Abstraction of Communication Channels in Promela: a Case Study*

Elena Fersman and Bengt Jonsson

Dept. of Computer Systems, P.O. Box 325, S-751 05 Uppsala, Sweden,
email: {elenaf,bengt}@docs.uu.se

Abstract. We present a case study of how abstractions can be applied to a protocol model, written in Promela, in order to make in amenable for exhaustive state-space exploration, e.g., by SPIN. The protocol is a simple version of the Five Packet Handshake Protocol, which is used in TCP for transmission of single messages. We present techniques for abstracting from actual values of messages, sequence numbers, and identifiers in the protocol. Instead, an abstract model of the protocol is constructed of variables which record whether variables and parameters of messages are equal or unequal. The abstraction works because the protocol handles identifiers and parameters of messages in a simple way. The abstracted model contains only on the order of a thousand states, and safety properties have been analyzed by SPIN.

1 Introduction

When trying to analyze any reasonable communication protocol using a model-checker, one of the biggest problems is to model it in such a way that the model-checker can analyze it exhaustively. Anyone who has used a model-checker can testify that this is not a trivial problem. In order to overcome the problem, one must deal with those aspects of a protocol that case the state-space to blow up: unbounded channels, large domains of sequence numbers and data, other data structures, etc. An experienced protocol modeler can make judicious modeling of such aspects without compromising the results of analysis. There are also general principles, e.g., based on abstract interpretation, that can guide approximation in modeling. However, we still do not have a generally available "cookbook" of standard recipes to be used in protocol modeling. This lack becomes apparent, e.g., in teaching. Students easily become frustrated when they discover the limitations of a model-checker; being able to provide well-proven and generally applicable cures to overcome such limitations would very likely contribute to the spreading of modeling and model-checking.

This paper does not attempt to present a "cook-book", but rather illustrate abstraction techniques techniques that can be applied to some common causes of state-space explosion in model-checking. More precisely, we present a case-study

* support in part by the ASTEC competence center, and by the Swedish Board for Industrial and Technical Development (NUTEK)

on the modeling in Promela of the so-called Five Packet Handshake Protocol, which is used in TCP for transmission of single messages. Descriptions of the protocol appear, e.g., in [Bel76] and [Lyn96, pp. 718–729]. We focus on abstraction of some aspects of the protocol that makes it hard to model and analyze it in SPIN:

- The protocol works under rather general assumption about the communication channel, which can reorder, lose, and duplicate messages.
- The protocol uses an unbounded set of identifiers.

These two aspects usually imply that it is not possible to perform exhaustive state-space exploration of a naive Promela model. We will therefore present some abstraction techniques which allow to decrease the state-space significantly, in order to make exhaustive analysis possible. The abstraction is based on the observation that the protocol handles identifiers and parameters of messages in a simple way: the only operation performed is equality test and generation of new identifiers. Inspired by this observation, we construct an abstract model of the protocol, where the values of variables and message parameters are totally removed. Instead, a set of boolean variables is used to record whether variables and message parameters are equal or unequal to each other. Based on a choice of boolean variables, an abstract model of the protocol is constructed, which simulates the original one in the usual sense of being a safety-preserving abstraction (e.g., [CGL94,DGG97]).

Related Work The basic principles underlying the construction of an abstract models are understood from e.g., [CC77,CGL94,DGG97]). Recently, they have become used as a means to make a model of a concurrent system amenable to model checking. One approach is based on *predicate abstraction*, proposed by Graf and Saïdi [GS97], in which the state-space of the abstract model is defined by a set of boolean variables. Each boolean variable corresponds to a set of concrete states, i.e., it can be regarded as a predicate on the states of the concrete model. The transitions of the abstract model are constructed by proving theorems about the pre- and postconditions of the statements of the concrete model. Graf and Saïdi use the interactive theorem prover PVS for this purpose. Das et al. [DDP99] instead use a specialized automated theorem-prover. Other works in this area are, e.g., by Bensalem et al. [BLO98], by Colon and Uribe [CU98], by Lesens and Saïdi [LS97], and by Saïdi and Shankar [SS99]. These works have not explicitly considered unbounded channels in the way considered in this paper.

The idea of abstracting contents of unbounded channels by recording only relevant information is in some sense the basis for some symbolic approaches to model checking of lossy channel systems [AJ96], and Petri Nets [AČJYK96]. In some sense, we have used the idea of "data-independence" [Wol86,JP93]. A contribution is to do this in a setting with unbounded channels.

Outline In the next section, we give a brief outline of the five packet handshake protocol. A Promela model of the protocol can be found in Section 2. In Section 3, we discuss how the state-space of the protocol can be reduced by some

abstractions. We also try to provide some general principles underlying these abstractions. Section 4 reporst on the effect of performing a drastic abstraction on the protocol Section 5 contains conclusions and directions for future work.

2 A Five Packet Handshake Protocol

In this section, we describe the Five Packet Handshake Protocol, which is used in TCP for transmission of single messages. Descriptions of the protocol appear, e.g., in [Bel76] and [Lyn96, pp. 718–729].

The protocol is intended to transmit single messages from a sender to a receiver. Before each message transmission, a pair of initialization messages must be exchanged in order to establish an appropriate sequence number, which will be associated with the message in question. After the transmission of a message, its receipt must be appropriately acknowledged, again by a pair of messages. Thus, five messages are required for the transmission of a single message, hence the name of the protocol.

The protocol is intended to work in the presence of losses, duplications, reorderings, and arbitrary delays in the channel. Additionally, the sender and receiver may crash, and be forced to reinitialize. Under these liberal conditions, the protocol may sometimes lose a message, but never transmit duplicates. It is well-known [Lyn96, Thm 22.13] that no protocol can implement a perfect FIFO channel under these assumptions.

Let us give a more detailed description of the protocol. The protocol consists of a Sender, a Receiver, and a communication medium, which can reorder, lose, and duplicate messages. The Sender receives a stream of messages from the environment, and it is the task of the protocol to transmit these in order to the Receiver, which then forwards them to its environment. In order to recover from crashes, both the Sender and the Receiver maintain a set of *unique identifiers* (UIDs), taken from an infinite set, in stable memory (i.e., this set is not affected by crashes). The set represents the set of UIDs that have previously been used, and shall not be used again.

The transmission of a message consists of exchanging five packets.

1. The *Sender* sends a packet of form $needuid(v)$, in which v is a fresh UID. This is a request for a UID from the Receiver to be used for the message transmission.
2. The *Receiver* sends a packet of form $accept(u, v)$ where v is the UID received in the previous *needuid* message, and u is a fresh UID to be used for the message transmission
3. The *Sender* sends the message m in a packet of form $send(m, u)$, where u is the UID just received in the *accept* message
4. The *Receiver* acknowledges the message by an $ack(u)$ message
5. The *Sender* closes the packet exchange by a $cleanup(u)$ message, which also tells the Receiver to stop using the UID u in any future packet exchange.

In order to recover from packet losses, any packet can be retransmitted if the next expected packet does not arrive within some time. In the description in [Lyn96, pp. 718–729], there is a difference in retransmission policy between different packet types: packets of type *needuid*, *accept*, and *send* can be retransmitted an arbitrary number of times, whereas an *ack* packet is transmitted only on the receipt of a *send* packet; this is done even if the *send* packet is for an "outdated" UID. A *cleanup* packet can be sent in two situations.

- on the receipt of an *ack* packet,
- on the receipt of an "outdated" *accept* packet.

When a Sender or a Receiver crashes, they return to their initial state, but keep the record of used UIDs. In a crash, the Sender may lose some messages that were scheduled for transmission to the Receiver.

In the following, we give a naive simplified Promela model of the protocol, which is taken rather directly from the model of [Lyn96, pp. 718–729]. For readability, we have here not included some aspects of the model:

- We have not included our modeling of the imperfections in the channels (loss, reordering, duplication). These can be modeled in different ways: either by changing the Promela code that sends and/or receives messages, or by adding a demon process which scrambles the contents of Channels.
- In the analyzed model, each message reception and the following sequence of local operations of a process are included within atomic brackets. A standard rule of thumb (e.g., [Lam90,MP92] is to to enclose any sequence triggered by a receive, potentially containing a resulting send, in atomic brackets. In the version shown here, we have omitted the atomic brackets for readability.

Some modeling conventions must be made before the protocol description can be turned into a Promela Model:

- The sequence of messages to be transmitted from Sender to Receiver will be the sequence of numbers $0, 1, 2, \ldots$ upto a maximum number `MaxMsg`.
- The sequence of UIDs used will similarly be chosen as the sequence $0, 1, 2, \ldots$ upto a maximum number
- We use a separate channel for each packet type, e.g., the channel `Sendchan` to carry packets of form $send(m, u)$.

Following is the naive Promela model of the protocol.

```
#define NULL        0       /* Undefined value of lastUID */
#define MaxSeq      200     /* How many messages to check*/
#define ChanSize    5       /* channel size */

chan Needuidchan = [ChanSize] of { byte };
chan Acceptchan = [ChanSize] of { byte , byte };
chan Sendchan = [ChanSize] of { byte , byte };
chan Ackchan = [ChanSize] of { byte };
chan Cleanupchan = [ChanSize] of { byte };
```

```
active proctype Sender()
{   byte    SaccUID,       /* UID used to get new sequence number */
            SmsgUID,       /* UID used as sequence number */
            SnextMsg;      /* The message to be transmitted */
    byte    u,v;           /* Used to receive parameters of messages */

Sidle: SnextMsg < MaxSeq ->          /* get next message to send*/
                    SnextMsg++ ;
       SaccUID < MaxSeq ->           /* get fresh UID*/
                    SaccUID++ ;
Sneeduid: do
          :: Needuidchan! SaccUID    /* (re)transmit first packet */
          :: Acceptchan? u, v ->     /* on reception of accept message */
                if
                :: v == SaccUID ->   /* if correct uid start sending */
                     SmsgUID = u ; break
                :: else ->           /* otherwise send cleanup */
                     Cleanupchan ! u
                fi
          :: Ackchan? u ->           /* on a spurious ack */
                Cleanupchan ! u      /* reply with cleanup */
          :: goto Scrash             /* crash */
          od;

Ssend: do
       :: Sendchan!SnextMsg,SmsgUID   /* (re)transmit message */
       :: Ackchan? u ->               /* on reception of ack */
             if
             :: (u == SmsgUID) ->     /* if correct uid */
                    Cleanupchan!u;    /* send cleanup and restart */
                    goto Sidle
             :: else -> Cleanupchan!u /* otherwise send cleanup */
             fi
       :: Acceptchan? u,v ->          /* if spurious accept */
             if
             :: (u != SmsgUID) ->     /* if old, send cleanup */
                    Cleanupchan!u
             :: else -> skip          /* if current, do nothing */
             fi
      :: goto Scrash                  /* crash */
      od ;

Scrash:   do                          /* lose some input msgs */
          :: SnextMsg < MaxSeq -> SnextMsg++
          :: skip -> break
          od;
          goto Sidle
}
```

```
active proctype Receiver()
{   byte    RaccUID,     /* UID used to ge new sequence number */
            RmsgUID,     /* UID used as sequence number */
            RlastUID,    /* rembembers last sequence number */
            RexpMsg;     /* The message to be received */
    byte    m,u,v;       /* Used to receive parameters of messages */

Ridle: RmsgUID < MaxSeq -> RmsgUID++ ; /* get fresh sequence number */
       do
       :: Needuidchan? RaccUID ->      /* when needuid arrives */
                        break          /* start sending accept */
       :: Sendchan?m,u ->              /* spurious send */
             if                        /* if old uid arrives */
             :: (u != RlastUID) -> Ackchan!u   /* send ack */
             :: else -> skip
             fi
       :: Cleanupchan?u -> skip        /* ignore cleanup */
       :: goto Rcrash                  /* crash */
       od;

Raccept: do
         :: Acceptchan! RmsgUID , RaccUID  /* (re)transmit msg 2 */
         :: Sendchan ? m , u ->            /* on reception of send */
              if
              :: (u == RmsgUID) ->         /* if correct uid */
                    RlastUID = u;          /* remember uid */
                    assert(m >= RexpMsg);  /* check ordering */
                    RexpMsg = m+1; break   /* update expected Msg */
              :: (u != RmsgUID && u != RlastUID) ->  /* if old uid */
                                        Ackchan!u    /* send ack */
              :: else -> skip
              fi
         :: Needuidchan? v -> skip          /* ignore needuid */
         :: Cleanupchan? u ->
                if
                :: (u == RmsgUID) ->        /* on cleanup */
                      RlastUID = NULL;      /* clean RlastUID */
                      goto Ridle
                :: else -> skip
                fi
         :: goto Rcrash                     /* crash */
         od;
Rack: do
      :: Ackchan!RmsgUID                /* (re)transmit ack */
      :: Cleanupchan?u ->               /* when cleanup arrives */
                if
                :: (u == RlastUID) ->   /* if current uid */
                      RlastUID = NULL;  /* restart */
                      goto Ridle
                :: else -> skip         /* else skip */
```

```
                fi
        :: Sendchan ? m , u ->              /* spurious send msg */
                if
                :: (u != RlastUID) ->       /* if old uid */
                      Ackchan!u             /* send ack */
                :: else -> skip
                fi
        :: Needuidchan? v -> skip           /* ignore needuid */
        :: goto Rcrash                      /* crash */
        od;
Rcrash:   RlastUID=NULL;
          goto Ridle
}
```

3 Abstractions

A protocol model such as the one shown in the preceding section has far too many states for an exhaustive analysis to be feasible. We therefore present, in this subsection, how to obtain a rather drastic abstraction of the protocol. The basic idea is to represent only as much information as needed to infer the possible continued behavior of the protocol. The abstraction will abstract the state variables and channels of the (concrete) protocol model by a set of (abstract) boolean variables. The concrete and the abstract model are related by a concretization function γ, which maps sets of states of the abstract model to sets of states of the concrete model. For each boolean variable b, we can regard the predicates b and $\neg b$ as sets of abstract states (the sets of states where b, resp. $\neg b$, is true). These sets correspond to the sets $\gamma(b)$ and $\gamma(\neg b)$ of concrete states. Note that, unlike some other approaches to predicate abstraction (e.g., [GS97,DDP99], the sets $\gamma(b)$ and $\gamma(\neg b)$ need not be disjoint. We do this for convenience, in the hope of getting a smaller abstract model. Having defined $\gamma(b)$ and $\gamma(\neg b)$ for all boolean variables, the function γ is extended to arbitrary sets in the natural way by

$$\begin{aligned} \gamma(\phi \ \wedge \ \phi') &\equiv \gamma(\phi) \ \wedge \ \gamma(\phi') \\ \gamma(\phi \ \vee \ \phi') &\equiv \gamma(\phi) \ \vee \ \gamma(\phi') \end{aligned} \quad .$$

The control states of the abstract model are identical to the control states of the concrete model.

We note that the only operations on the variables of the protocol are check for equality, and assignment of a fresh value to a variable. For the variables `SnextMsg` and `RnexpMsg` that model the transmitted messages, this is not quite true, but we can make it true by not modeling them as integers. Instead, we let each new message, which is to be transmitted by the Sender, be a fresh value. When receiving a message, we let the receiver check that the received message is equal to the message that the Sender is trying to send. We use a boolean flag to check for duplicate receptions.

We construct an abstraction for variables that are used in this way by including, for each pair x, x' of state variables of the protocol, an abstract boolean

variable, named x_eq_x', which is true if and only if $x = x'$. This means that

$$\gamma(x\_\mathrm{eq}\_x') \equiv x = x' \qquad \text{and} \qquad \gamma(\neg\ x\_\mathrm{eq}\_x') \equiv x \neq x' \qquad .$$

The abstraction of channel contents is slightly more complicated. Intuitively, the contents of a channel may influence the future behavior of the protocol by containing messages. The potential influence of receiving a message can be determined by checking whether its parameters are equal or unequal to state variables of the protocol, and also to parameters of other messages, possibly in other channels.

Based on this idea, we construct an abstraction of channel contents using boolean variables as follows. We regard each message as a tuple of parameters. For instance, the messages in channels `Needuidchan`, `Ackchan`, and `Cleanupchan` are tuples with one elements, and the messages in channels `Acceptchan` and `Sendchan` are tuples with two elements. An abstract variable b will record whether some channel c contains a message, whose elements satisfy some equality constraint over its elements. An *equality constraint* over a set of parameters is a conjunction of formulas of form $v = v'$ or $v \neq v'$, where v, v' are either program variables or parameters of the message. We use the name c_mc_ψ for the boolean variable, which records whether the channel c *may contain* a message $\langle u_1, \ldots, u_n \rangle$ which satisfies the equality constraint ψ over the parameters $u_1, \ldots, u_n$. Note the formulation *may contain* (abbreviated to `mc` in the variable name): since any message in a channel can be lost arbitrarily, we can only be sure that a certain message is *not* in the channel (i.e., if it was never sent). Formally, this is reflected by defining the concretization of c_mc_ψ, as follows.

$$\begin{aligned} \gamma(c\_\mathrm{mc}\_\psi) &\equiv \mathit{true} \\ \gamma(\neg\ c\_\mathrm{mc}\_\psi) &\equiv \neg\exists \langle u_1, \ldots, u_n \rangle \in c\ .\ \psi \end{aligned} \qquad .$$

Note that $\gamma(c\_\mathrm{mc}\_\psi)$ and $\gamma(\neg\ c\_\mathrm{mc}\_\psi)$ overlap. We do this in the hope of creating a small abstract model. Namely, if we would have defined $\gamma(c\_\mathrm{mc}\_\psi)$ as the negation of $\gamma(\neg\ c\_\mathrm{mc}\_\psi)$, then for any reachable abstract state where $\gamma(c\_\mathrm{mc}\_\psi)$ is true, we would also have to include the corresponding abstract state where $\gamma(c\_\mathrm{mc}\_\psi)$ is false, since the channel can always lose messages. As a further slight optimization, we require that at least one conjunct of ψ is an equality: if all conjuncts in ψ are inequalities, then any strange or outdated message will make the variable $\gamma(c\_\mathrm{mc}\_\psi)$ true in the abstract model, meaning that the variable is uninteresting.

As an example, the abstract variable `Needuidchan_mc_`$(u =$ `SaccUID`$)$ (which in the Promela code will be written `Needuidchan_mc_SaccUID`) records whether the channel `Needuidchan` may contain a message $\langle u \rangle$ such that u is equal to `SaccUID`.

In principle, the abstraction could contain a boolean variable x_eq_x' for each pair x, x' of program variables, and a variable c_mc_ψ for each channel c and corresponding equality constraint ψ. Many of these will turn out to be dead variables in a static analysis of the abstract model, and so are not necessary.

Having chosen a set of abstract variables, with corresponding meanings, we should now construct an abstract Promela model, which simulates the concrete protocol model. This means that if the concrete model can make an atomic step from control point p to p' while changing the state of variables and channels from s to s', then for each state t of the abstract variables such that $\gamma(t) = s$, there should be a state t' of the abstract variables such that $\gamma(t') = s'$ and such that the abstract model can make a step from control point p to p' while changing the state of variables t to t'. All safety properties of the abstract model will then also be satisfied by the concrete model [CGL94,DGG97]). Below, we make one suggestion for how this can be carried out.

We regard the concrete Promela model as consisting of a collection of guarded commands. For instance, an atomic statement of form

$$\begin{array}{l}\texttt{Somechan?}u1, u2\ \texttt{->}\ \texttt{if}\\ \qquad\qquad\qquad\qquad\quad \texttt{::}\ guard_1\ \texttt{->}\ stmts_1\\ \qquad\qquad\qquad\qquad\qquad\quad \cdots\\ \qquad\qquad\qquad\qquad\quad \texttt{::}\ guard_n\ \texttt{->}\ stmts_n\\ \qquad\qquad\qquad\qquad\quad \texttt{fi}\end{array}$$

is regarded as consisting of a set of atomic guarded commands of form

$$GC_i \quad \equiv \quad \texttt{Somechan?}u1, u2\ \texttt{->}\ guard_i\ \texttt{->}\ stmts_i$$

for $i = 1, \ldots, n$ (We assume that the statement can not deadlock after reception of a message from `Somechan`).

For each GC_i, we construct a set of abstract statements, which are guarded by an expression g over abstract variables such that the set $\gamma(g)$ includes the set $\exists\langle u1, u2\rangle \in \texttt{Somechan} \wedge guard_i$ corresponding to the guard of GC_i. Furthermore, the abstract statements must update all abstract variables that may be affected by the assignments and send statements in $stmts_i$. The updates to each such abstract variable b is guided by a postcondition $sp(GC_i, true)$ of GC_i with respect to the predicate *true*. More precisely, b is assigned to `TRUE` under a condition *truecond*, such that $\gamma(truecond)$ is implied by the conjunction of $sp(GC_i, true)$ and $\gamma(b)$. Analogously, b is assigned to `FALSE` under a condition *falsecond*, such that $\gamma(falsecond)$ is implied by the conjunction of $sp(GC_i, true)$ and $\gamma(\neg b)$. Alternatively, abstract variables of form x_eq_x' can be updated by an expression *exp* such that such that $x = x' \Leftrightarrow \gamma(exp)$ is implied by $sp(GC_i, true)$. Note that the postcondition may refer to the values of variables before the statement.

Let us consider some examples. The statement

```
Ackchan? u ->
       if
       :: (u == SmsgUID) -> Cleanupchan!u;  goto Sidle
       :: else -> Cleanupchan!u
       fi
```

which occurs within atomic brackets, is regarded as two guarded commands. The first one,

```
Ackchan? u -> (u == SmsgUID) -> Cleanupchan!u;  goto Sidle
```

has the guard

$$\exists \langle u \rangle \in \texttt{Ackchan} \,.\, u = \texttt{SmsgUID}$$

and the postcondition (disregarding the `goto`)

$$\exists \langle u \rangle \in \texttt{Ackchan}^- \,.\, u = \texttt{SmsgUID} \;\wedge\; \texttt{Cleanupchan} = \texttt{Cleanupchan}^- \cup \langle \texttt{SmsgUID} \rangle$$

where we use v^- to denote the value of v before the statement. The guard can be abstraced by the predicate Ackchan_mc_$(u = \texttt{SmsgUID})$, which in the code is given the name `Ackchan_mc_SmsgUID`. Since the channel `Cleanupchan` is updated, the abstract statement must update variables of form Cleanupchan_mc_ψ. For instance, the variable Cleanupchan_mc_$(u = \texttt{RmsgUID})$ will be true after the statement, if either it was true before the statement, or if `RmsgUID` is equal to `SmsgUID`. The same holds for a variable of form Cleanupchan_mc_$(u = \texttt{RlastUID})$. Summarizing, the corresponding abstract statement will have the form

```
Ackchan_mc_SmsgUID ->
   Cleanupchan_mc_RmsgUID = (Cleanupchan_mc_RmsgUID || RmsgUID_eq_SmsgUID);
   Cleanupchan_mc_RlastUID = (Cleanupchan_mc_RlastUID || RlastUID_eq_SmsgU);
   goto Sidle
```

Similarly, the second statement, which can be written

```
Ackchan? u -> (u != SmsgUID) -> Cleanupchan!u
```

can be abstracted to

```
Cleanupchan_mc_RmsgUID =
 (Cleanupchan_mc_RmsgUID || (Ackchan_mc_RmsgUID && ! SmsgUID_eq_RmsgUID));
Cleanupchan_mc_RlastUID =
 (Cleanupchan_mc_RlastUID || (Ackchan_mc_RlastUID && ! SmsgUID_eq_RlastUID))
```

Note here that we have transformed the control structure by omitting the guard, which would be abstracted by Ackchan_mc_$(u \neq \texttt{SmsgUID})$. Such a guard will probably be true most of the time, so we omit it.

4 Abstraction of the Protocol

In this section, we report on our manual application of the abstraction technique of the previous section to the Five Packet Handshake Protocol. In the abstract protocol model, boolean variables are chosen as described in the previous section. A (manual) dependency analysis is used to avoid including abstract variables that do not affect (directly or indirectly) the control flow of the protocol. The code of the abstract model is generated according to the principles in the preceding section. Some things to mention are the following.

- The set of messages, which in the concrete model are represented by integers, and assigned to variables `SnextMsg` and `RexpMsg`, is treated in the same way as UIDs. In order to do that, we must first change the mechanism for checking that messages are not duplicated by the protocol. In the model of Section 2, this mechanism relies on the fact that messages are generated as increasing integers. We change this mechanism as follows. When a message is received, it is checked that it is equal to the message `SnextMsg` that the sender is trying to send. In addition, a boolean flag `received_SnextMsg` records whether this message has been received previously. The flag is reset when a new message is considered for transmission by the sender.
- Since the boolean variables for comparisons are not local to any process, we make all variables global.
- Names of variables are chosen to reflect their meaning: for instance, `RaccUID_eq_SaccUID` is *true* iff the variables `RaccUID` and `SaccUID` have the same value. The variable `Accchan_mc_neg_RmsgUID_and_SaccUID` is *true* if the channel `Acceptchan` may contain a message `u,v`, where the value of `u` is different from that of `RmsgUID` and `v` is equal to `SaccUID`.
- Most of the abstract statements have been constructed in a rather uniform way. One statement, which maybe was not so straightforward, is the abstraction of

  ```
  Acceptchan? u, v -> v == SaccUID -> SmsgUID = u ; break
  ```

  which is the normal reception of an *accept* message by the Sender, at control point `Sneeduid`. One problem concerned the update of `Ackchan_mc_SmsgUID`, which records whether the channel `Ackhan` may contain the message `SmsgUID`. The problem is that the above statement assigns a value of `SmsgUID`, about which it gives no information. A safe abstraction would be to set `Ackchan_mc_SmsgUID` to true. This abstraction is sufficient to prove absense of duplication, but cannot prove that the protocol delivers all messages in the absense of crashes: the reason is that the abstract statement `Ackchan_mc_SmsgUID = TRUE` automatically "inserts" an acknowledgment into the channel `Ackchan`.
  A better solution is to guard the statement `Ackchan_mc_SmsgUID = TRUE` by some checks. For instance, if the channel `Acceptchan` does not contain any message $\langle$`u,v`$\rangle$ where u is different from `RmsgUID` and v is equal to `SmsgUID`, then the statement can be guarded by `Ackchan_mc_RmsgUID = TRUE`, since `u` must then be equal to `RmsgUID`. We can make a similar check for `RlastUID`. The result is the abstract statement

  ```
  Ackchan_mc_SmsgUID =
      (   (Accchan_mc_neg_RmsgUID_and_SaccUID
           && Accchan_mc_neg_RlastUID_and_SaccUID
          )
       || (Accchan_mc_RmsgUID_and_SaccUID && Ackchan_mc_RmsgUID)
       || (Accchan_mc_RlastUID_and_SaccUID && Ackchan_mc_RlastUID)
      ) ;
  ```

A version of the abstract protocol model, without comments, is given in the appendix.

Analysis of Abstract Model The abstract Protocol model shown above has a state space defined by control points and 20 boolean variables. The number of reachable states, as reported by SPIN, is 575. SPIN was used to analyze that the protocol delivers messages without duplication. On the other hand, the protocol can lose messages. If the possibility of crashes is excluded, then the number of reachable states becomes only 46, and SPIN could check that no messages are lost. This check was conducted using the boolean flag `received_SnextMsg`.

5 Discussion and Conclusion

We have reported on the modeling and verification in SPIN of a protocol for single message transmission in TCP. A naive model of the protocol has far too many states for making an exhaustive analysis of the state-space possible. This situation is typical for any protocol which uses asynchronous message passing, where there are more than a couple of different message types.

The main part of the paper presents some techniques for reducing the state space by abstracting away the channels and data variables with unbounded ranges. In the particular case of this protocol, we could replace variables and channels by equality constraints, which were adapted to expressing properties of messages in channels. A contribution is to do this in a setting with unbounded channels. Since we cannot perform symbolic manipulations in SPIN, we have to represent the equality constraints in some way: we have chosen to represent each relevant equality constraint by a boolean variable. Other representations could have been possible, such as choosing a small finite domain where values are reused in an appropriate way (as done, e.g., in [JP93]), but we guess that such a representation would be less efficient in the presence of message channels.

In this work, we have produced the abstraction manually. This introduces a substantial risk for errors, and it would, of course, be desirable to perform the abstractions automatically. The abstractions of most statements were rather straight-forward, but a few required some care in order to generate a sufficiently detailed abstract model.

One motivation for the work is our experience that, as part of teaching formal methods, and protocol validation, it is important to teach good techniques for producing compact and manageable protocol models. In order not to lose the essence of formality, it is important that students understand how to make "correct" simplifications of a protocol, by means of abstractions (preferably safe abstractions), protocol transformations, etc. The principles for making abstractions are known, but a limited number of examples of their practical use have been published in the literature, in a way that can be read by students. With time, a well-documented library of safe abstractions should be developed.

Acknowledgments We are grateful to the reviewers for insightful comments, which helped improve the paper considerably.

References

[AČJYK96] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc.* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[Bel76] D. Belsnes. Single-message communication. *IEEE Trans. on Computers*, COM-24(2):190–194, Feb. 1976.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 1998.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc.* 4^{th} *ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[CGL94] E. M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5), Sept. 1994.

[CU98] M.A. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc.* 10^{th} *Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer Verlag, 1998.

[DDP99] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, 1999.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2), 1997.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc.* 9^{th} *Int. Conf. on Computer Aided Verification*, volume 1254, Haifa, Israel, 1997. Springer Verlag.

[JP93] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2):272–302, Dec. 1993.

[Lam90] L. Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4(2):59–68, 1990.

[LS97] D. Lesens and H. Saidi. Abstraction of parameterized networks. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1992.

[SS99] H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, 1999.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc.* 13^{th} *ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.

A Promela Code of the Abstract Model

In this appendix, we give a complete Promela listing of an abstract model of the protocol. The abstraction was performed by hand. In order to fit the code on paper, names of variables have been abbreviated in comparison with their description in the paper.

For instance, the variable Acccchan_mc_RmsgUID_and_neg_SaccUID, which records whether the channel Acceptchan may contain a message $\langle$RmsgUID, $v\rangle$ such that v is different from the value of SaccUID is written as Acc_mc_RmsgID_a_n_SaccID.

```
#define FALSE       0
#define TRUE        1

bit RaccID_eq_SaccID;
bit RmsgID_eq_SmsgID, RlastID_eq_SmsgID, RmsgID_eq_RlastID;
bit      Nu_mc_SaccID,

         Acc_mc_RmsgID_a_SaccID,
         Acc_mc_RmsgID_a_n_SaccID,
         Acc_mc_n_RmsgID_a_SaccID,
         Acc_mc_RlastID_a_SaccID,
         Acc_mc_RlastID_a_n_SaccID,
         Acc_mc_n_RlastID_a_SaccID,

         Send_mc_SmsgID,
         Send_mc_SnextMsg_a_RmsgID,
         Send_mc_n_SnextMsg_a_RmsgID,

         Ack_mc_SmsgID,
         Ack_mc_RmsgID,
         Ack_mc_RlastID,

         Cu_mc_RmsgID,
         Cu_mc_RlastID;

bit Received_SnextMsg = TRUE;

active proctype Sender()
{

Sidle: atomic{
       Send_mc_SnextMsg_a_RmsgID = FALSE;
       Received_SnextMsg = FALSE;

       RaccID_eq_SaccID = FALSE;
       Nu_mc_SaccID = FALSE;
       Acc_mc_RmsgID_a_SaccID = FALSE;
       Acc_mc_n_RmsgID_a_SaccID = FALSE;
       Acc_mc_RlastID_a_SaccID = FALSE;
       Acc_mc_n_RlastID_a_SaccID = FALSE
```

```
       };

SneedID: do
      :: Nu_mc_SaccID = TRUE     /* (re)transmit the needID message */
      :: atomic{(Acc_mc_RmsgID_a_SaccID || Acc_mc_n_RmsgID_a_SaccID) ->
           Ack_mc_SmsgID =
            (   (Acc_mc_n_RmsgID_a_SaccID && Acc_mc_n_RlastID_a_SaccID)
             || (Acc_mc_RmsgID_a_SaccID && Ack_mc_RmsgID)
             || (Acc_mc_RlastID_a_SaccID && Ack_mc_RlastID)
            ) ;
           if :: Acc_mc_RmsgID_a_SaccID -> RmsgID_eq_SmsgID = TRUE
              :: Acc_mc_n_RmsgID_a_SaccID -> RmsgID_eq_SmsgID = FALSE
           fi;
           if :: Acc_mc_RlastID_a_SaccID -> RlastID_eq_SmsgID = TRUE
              :: Acc_mc_n_RlastID_a_SaccID -> RlastID_eq_SmsgID = FALSE
           fi;
           break
         }
      :: atomic{Cu_mc_RmsgID = (Cu_mc_RmsgID || Acc_mc_RmsgID_a_n_SaccID);
          Cu_mc_RlastID = (Cu_mc_RlastID || Acc_mc_RlastID_a_n_SaccID)
         }
      :: atomic{Cu_mc_RmsgID = (Cu_mc_RmsgID || Ack_mc_RmsgID);
           Cu_mc_RlastID = (Cu_mc_RlastID || Ack_mc_RlastID)
         }
       :: goto Scrash
      od;

Ssend: do
      :: atomic{Send_mc_SnextMsg_a_RmsgID =
                   (Send_mc_SnextMsg_a_RmsgID || RmsgID_eq_SmsgID);
           Send_mc_SmsgID = TRUE
         }
      :: atomic{Ack_mc_SmsgID ->
              Cu_mc_RmsgID =  (Cu_mc_RmsgID || RmsgID_eq_SmsgID);
              Cu_mc_RlastID =  (Cu_mc_RlastID || RlastID_eq_SmsgID);
              goto Sidle
         }
      :: atomic{Cu_mc_RmsgID =
              (Cu_mc_RmsgID || (Ack_mc_RmsgID && ! RmsgID_eq_SmsgID));
          Cu_mc_RlastID =
             (Cu_mc_RlastID || (Ack_mc_RlastID && ! RlastID_eq_SmsgID))
         }
      :: atomic{Cu_mc_RmsgID =
            (    Cu_mc_RmsgID
             || ((Acc_mc_RmsgID_a_SaccID ||Acc_mc_RmsgID_a_n_SaccID)
                 && ! RmsgID_eq_SmsgID));
           Cu_mc_RlastID =
             (    Cu_mc_RlastID
              || ((Acc_mc_RlastID_a_SaccID ||Acc_mc_RlastID_a_n_SaccID)
                  && ! RlastID_eq_SmsgID));
```

```
         }
        :: goto Scrash
      od ;

Scrash: goto Sidle
}

active proctype Receiver()
{

Ridle: atomic{
       RmsgID_eq_SmsgID = FALSE;
       RmsgID_eq_RlastID = FALSE;
       Acc_mc_RmsgID_a_SaccID = FALSE;
       Acc_mc_RmsgID_a_n_SaccID = FALSE;
       Send_mc_SnextMsg_a_RmsgID = FALSE;
       Send_mc_n_SnextMsg_a_RmsgID = FALSE;
       Ack_mc_RmsgID = FALSE;
       Cu_mc_RmsgID = FALSE};

      do
      :: atomic{Nu_mc_SaccID -> RaccID_eq_SaccID = TRUE; break}
      :: atomic{TRUE -> RaccID_eq_SaccID = FALSE; break}
      :: atomic{Ack_mc_SmsgID =
             (Ack_mc_SmsgID || (Send_mc_SmsgID && ! RlastID_eq_SmsgID));
           Ack_mc_RmsgID =
             (   Ack_mc_RmsgID
              || ((Send_mc_SnextMsg_a_RmsgID ||Send_mc_SnextMsg_a_RmsgID)
                   && ! RmsgID_eq_RlastID))
         }
        :: goto Rcrash
      od;

Raccept:
      do
      :: atomic{Acc_mc_RmsgID_a_SaccID =
                    (Acc_mc_RmsgID_a_SaccID || RaccID_eq_SaccID);
                Acc_mc_RmsgID_a_n_SaccID =
                    (Acc_mc_RmsgID_a_n_SaccID || ! RaccID_eq_SaccID);
                Acc_mc_RlastID_a_SaccID =
                     (Acc_mc_RlastID_a_SaccID
                      || (RmsgID_eq_RlastID && RaccID_eq_SaccID));
                Acc_mc_RlastID_a_n_SaccID =
                     (Acc_mc_RlastID_a_n_SaccID
                      || (RmsgID_eq_RlastID && ! RaccID_eq_SaccID));
                Acc_mc_n_RlastID_a_SaccID =
                     (Acc_mc_n_RlastID_a_SaccID
                      || (! RmsgID_eq_RlastID && RaccID_eq_SaccID))
        }
      :: assert (   !(Send_mc_n_SnextMsg_a_RmsgID
```

```
                || (Send_mc_SnextMsg_a_RmsgID && Received_SnextMsg)))
        :: atomic{(Send_mc_SnextMsg_a_RmsgID ||Send_mc_n_SnextMsg_a_RmsgID)
                  ->  RmsgID_eq_RlastID = TRUE;
                      RlastID_eq_SmsgID = RmsgID_eq_SmsgID;
                      Acc_mc_RlastID_a_SaccID = Acc_mc_RmsgID_a_SaccID;
                      Acc_mc_RlastID_a_n_SaccID = Acc_mc_RmsgID_a_n_SaccID;
                      Acc_mc_n_RlastID_a_SaccID = Acc_mc_n_RmsgID_a_SaccID;
                      Cu_mc_RlastID = Cu_mc_RmsgID;
                      Received_SnextMsg = TRUE;
                      break
            }
        :: atomic{(Send_mc_SmsgID && ! RmsgID_eq_SmsgID && ! RlastID_eq_SmsgID)
                       ->  Ack_mc_SmsgID = TRUE
           }
        :: atomic{Cu_mc_RmsgID ->
                  RmsgID_eq_RlastID = FALSE;
                  RlastID_eq_SmsgID = FALSE;
                  Acc_mc_n_RlastID_a_SaccID =
                        (Acc_mc_n_RlastID_a_SaccID || Acc_mc_RlastID_a_SaccID);
                  Acc_mc_RlastID_a_SaccID = FALSE;
                  Acc_mc_RlastID_a_n_SaccID = FALSE;
                  Ack_mc_RlastID = FALSE;
                  Cu_mc_RlastID = FALSE;
                  goto Ridle
            }
          :: goto Rcrash
        od;
Rack:
        do
        :: atomic{Ack_mc_RmsgID = TRUE;
                  Ack_mc_SmsgID = (Ack_mc_SmsgID || RmsgID_eq_SmsgID);
                  Ack_mc_RlastID = (Ack_mc_RlastID || RmsgID_eq_RlastID)
           }
        :: atomic{Cu_mc_RlastID ->
                  RmsgID_eq_RlastID = FALSE;
                  RlastID_eq_SmsgID = FALSE;
                  Acc_mc_n_RlastID_a_SaccID =
                        (Acc_mc_n_RlastID_a_SaccID || Acc_mc_RlastID_a_SaccID);
                  Acc_mc_RlastID_a_SaccID = FALSE;
                  Acc_mc_RlastID_a_n_SaccID = FALSE;
                  Ack_mc_RlastID = FALSE;
                  Cu_mc_RlastID = FALSE;
                  goto Ridle
            }
        :: atomic{
                Ack_mc_SmsgID =
                    (Ack_mc_SmsgID || (Send_mc_SmsgID && ! RlastID_eq_SmsgID));
                Ack_mc_RmsgID =
                    (   Ack_mc_RmsgID
                     || ((Send_mc_SnextMsg_a_RmsgID ||Send_mc_SnextMsg_a_RmsgID)
```

```
                        && ! RmsgID_eq_RlastID))
         }
        :: goto Rcrash
      od;
Rcrash: atomic{RmsgID_eq_RlastID = FALSE;
               RlastID_eq_SmsgID = FALSE;
               Acc_mc_n_RlastID_a_SaccID =
                  (Acc_mc_n_RlastID_a_SaccID || Acc_mc_RlastID_a_SaccID);
               Acc_mc_RlastID_a_SaccID = FALSE;
               Acc_mc_RlastID_a_n_SaccID = FALSE;
               Ack_mc_RlastID = FALSE;
               Cu_mc_RlastID = FALSE;
               goto Ridle
        }
}
```