are automata extended with unbounded integer variables. FAST uses accelerations (sometimes called meta-transitions) to enable it to calculate the exact effect of iterating a behavioural loop an arbitrary number of times, and produces a symbolic occurrence graph representing the infinite state system. Details on counter systems and the theory behind FAST can be found in [17, 10, 30, 19].

The places of a CPN are transformed into a set of counter system variables and a single counter system state. This transformation is straightforward if the types of the places are or can be mapped to integers (e.g. enumerated types). If a place p has a type Type(p) that can be mapped to the integers by an injective mapping, \( I_p : Type(p) \rightarrow \mathbb{N} \), and p always contains one token (\( \forall M \in [M_0], |M(p)| = 1 \),
then we can create an integer variable \( v_p \) in the CS, that takes the values of the token in the place transformed by \( I_p \) for each marking. This is the case both for the places in the sender and those in the receiver of our SWP CPN model.

However, the stop-and-wait protocol uses two FIFO queues: one for messages and one for acknowledgements, represented by places \( \text{mess\_channel} \) and \( \text{ack\_channel} \) both typed by a message list, where messages are represented by sequence numbers (integers). These queues can be any size, depending on the maximum number of retransmissions [8]. The values of the sequence numbers depend on the \( \text{MaxSeqNb} \) parameter. Because the sequence numbers are integers we can store the value of a queue item in a variable, and the number of queue items of that value in an associated variable. As long as sequence numbers do not wrap, we can always remove the item with the ‘smallest’ value from the queue and hence maintain FIFO order. However, this will require an unbounded number of variables in the general case, but not if the queue can only contain a finite number of values at any one time. For example, if the queue can contain only one message value at a time, then it can be represented by two variables: one storing the message value and a second storing the number of messages in the queue.

For the SWP operating over FIFO channels it turns out that there can be at most two different messages (represented by their sequence numbers) in the queue at any one time and that the messages of the same type are contiguous in the queue. Thus the queue is of the form \( \text{mess}_1*\text{mess}_2* \). (Before doing the analysis, this property is a conjecture, so we must check that this property holds as a first step in validating the model.) Therefore, the queue can be modelled using four variables:

- \( \text{Old} \) is the smallest/oldest sequence number (modulo \( \text{MaxSeqNb} \)) that is in the queue;
- \( \text{New} \) is the latest sequence number that was put in the queue;
- \( \text{NbOld} \) is the number of messages with sequence number \( \text{Old} \);
- \( \text{NbNew} \) is the number of messages with sequence number \( \text{New} \).

Now, we will explain how to add messages to and remove messages from the queue. We will also show that this is done in a consistent manner.

The queue can contain:

1. no message. Hence \( \text{NbOld} = \text{NbNew} = 0 \);
2. one type of message. Then, \( \text{Old} = \text{New} \) and \( \text{NbOld} = \text{NbNew} \neq 0 \);
3. two types of message. Thus, \( \text{Old} \neq \text{New} \), \( \text{NbOld} \neq 0 \) and \( \text{NbNew} \neq 0 \).

In the following, a prime denotes the value of the variable after an action has been performed. Variables that do not change are not mentioned.

First, consider adding a message with sequence number \( \text{mess} \). If the queue is in state (numbered as above):
1. The new message is the only one. Therefore, after adding the message: \( \text{Old}' = \text{New}' = \text{mess} \) and \( \text{NbOld}' = \text{NbNew}' = 1 \). This is consistent with the above statement for a queue having a single message, hence containing only one type of message;

2. The new message can be either:
   - of the same type as those already in the queue. Then, after adding the new message, we have: \( \text{NbOld}' = \text{NbNew}' = \text{NbOld} + 1 (= \text{NbNew} + 1) \). This is consistent with the queue containing a single type of message;
   - of a new type. Thus, \( \text{New}' = \text{mess} \) and \( \text{NbNew}' = 1 \). This is consistent with the queue now having two types of message.

3. In this case, only a \( \text{New} \) message (i.e. a duplicate) can be added to the queue and hence \( \text{NbNew}' = \text{NbNew} + 1 \). This is consistent with the queue containing exactly two types of message.

Now, we explain how to remove a message \( \text{mess} \). If the queue is in state:

1. It is empty, so this case should never occur as there is nothing to consume;

2. The message consumed is of the single type in the queue. Hence: \( \text{mess} = \text{Old} = \text{New} \) and \( \text{NbOld}' = \text{NbNew}' = \text{NbOld} - 1 = \text{NbNew} - 1 \). Note that the resulting queue can either contain messages of the same single type or no message at all;

3. The message consumed can be of type either \( \text{New} \) or \( \text{Old} \). Both cases can be handled in a similar manner, but in this paper, the queues considered are FIFO. Therefore, the message consumed is the oldest in the queue, i.e. \( \text{mess} = \text{Old} \). Then two cases can be considered:
   - The message consumed is the last one of type \( \text{Old} \), i.e. \( \text{NbOld} = 1 \). Then the resulting queue contains a single type of message, \( \text{Old}' = \text{New} \) and \( \text{NbOld}' = \text{NbNew} \);
   - There are several messages of type \( \text{Old} \) in the queue. Then, \( \text{NbOld}' = \text{NbOld} - 1 \).

4 The SWP CS Model

The CPN model of the stop-and-wait protocol can now be transformed into a counter system by application of the techniques from the previous section.

4.1 SWP CS Variables

We first start with mapping the places of the SWP CPN to CS variables.

\text{sender.state} can take two values, i.e. \text{s.ready} or \text{wait.ack}. It is coded, in the CS, using a variable \text{SState}, with values 1 and 0 respectively;
send_seq_no becomes a variable SSeqNb, containing the last not-acknowledged sequence number;

retrans_counter is a variable Retrans, counting the number of retransmissions that have occurred for message number SSeqNb;

receiver_state can take two different values, i.e. ready or process. It is represented using a variable RState, with values 1 and 0, respectively;

recv_seq_no becomes a variable RSeqNb, containing the number of the next expected message;

mess_channel is modelled in the counter system using, as explained before, 4 variables MCOld, MCNew, NbMCOld and NbMCNew. They represent respectively, the sequence number of the message in the channel that was put first, the sequence number of the message in the channel that was put last, and the numbers of such messages;

ack_channel is modelled similarly with variables ACOld, NbACOId, ACNew and NbACNew.

Two other variables are needed for the parameters of the system: the maximum sequence number MaxSeqNb and the maximum number of retransmissions MaxRetrans.

### 4.2 SWP CS Transitions

When modelling the transitions, we must ensure that all possible cases are taken into account. In fact, we will only include firable transitions in the model, and not transitions that can never occur (dead transitions). It is important to reduce the number of transitions as in our case transition compositions (see Section 6.2) depend on the square of the number of transitions, thus increasing the computation time significantly. However we will check in Section 6 that this is the case, to ensure that nothing is missing. The transitions operate on the 4 variables describing queues. The wrapping from MaxSeqNb to 0 must be taken into account. The value of the other variables are changed as described in the CPN model.

### 4.3 The Stop-and-Wait protocol CS Model

Here, we show an excerpt of the SWP CS model, illustrating FAST model input. The model describes in a natural way the counter system to analyse. It comprises the integer variables identified above, the single state marking of the counter system obtained from the Petri net, and specifications of the transitions. Each transition is described by its source and destination states (from and to fields), which is here always the state marking. A guard is associated with each transition, giving an enabling condition on the values of the variables. The effect of the transition is given in action, describing how the values of variables are changed when the
transition occurs. The symbol \&\& indicates logical AND, \|\| represents logical OR, \! indicates negation, and the prime notation is as defined in Section 3.

model SWP {
    var SState, SSeqNb, Retrans, MaxRetrans, MCOld, MCNew, NbMCOld, NbMCNew, 
    ACOld, ACNew, NbACOld, NbACNew, RSeqNb, RState, MaxSeqNb;
    states marking;
}

// send: case new message with no message in queue
transition sendM1 := {
    from := marking;
    to := marking;
    guard := SState=1 && NbMCOld=0;
    action := SState'=0, MCNew'=SSeqNb, NbMCNew'=1, MCOld'=SSeqNb, NbMCOld'=1;};

// receive duplicate: case message with seq nb MCNew = MCOld
transition receiveM1 := {
    from := marking;
    to := marking;
    guard := RState=1 && NbMCOld>0 && !(MCOld=RSeqNb) && MCOld=MCNew;
    action := RState'=0, NbMCOld'= NbMCOld-1, NbMCNew'=NbMCNew-1;};

// receive duplicate ack: case ack with seq nb ACNew = ACOld
transition recdupack1 := {
    from := marking;
    to := marking;
    guard := NbACOld>0 && ACOld=ACNew && ((SSeqNb=MaxSeqNb && ! (ACOld=0))
    || (SSeqNb<MaxSeqNb && !(ACOld=SSeqNb+1)));
    action := NbACOld'=NbACOld-1, NbACNew'=NbACNew-1;};

// receive expected ack
transition recack := {
    from := marking;
    to := marking;
    guard := NbACOld>0 && ACOld=ACNew && SState=0 && ((SSeqNb=MaxSeqNb &&
    ACOld=0) || (SSeqNb<MaxSeqNb && ACOld=SSeqNb+1));
    action := NbACOld'=NbACOld-1, NbACNew'=NbACNew-1, SState'=1,
    Retrans'=0, SSeqNb'=ACOld;};

...
5.1 Model Soundness

For the model to be sound, we need to verify the modelling assumptions. Our model is correct if both the message and acknowledgement channels: contain no more than two different types of message, where the ‘type’ of the message refers to its sequence number (i.e. Old and New from Section 3); and all messages of the same type are contiguous in the queue (i.e. the contents of the queue is of the form Old*New). To verify this, we check that if there are already two types of message in the queue (i.e. Old and New), only transitions which can add a New message are enabled.

We also verify the completeness of the model, i.e. that all the relevant cases are taken into account by transitions. Hence, all cases that are not explicitly described by the guards can never occur. This is done by verifying that there is no reachable marking that enables similar transitions, but with different conditions concerning the channel contents.

5.2 SWP Properties

We wish to prove the following SWP properties:

Consecutive sequence numbers If there are different types of message in a channel, they have consecutive numbers. Hence:

\[
\begin{align*}
MCOld \neq MCNew & \Rightarrow (MCNew = MCOld + 1 \vee (MCNew = 0 \land MCOld = \text{MaxSeqNb})) \\
ACOld \neq ACNew & \Rightarrow (ACNew = ACOld + 1 \vee (ACNew = 0 \land ACOld = \text{MaxSeqNb}))
\end{align*}
\]

Number of messages in channels The lowest upper bounds for the number of messages in both channels, and the lowest upper bound on the total number of messages (i.e. messages plus acknowledgements) is \(2\text{MaxRetrans} + 1\). This is checked by counting the messages in the channels. The number of messages in the message channel is:

\[
Nb\_Messages = \begin{cases} 
\text{if MCOld} \neq \text{MCNew then} & Nb\_MC\_Old + Nb\_MC\_New \\
\text{else} & Nb\_MC\_Old 
\end{cases}
\]

Hence, for the message channel:

\[
Nb\_Messages \leq 2\text{MaxRetrans} + 1
\]

should hold over all reachable markings, but

\[
Nb\_Messages \leq 2\text{MaxRetrans}
\]

should not. Similarly for the acknowledgement channel. The boundedness property can be even more precise, taking into account the types (sequence numbers) of messages:

\[
\begin{align*}
\text{if MCOld} \neq \text{MCNew then} & Nb\_MC\_Old \leq \text{MaxRetrans} \land Nb\_MC\_New \leq \text{MaxRetrans} + 1 \\
\text{else} & Nb\_MC\_Old \leq \text{MaxRetrans} + 1
\end{align*}
\]
Stop-and-Wait Property  A sent message is received before the next (new) message is sent (i.e. alternating send and receive events.)

No data loss  Each original message (or a retransmission) is eventually received, except for the last message in case the original plus all retransmissions were lost, and the maximum number of retransmissions is reached.

No duplication  When a duplicate message arrives, it is detected as such and discarded. No duplicate message is mistakenly accepted as a new one.

In-sequence delivery  The messages are received in the order they are sent.

Deadlocks  When using reliable channels, there should be no deadlock. When using unreliable channels, only expected deadlocks should exist: the maximum number of retransmissions is reached but the sender is stuck waiting for an acknowledgement, and both message and acknowledgement channels are empty:

\[
\text{retrans} = \text{MaxRetrans}, \ S\text{State} = 0, \\
\text{MCOld} = \text{MCNew}, \ Nb\text{MCOld} = Nb\text{MCNew} = 0, \\
\text{ACOld} = \text{ACNew}, \ Nb\text{ACOld} = Nb\text{ACNew} = 0
\]

5.3 Instrumentation of the model

In order to check several of the properties, some instrumentation of the model is required. We add a variable \(SRprop\), which is set to the sequence number plus 1, when a new message is sent. When an expected message (i.e. not a duplicate) is received, this variable is set to 0. Checking the stop-and-wait property then amounts to verifying that there is no pending new message in the message channel when the sender is ready to send, i.e. no state such that:

\[SRprop > 0 \land S\text{State} = 1\]

When operating over a FIFO medium, because the stop-and-wait property holds (a new message can only be sent if the expected one was received) it follows that there is no loss of data (except possibly the last message as described.)

To verify the no duplication property, we check that there is no state such that the receiver is ready to accept a new message with sequence number other than the most recently sent by the sender, i.e. there is no state such that:

\[SRprop = \text{MCOld} + 1 \land R\text{State} = 1 \land Nb\text{MCOld} > 0 \land \neg(\text{MCOld} = \text{RSeqNb})\]

Effectively, when a duplicate is received, the value of \(SRprop\) should be either 0 if no new message has yet been sent by the sender, or corresponds to the sequence number plus 1 of the new message sent, i.e. a different sequence number to the duplicate being received.

Finally, to prove the in-sequence delivery property, we note that variable \(SRprop\) contains the number (plus one) of the last new message sent, and that it is not
possible to receive an original message with a sequence number different to that most recently sent, i.e.:

\[\neg (SRprop = MCOld + 1) \land RState = 1 \land NbMCOld > 0 \land MCOld = RSeqNb\]

6 Analysis of SWP using FAST

6.1 Introduction to FAST

FAST [3, 4] is a tool dedicated to checking safety properties on counter systems. The main issue addressed by FAST is the exact computation of the (infinite) state space. On such a complex problem, although FAST uses a semi-algorithm which is not guaranteed to terminate, experiments with its use on practical examples have been promising [16].

6.1.1 Inputs and Outputs

FAST inputs are in the form of both a model and a strategy for the analysis. Outputs are messages indicating whether the system satisfies a property or not. The model input format was described in Section 4 where an excerpt of our SWP model was given.

The strategy is the sequence of computations to perform in order to check the validity of the system. The strategy language is a script language which operates on regions (sets of states), transitions and booleans. All the usual operators on sets are available and primitives to compute the reachability set (forward or backward) are provided. Checking a safety property involves declaring the initial states, computing the reachability set \(A\), declaring the property to check (good states) \(B\), and testing if \(A \subseteq B\).

Here, we show an excerpt of the SWP CS strategy, illustrating FAST strategy input.

```plaintext
strategy SWP {
...  
Region init := {SState=1 && SSeqNb=0 && Retrans=0 && MCOld=0 && MCNew=0 && NbMCOld=0 && NbMCNew=0 && ...};

Region reach := post*(init, t, 2);
  
// Consecutive sequence numbers in Message channel
Region diffminmaxM := {(MCOld=MCNew) || (MCNew=MCOld+1) || (MCOld=MaxSeqNb && MCNew=0)};

if (subSet(reach,diffminmaxM)) then
  print("M channel consecutive seq numbers OK");
else print("M channel consecutive seq numbers NOK");
endif

...
}
```
First, a region $\text{init}$ is declared, used to describe the initial states. Then, the set of reachable states $\text{reach}$ is computed from $\text{init}$, using forward reachability (function $\text{post}^*$). Region $\text{diffminmaxM}$ characterises the set of states with consecutive sequence numbers in the message channel. If $\text{reach}$ is a subset of $\text{diffminmaxM}$ then the consecutive sequence numbers property is satisfied, otherwise it is not. An appropriate message is printed.

### 6.1.2 Architecture

The FAST computational engine can be used as a standalone application, or with a graphical user interface in a client-server architecture [4]:

- **the server** is the computation engine of FAST. It contains a Presburger library, the acceleration algorithm and the search heuristics;
- **the client** is a front-end which allows interaction with the server through a graphical user interface. This interface facilitates guided editing of models and strategies, with features such as pretty printing and predefined strategies. Once the computation starts, feedback is supplied through different measures and graphs (time elapsed, memory used, number of states, ...).

### 6.2 Analysis Results

The results obtained by FAST confirm the expected properties from Section 5. They are automatically checked for all values of the MaxRetrans parameter, although FAST did not terminate in a reasonable amount of time when the maximum sequence number was also a parameter. Therefore, we conducted separate runs of FAST with MaxSeqNb fixed, over the range from 1 to 5. The analysis was performed on the lossy channel model as well as on a model with reliable channels (where the loss transitions were removed).

Column **Channels** in Table 1 indicates whether the channel is reliable or lossy. MaxSeqNb gives the values of the variable for the experiment.

The computation is done at a reasonable or even low cost w.r.t. both time and memory usage, as shown by the experimental results in Table 1 for $1 \leq \text{MaxSeqNb} \leq 5$. The FAST computation is divided into three steps (for technical details, see e.g. [17]):

- transition compositions which take 2 minutes 5 seconds in the lossy case (529 compositions) and 1 minute 36 seconds in the reliable case (289 compositions);
- accelerations computation which takes 31 seconds for 126 cycles in the lossy case and 1 minute 13 seconds for 105 cycles in the reliable case;
- applying the accelerations to construct the state space.
The computation time for compositions and accelerations is exactly the same for all cases, as the same preliminary computations are performed. The differences in time result from the size of the internal representation of each element.

Table 1 gives the total computation times, as well as the peak memory recorded. Column Nb states indicates the number of symbolic states at the end of the computation.

<table>
<thead>
<tr>
<th>Channels</th>
<th>MaxSeqNb</th>
<th>Nb compositions</th>
<th>time (hh:mm:ss)</th>
<th>memory (MB)</th>
<th>Nb states</th>
<th>Nb accelerations</th>
<th>Nb cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable 1</td>
<td>289</td>
<td>00:07:34</td>
<td>31</td>
<td>74</td>
<td>64</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Reliable 2</td>
<td>289</td>
<td>00:36:29</td>
<td>37</td>
<td>167</td>
<td>113</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Reliable 3</td>
<td>289</td>
<td>00:54:26</td>
<td>44</td>
<td>169</td>
<td>120</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Reliable 4</td>
<td>289</td>
<td>02:07:14</td>
<td>48</td>
<td>349</td>
<td>123</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Reliable 5</td>
<td>289</td>
<td>03:00:16</td>
<td>48</td>
<td>360</td>
<td>181</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Lossy 1</td>
<td>529</td>
<td>00:12:29</td>
<td>19</td>
<td>87</td>
<td>60</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>Lossy 2</td>
<td>529</td>
<td>00:33:52</td>
<td>23</td>
<td>205</td>
<td>132</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>Lossy 3</td>
<td>529</td>
<td>01:28:56</td>
<td>27</td>
<td>199</td>
<td>193</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>Lossy 4</td>
<td>529</td>
<td>03:04:54</td>
<td>38</td>
<td>446</td>
<td>202</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>Lossy 5</td>
<td>529</td>
<td>03:30:21</td>
<td>39</td>
<td>432</td>
<td>233</td>
<td>126</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Experimental results

7 Conclusions and Future Work

Finite state methods for protocol verification can fail due to state explosion when considering ranges of values for important parameters such as the maximum number of retransmissions or the size of the sequence number space. When considering these parameters, we would like to provide a general result that allows protocol properties to be proved for any value of each parameter. When arbitrary values are considered, we need to generate an infinite number of finite state spaces, one for each value of the parameter. (This is quite different from considering, for example, the specific case of no limit on the number of retransmissions, which gives rise to a single infinite state system.)

This paper has addressed this problem for the stop-and-wait class of protocols, where we modelled the parameters explicitly. We used a recently developed tool called FAST to facilitate parametric verification. FAST allows symbolic state spaces to be generated by taking advantage of encoding arbitrary iterations of sequences of events, known as accelerations. It is based on counter systems, which are automata where states are vectors of (unbounded) integers.

The stop-and-wait protocol (SWP) has two parameters: MaxRetrans representing the maximum number of retransmissions; and MaxSeqNb representing the maximum sequence number that can be used. In previous work [8] we modelled the SWP using Coloured Petri Nets and provided a hand proof that the bound on the number of messages in the FIFO communication channel was 2 MaxRetrans +
1. However, we were only able to prove other properties, such as the stop-and-wait property of alternating sends and receives, for up to 10 bit sequence numbers and with up to 4 retransmissions using automated finite state techniques.

In this paper we have overcome these limitations for the MaxRetrans parameter. Fully automatic proofs have been obtained for channel bounds (confirming the previous hand proofs and including proving that the sum of the messages and acknowledgements in the channels does not exceed 2 \(\text{MaxRetrans} + 1\)), the stop-and-wait property, that there is no loss of messages (except for the last one when the maximum number of retransmissions is reached), no duplication and that messages are delivered in-sequence. This has been done for \(1 \leq \text{MaxSeqNb} \leq 5\). Unfortunately, FAST does not terminate in a reasonable amount of time when MaxSeqNb is considered as an unbounded parameter, or for values greater than 5. However, we believe this experience will assist us with hand proofs that the results hold for any positive integer value of MaxSeqNb.

Further we have shown how to translate our CPN model into a counter system by using a novel approach to represent a FIFO queue by 4 integer variables. This is valid when the queue can hold only two types of message indicated by their sequence numbers and all messages of the same sequence number are adjacent. This condition is proved using FAST as part of model validation. Some general guidance has also been given for translating CPNs to counter systems.

Future challenges include generalising the method to channels that allow re-ordering of messages and formally incorporating data independence, which has been assumed in our work so far. Other ways of representing queues (perhaps with one integer variable) that are efficient and suit the FAST framework of Presburger arithmetic could be investigated. A more general and formal translation of CPNs into counter systems is also of interest, to allow models that have already been constructed in CPNs to be automatically translated and input to FAST. Automatically translating the properties formulated on the CPN model to those on the counter system and translating the results back is also an interesting issue. We would also like to investigate the use of other tools such as TReX and compare them with FAST.

Acknowledgements

We thank the anonymous referees for their useful comments which helped us to improve our paper.

References


