

Fighting Livelock in the GNU i-Protocol: A Case Study in Explicit-State Model Checking

Yifei Dong*, Xiaoqun Du**, Gerard J. Holzmann***, Scott A. Smolka*

*Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400, USA

**Cadence Design Systems, Inc., 35 Spring Street, New Providence, NJ 07974, USA

***Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

Abstract. The i-protocol, an optimized sliding-window protocol for GNU uucp, first came to our attention in 1995 when we used the Concurrency Factory’s local model checker to detect, locate, and correct a non-trivial livelock in version 1.04 of the protocol. Since then, we have conducted a systematic case study on the protocol using four verification tools, *viz.* *Cospan*, *Mur φ* , *Spin*, and *XMC*, each of which supports some form of explicit-state model checking. Our results show that although the i-protocol is inherently complex—the size of its state space grows exponentially in the window size and it deploys several sophisticated optimizations aimed at minimizing control-message and retransmission overhead—it is nonetheless amenable to a number of general-purpose abstraction techniques whose application can significantly reduce the size of the protocol’s state space.

Key words: explicit-state model checking – livelock – protocol verification – sliding-window protocol

1 Introduction

Model checking [EC81,QS82,VW86,CES86] is a verification technique based on exhaustive state-space analysis, aimed at determining whether a system specification possesses a property expressed as a temporal-logic formula. Model checking has enjoyed considerable success in verifying, or finding design errors in, real-life systems. Accounts of some of these success stories can be found in e.g., [CW96] and [Hol97].

In this paper, we report on our experience in using model checking—as provided by four widely used verification tools—to detect and correct a non-trivial livelock

in a bidirectional sliding-window protocol. The tools we used are *Cospan* [HHK96], *Mur φ* [Di196], *Spin* [Hol97], and *XMC* [RRR⁺97], each of which supports some variety of explicit-state model checking. As the name implies, *explicit-state* model checkers store and manipulate states explicitly, using a data structure such as a hash table (one table entry per state). *Symbolic* model checkers, on the other hand, deploy data structures such as binary decision diagrams (BDDs) to represent *sets* of states; see, e.g., [McM93]. In practice, explicit-state model checkers tend to perform better than symbolic ones on protocols, such as those for cache-coherence and communications, while the reverse is true for hardware designs [HD93].

The protocol that we investigate, the i-protocol, is part of the GNU uucp package available from the Free Software Foundation, and is used for file transfers over serial lines. The i-protocol sits on the uucp protocol stack; its purpose is to ensure ordered reliable duplex communication between two sites. At its lower interface, the i-protocol assumes unreliable (lossy) packet-based FIFO connectivity. To its upper interface, it provides *reliable* packet-based FIFO service. A distinguishing feature of the i-protocol is the sophisticated manner in which it attempts to minimize control-message and retransmission overhead. The GNU uucp package also contains the g- and j-protocols, which are variants of the i-protocol.

A problem with the i-protocol, GNU uucp version 1.04, was first noticed by Gene Stark¹ while trying to transfer large files from a remote computer to his home PC over a modem line. In particular, it appeared that, under certain message-loss conditions, the protocol would enter a “confused” state and eventually drop the connection. In order to diagnose this problem, we extracted an abstract version of the i-protocol from its source code, consisting of approximately 1500 lines of C code. We formalized

* Dong and Smolka’s research supported in part by NSF Grants CCR-9988155 and CCR-9705998; ARO grants DAAD190110003 and DAAD190110019; and ONR grant N000140110967.

¹ Eugene Stark is a Professor of Computer Science at SUNY–Stony Brook.

this abstraction of the protocol in VPL (Value Passing Language), the input language of the Concurrency Factory specification and verification tool set [CLSS96].

The VPL source of the i-protocol was then subjected to a series of model-checking experiments using the Concurrency Factory’s local model checker for the modal mu-calculus [RS97]. This led us to the root of the problem: a livelock that occurs when a particular series of message losses drives the protocol into a state where the communicating parties enter into a cycle of fruitless message exchanges without any packets being delivered to the upper-layer entities. Seeing no progress, the two sides close the connection, which must then be re-established. If the communication line is sufficiently noisy, or if one of the sides is slow in emptying communication buffers, say due to disk waits leading to buffer overflows, the chances of this scenario recurring are high, and can result in extremely poor performance.

Using the Concurrency Factory’s diagnostic facility, we were able to pinpoint and subsequently “patch” the bug in the VPL code. The fix to the protocol consists of a simple change in the way negative acknowledgments are handled. The livelock error was fixed independently by Ian Taylor, the i-protocol’s original developer, in GNU uucp version 1.05.

Since applying the Concurrency Factory to the problem, we have conducted a systematic case study of the i-protocol using the *Cospan*, *Mur ϕ* , *Spin*, and *XMC* verification tools. The i-protocol makes for a particularly interesting case study in protocol verification for several reasons. First, the version we originally model checked has a bug, i.e. the livelock error, and hence the protocol can be used to gauge a tool’s ability to uncover errors of this nature. In this case, we are more interested in debugging or refutation than in verification.

Secondly, the size of the i-protocol’s state space grows exponentially in the window size, and the entirety of this state space must be considered to verify that the protocol, with the livelock error eliminated, is deadlock- or livelock-free. Also, the i-protocol is an asynchronous, low-level software system equipped with a number of optimizations aimed at minimizing control-message and retransmission overhead. These optimizations further add to the protocol’s complexity.

Thirdly, because of the i-protocol’s inherent complexity, a novice tool user would immediately encounter difficulties in trying to analyze the protocol, due to the fact that the size of a system’s state space is in general exponential in the size of the system’s specification. This phenomenon is referred to as *state-space explosion*.

It is therefore imperative that certain modeling guidelines be followed when specifying the i-protocol in the input language of a model checker, to limit the effects of state-space explosion. Such guidelines are usually tool-specific and require a detailed knowledge of the tool’s modeling language to be able to deploy effectively. An informed choice of tool run-time options is also essen-

tial. Similarly, the results of our case study show that state-space explosion can be further curtailed by applying certain general-purpose abstraction techniques, several of which are identified in [Hol98].

Indeed, the main contribution of this paper is to identify the modeling guidelines, run-time options, and abstractions that allowed us to effectively and efficiently model check the i-protocol, and to present the supporting tool performance data. Moreover, we believe that many of these techniques will be applicable to a broad class of protocols.

In related work, Chamillard *et al.* [CCA96] and Corbett [Cor96] have applied a variety of model checkers to a benchmarking suite of Ada tasking programs, so that the performance of these model checkers could be compared on reachability properties. In contrast, the main purpose of our study is to identify both tool-specific and tool-general techniques that can be deployed to achieve optimum performance of model checkers on real-life protocols. Also, we are primarily interested in livelock properties in this case study as opposed to reachability ones.

The remainder of the paper develops along the following lines. Section 2 gives a detailed account of the i-protocol, with an emphasis on how we modeled the protocol for verification purposes. Section 3 discusses the livelock that we discovered, and shows how a small change to the protocol effectively eliminates this form of livelock. Section 4 describes the salient features of the tools used in the case study. Section 5 describes the abstractions we applied to the i-protocol specification. Section 6 summarizes the results of our model-checking experiments. Section 7 contains our concluding remarks.

We have constructed a web site [D⁺00] for the i-protocol case study where one can find the C source code of versions 1.04 and 1.06 of the i-protocol, the original VPL encoding, and the various specifications of the protocol (18 in total) analyzed in this paper. The results of this paper build upon those presented in [Hol99] and improve upon and supersede the results reported in [DDR⁺99].

2 Modeling the i-Protocol

In this section we introduce the i-protocol, and describe how we modeled it for verification purposes.

The i-protocol is a sliding-window protocol, but with some optimizations, to be described later, aimed at reducing the acknowledgment and retransmission traffic. The window size, along with other “steady-state” protocol parameters such as data-packet size, line-quality and error-handling parameters, timeout values, acknowledgment high watermarks, and data and message buffer sizes, are decided at the parameter-negotiation stage of connection set-up. Since we are concerned with data-transfer properties, we do not explicitly model connection set-up, parameter-negotiation, error and line-quality

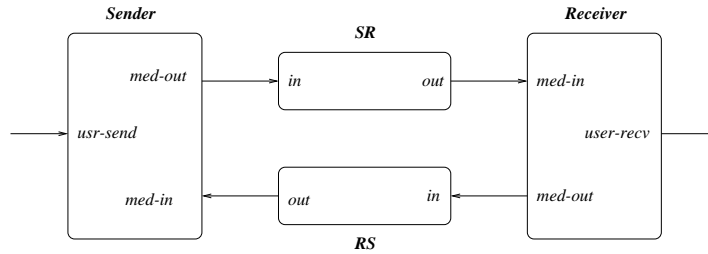


Fig. 1: Simplified structure of the i-protocol. *SR* and *RS* represent the media between the sender and the receiver.

monitoring, and connection shutdown. In particular, the window size for our model is a parameter that is fixed at “compile time.”

The protocol is intended to provide reliable, full duplex, FIFO service to its upper interface, given a full duplex, unreliable, FIFO packet-based communication service by its lower interface. It is convenient to imagine each side of the protocol as consisting of two halves: a sender half that sends data packets to, and receives acknowledgments from, the receiver half on the other side, and a receiver half that receives data packets from, and sends acknowledgments to, the sender half on the other side. The structure of the protocol is illustrated in Figure 1. To simplify the presentation, the figure depicts the version consisting of only the sender half of the protocol on one side and the receiver half on the other.

To allow for communication latency, the sender can send several packets without waiting for acknowledgments. If the window size is W , then the sender can have up to W contiguous packets unacknowledged at any time. These packets are stamped with sequence numbers when received from the upper layer; sequence numbers range from 0 to $SEQ - 1$. The i-protocol, as implemented in GNU ucp, uses a fixed value of $SEQ = 32$, and is intended for window sizes up to, but not exceeding, 16.

To cut down on the acknowledgment traffic, the receiver can piggyback its acknowledgments on top of normal data, or other control traffic. When both sides are exchanging data packets, this is often sufficient to keep the connection going without the need for explicit acknowledgments. However, when a side is only receiving data, it needs to send explicit ACKs. In this case, as an optimization, ACKs are sent only at half-window boundaries, i.e., one for every $\lceil W/2 \rceil$ packets received.

The sender half uses the following main state variables, each of which ranges over $0..SEQ - 1$. A variable *sendseq* is used to stamp the next user-level message from the upper layer. Its value gives the upper edge (exclusive) of the sender’s “active window.” The variable *rack* is used to keep track of acknowledgments from the remote, and its value gives the lower edge (exclusive) of the sender’s active window. At our level of abstraction, the data contents of a packet are not modeled, and so the sender does not explicitly buffer unsent messages.²

The main data structures used by the receiver half are as follows. A variable *recseq* is used to record the sequence number up to and including which all packets have been successfully received from the remote and delivered to the upper layer. The variable *lack* records the sequence number up to which an acknowledgment, either explicit (via an ACK) or implicit (via a piggybacked acknowledgment in a DATA or NAK packet) has been most recently sent to the remote. The receiver’s active window consists of the sequence numbers from $lack + 1$ through $lack + W$ (modulo SEQ).³ A boolean array *recbuf* of size SEQ indicates the sequence numbers in this window that have been received (out of order) and are being buffered for returning to the upper layer. This buffering is required in order to deliver packets in the correct order to the upper layer. Another boolean array *nakd* of size SEQ is used to remember the sequence numbers that have recently been negatively acknowledged. As in the case of the sender, the receiver does not explicitly buffer packets, recording only whether a message has been received from the remote, but not yet delivered to the upper layer.

The protocol initialization code sets *lack*, *rack* and *recseq* to 0, *sendseq* to 1, and all entries in the arrays *nakd* and *recbuf* to false. The protocol’s main loop consists of busy waiting for one of three events to occur: the arrival of an incoming packet, a user request to send a packet, and timeout. These events, and the actions taken by the protocol in response to these events, are described in more detail below. See also the three flowcharts in Figure 2.

(E1): a packet arrival over the communication link (lower-layer interface): the packet is first checked for header checksum errors, and silently discarded if it has a header error. Otherwise, if the piggybacked acknowledgment is for a sequence number in the sender’s active window, this is used to update *rack*. This subsumes the handling of explicit ACK packets. That is, if the type of the incoming packet is an ACK, no further processing beyond what was just described is required. If the incoming packet is a NAK for a sequence number in the sender’s active window, the requested DATA packet is resent. If the incoming packet is a DATA packet, its data checksum is first

² This is a *data independence* abstraction [Wol86].

³ Unless stated otherwise, we shall henceforth assume that all arithmetic is modulo SEQ .

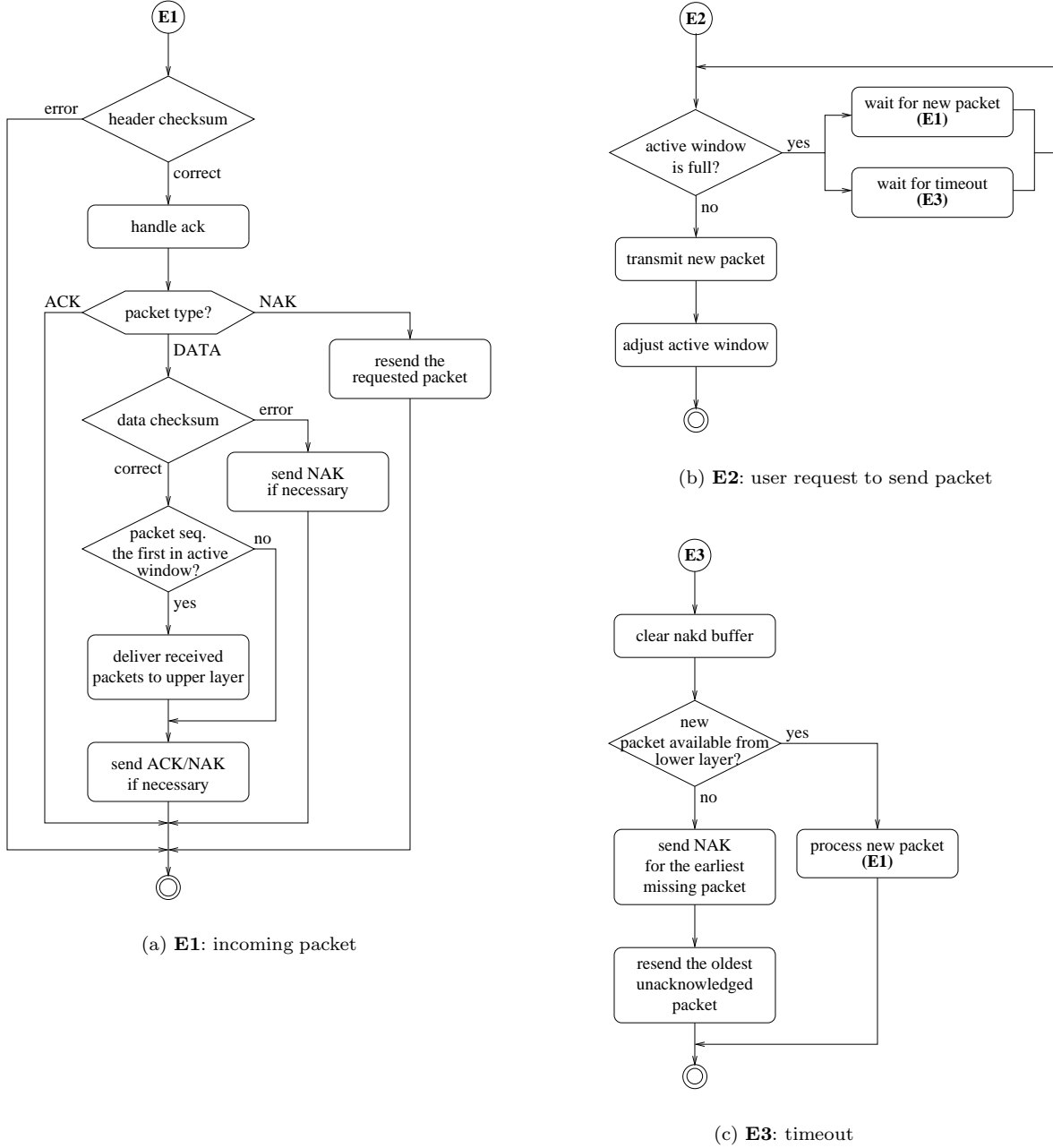


Fig. 2: Flowcharts of event handlers. Double circles represent return points.

verified. If the data is corrupted, the packet’s sequence number is in the receiver’s active window, and a NAK has not been sent for that sequence number since the previous timeout (its *nakd* entry is false), then a NAK is sent for that sequence number, and its *nakd* entry is set to true. If, on the other hand, the data is valid, and the packet number is the first in its active window (bears the sequence number $recseq + 1$), then the newly arrived packet is delivered to the upper layer. Furthermore, any later packets that have been buffered, and all of whose “predecessors” have been delivered to the upper layer,

are also returned, in order, to the upper layer. At each point, *recseq* is appropriately incremented, thus shifting up the active window.

If it is subsequently found that $\lceil W/2 \rceil$ or more packets have been received since the last ACK (implicit or explicit) was sent, an explicit ACK is generated for *recseq*, and *lack* appropriately updated. If, however, the sequence number of the newly arrived DATA packet is not equal to $recseq + 1$, meaning that there are some missing sequence numbers in between, the newly arrived packet is buffered (in *recbuf*), if not already received, and NAKs

are generated for all “earlier” missing packets for which a NAK has not been sent since the last timeout.

(E2): a user request to send a new message (upper-layer interface): The sender first checks if there is an opening in its active window (i.e., the active window size is less than W). If so, the new message is transmitted after being assigned the next new sequence number (*sendseq*), and the sender’s active window’s upper edge suitably adjusted. If, however, the sender’s window is full, it must wait for an opening (created by the receipt of an ACK, see above) before it can send the new message. In this case, it busy-waits in a loop, waiting for the arrival of a new packet (see (E1) above), or for the occurrence of a timeout (see (E3) below).

(E3): a timeout: The *nakd* buffer is first cleared, signaling that fresh NAKs may need to be sent out. If there is no packet in the receive buffer (from the lower interface), then a NAK is sent out for the “earliest” missing sequence number (*recseq*+1) in the active receiving window. Further, the sender resends the “oldest” message (if one exists in its active window), for which it has not received an acknowledgment from the remote. If, on the other hand, there is a packet available from the lower interface, we follow (E1) above.

Our models of the i-protocol are based on the simplified structure of the protocol depicted in Figure 1, where data packets are transmitted in one direction only: from the sender side to the receiver side. As a result of this assumption, several simplifications to the protocol’s logic are possible. For example, since the sender does not receive data packets, the handling of data packets in (E1) is removed from the sender’s specification. Similarly, (E2), in its entirety, is removed from the receiver’s specification. On the other hand, both the sender and receiver execute (slightly different versions of) the timeout procedure.

The medium processes are modeled as FIFO buffers of size 1 that may potentially lose or corrupt packets. That is, after receiving a packet, a medium process non-deterministically chooses to either transmit the packet correctly, transmit a corrupted packet, or drop the packet entirely. Packet corruption is modeled by two boolean checksum fields, one for packet headers and the other for data. A packet is correctly transmitted if both checksums are true. If either checksum is false, it is corrupted.

Our models of the i-protocol were derived from the C code of the implementation and involved a number of abstractions aimed at reducing the size of the protocol’s state space. Most of these are discussed at length in Section 5, but one abstraction that is used in all our models is to reduce the size of the message-sequence space from a fixed value of $SEQ = 32$ (a defined constant in the GNU implementation) to the value $2W$, where W is the window size of the protocol.⁴ The rationale behind this abstraction is that the i-protocol is

⁴ It is worth noting that our abstraction of the message sequence space does not affect the window size W . This is a parameter

a sliding-window protocol that implements a “selective repeat” strategy [Tan96]: the receiver will accept and store a packet from the sender that falls within the receiver’s window regardless of whether or not earlier packets in the window have been damaged or lost. The packet is kept within the data link layer and not passed to the network layer until all lower-numbered packets have been delivered to the network layer in the correct order. Tanenbaum notes that this strategy allows the receiver to receive packets out of order and refers to this capability as “nonsequential receive.”

When the receiver advances its window after correctly receiving a window’s worth of packets, the new receive window must not overlap with the sender’s original send window; otherwise, the receiver will not subsequently be able to distinguish new packets from retransmitted ones that it has already delivered to the network layer. Tanenbaum has observed that to ensure there is no overlap, the maximum window size must be at most half of the range of sequence numbers. Therefore, by changing SEQ from 32 to $2W$, we greatly reduce the size of the i-protocol’s state space (when W is instantiated with small values such as 1 and 2) and, at the same time, arrive at a window size that is appropriate for the selective-repeat strategy. Indeed, for a system where SEQ is larger than $2W$, many of its configurations will be observationally equivalent [Mil89] to one another.⁵

3 The Livelock Error

The livelock error exhibited by version 1.04 of the i-protocol is illustrated in Figure 3 for the case of $W = 2$, medium buffer capacity 1, and assuming that one side acts as a sender and the other as a receiver. Initially, DATA1 sent by the sender is successfully received by the receiver, which responds with ACK1. This ACK is dropped by the medium. The sender then sends DATA2, which is also lost. The sender then enters its timeout procedure, where it first sends out NAK1 for the data packet it is expecting from the receiver, despite the fact that the receiver is not sending it any data. The sender then resends the packet DATA1. These (and all subsequent packets) are correctly delivered by the medium. Meanwhile, the receiver also times out, but finding the messages NAK1 and DATA1 in its receive buffer, processes them. However, it silently ignores NAK1, since it has never sent a DATA packet with sequence number 1. It also ignores DATA1, since 1 is not in its current receive window. This cycle can now repeat forever, with the sender

of the protocol whose value is determined during the parameter-negotiation stage of the protocol.

⁵ We strongly conjecture that, for $W \leq \lfloor SEQ/2 \rfloor$, our models of the i-protocol using a sequence space of $2W$ are observationally equivalent to the corresponding models in which $SEQ = 32$. The proof of this result would involve exhibiting a bisimulation between the concrete model (in which $SEQ = 32$) and the abstract model (in which $SEQ = 2W$).

sending messages to the receiver, which the receiver ignores, resulting in no messages being accepted from, or delivered to, the upper layer in spite of the medium behaving perfectly from this point onwards.

The livelock error arises because there is no flow of information from the receiver to the sender regarding the sequence numbers up to which the receiver has received all messages. A simple fix for this problem consists of sending an up-to-date ACK on the receipt of a NAK for sequence number *sendseq*, provided that the active send window is empty.

4 The Verification Tools

For each of the tools used in our case study, we give a brief overview of the tool’s functionality, describe the modeling techniques supported by the tool that were most important in the case study, and, for illustrative purposes, show how we encoded the sender process of the i-protocol in the tool’s input language. We also discuss how the property of “potential livelock” was encoded and verified.

4.1 Cospan

Cospan [HHK96] is a model checker for synchronous systems based on the theory of ω -automata [Tho90]. In this framework, the system to be verified is specified as an ω -automaton P , the task the system is intended to perform is specified as an ω -automaton T , and verification is accomplished by checking for containment of the language $\mathcal{L}(P)$ in the language $\mathcal{L}(T)$. **Cospan** performs this check by considering the infinite behaviors of the product automaton of P and T against the “acceptance structure” of T . “Recur edges” and “cycle sets” are examples of acceptance structures and the former is discussed in some detail below. **Cospan** implements both a symbolic (BDD-based) algorithm for checking language containment and an explicit state-enumeration algorithm. The former is “on-the-fly” in the sense that it performs the model checking during construction of the state transition graph for the system. The BDD-based algorithm does not generate an explicit representation of the state transition graph for the system; it checks for violations of safety properties during reachability analysis. **Cospan** terminates as soon as such a violation is encountered.

4.1.1 The S/R Specification Language

The input language of **Cospan** is S/R. S/R is a data-flow language: it concentrates on how data is updated and has few control structures. A system in S/R is typically given as the parallel composition of several component processes, where each process contains a set of assignments to variables. The order of appearance of the assignment statements does not affect the semantics. The

system environment must be explicitly modeled as an S/R process, with inputs to the system assigned non-deterministically.

S/R has two kinds of variables: state variables and selection variables. A state of an S/R program is a valuation of its state variables. A transition of an S/R program is an assignment of new values to the state variables of its component processes. Assignment statements have guards that may involve both state and selection variables. The guards serve as conditions for the transitions.

Selection variables model the input/output of a system component. When a selection variable of a component is imported by another component, it serves as an output of the first component and an input to the second component. The values of selection variables are determined by the state of the S/R program and they are also used in transition conditions to help determine the next state.

S/R is also a synchronous language. All system components change their local states simultaneously as all state variables are updated simultaneously. Communication among processes is achieved through importing and exporting selection variables. Asynchronous composition can be modeled by introducing nondeterministic delay in the components, or through the use of a nondeterministic scheduler. A nondeterministic scheduler is essentially a random-number generator, and when there are multiple enabled transitions, the current value of the random number determines which transition is selected for execution. This creates the interleaving needed for modeling asynchronous systems, and represents the approach we followed in modeling the i-protocol (which is indeed asynchronous) in S/R.

S/R supports nondeterministic, conditional (i.e., **if-then-else**) variable assignments; variables of type bounded-integer, enumerated and boolean; arrays and records; and integer and bit-vector arithmetic. Modular hierarchy, scoping, homomorphism declaration, and Buchi fairness are also available. Modular hierarchy is introduced into an S/R program through the declaration of nested processes. The scope of a variable is the process in which the variable is declared, unless the variable is explicitly exported. A homomorphism declaration specifies how an abstract system model maps onto a concrete model. Buchi fairness and generalized Buchi fairness (where multiple states are required to be visited infinitely often) can be modeled in S/R programs through the use of cycle sets and recur edges.

4.1.2 Modeling the i-Protocol in S/R

We modeled the i-protocol in **Cospan** as the parallel composition of a sender and a receiver process together with processes modeling the medium and upper-layer entities. Our encoding of the protocol can be better understood by considering the S/R code for the sender process, which is given in Figure 4.

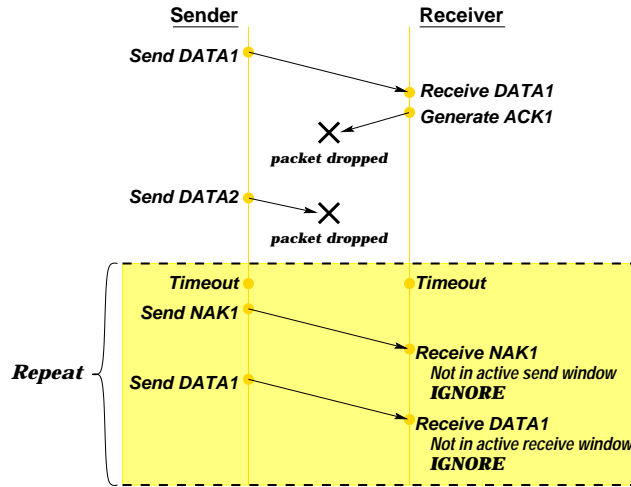


Fig. 3: Error scenario for livelock in version 1.04 of the i-protocol.

The `import` statement in the sender process imports the processes `s`, `medium[0]`, and `medium[1]` so that the sender can access the selection variables of these processes. The process `s` models an integer of finite range, and is used to implement a nondeterministic scheduler. Conditional tests on the value of `s` are associated with various transitions of the sender and receiver processes to create interleaving. The processes `medium[0]` and `medium[1]` are the medium processes.

The next several statements declare the state and selection variables of the sender process. State variables are declared using the keyword `stvar`, followed by the variable names and the variable type. For example, the state variable `seq`, which represents the sequence number of the incoming packet, is of the finite range type $(0..MAX_SEQ)$, where `MAX_SEQ` is a constant defined elsewhere to be $2W - 1$ for window size W . The state variable `$` is an enumerated type capturing the local control points of the sender process (`beginning`, `nak_resend`, `timeout_resend`).

Selection variables are declared similarly using the keyword `selvar`. A special case is the statement

```
selvar # := $
```

where the selection variable `#` is declared to have the same type, and always the same value, as the state variable `$`. In the sender process, the two most important selection variables are `pop1` and `push0`, boolean variables indicating if the sender is ready to communicate with `medium[1]` and `medium[0]`, respectively. The selection variables `outseq`, `outak`, and `outpty` represent respectively the following information about packets transmitted by the sender: the sequence number, the piggybacked acknowledgment sequence number, and the packet type. The initial values of the state variables are defined using the `init` statement, whereas no initial value designations are needed for selection variables, because the values of selection variables are determined by the current state.

Assignment statements (designated by the keyword `asgn`) determine how the variables change their values. State variables are assigned using the operator “`->`” and selection variables are assigned using “`:=`”. The operator “`+`” indicates addition or logical-or, whereas “`*`” indicates multiplication or logical-and. The operator “`:`” denotes selection: $(s:0,2)$ is true if `s` is either 0 or 2. The operator pair “`? !`” is used for writing “if...then...else...” expressions. For example, the statement

```
asgn seq -> medium[1].seq ? pop1 | seq
```

assigns the state variable `seq` the value of the variable `seq` of `medium[1]` (`medium[1].seq`) if `pop1` is true; otherwise it is assigned its own value (`seq`).

The assignment statements assign values to all the state variables except `$`, which captures the local control points of the sender process. This variable is assigned using the “transition structure”, the section of code in Figure 4 delimited by the keyword `trans`. The transition structure comprises a sequence of “transition blocks”, each of which has an associated state predicate followed by a list of transitions. For example, in the transition block

```
nak_resend
-> beginning: (s:0) * (medium[0].#:empty)
-> $: else;
```

the state predicate is `nak_resend` (shorthand for the condition $(\$ = nak_resend)$) and the block contains two transitions, each starting with `->`. The intent here is that if the variable `$` has the value `nak_resend`, then its next value will be `beginning` if the condition $(s:0) * (medium[0].#:empty)$ is true; otherwise its next value will be the same as its current value (`$`).

Checksums are not directly modeled in the sender process but rather are declared as selection variables in the medium processes `medium[0]` and `medium[1]`. When the sender receives a packet from `medium[1]`, it reads the

```

proc sender
  import s, medium[0], medium[1]
  stvar sendseq, rack, seq:      (0.. MAX_SEQ)          /* finite range type          */
  stvar $:                       (beginning, nak_resend, timeout_resend)
  /* enumerated type; its values can be regarded as names for local control points          */
  selvar # := $                  /* # is always equal to $          */
  selvar outseq, outak:          (0.. MAX_SEQ)          /* seq and ack of out-going packets */
  selvar outpty:                 (ack, nak, data)        /* packet type of out-going packets */
  selvar push0, pop1, openwindow: boolean
  /* if push0 is true, send to medium[0]; if pop1 is true, receive from medium[1]          */
  init $ := beginning, seq := 0, sendseq := 1, rack := 0

  /* What follows are assignments for variables. An expression of the form (s:0) means (s=0) */
  /* An expression of the form (a ? p | b) means (if p then a else b)                       */
  asgn sendseq -> (sendseq+1) mod dbl_WINSIZE ? push0 * ($:beginning) * (s:0) | sendseq
  /* increment sendseq by 1 if data is being sent. "->" is for state variable assignments  */
  asgn rack    -> medium[1].ak ? pop1 * (medium[1].hck = 1) /* update rack when necessary */
                    * (~((medium[1].ak = sendseq)
                        + (diff(medium[1].ak, rack) > WINSIZE)
                        + (diff(sendseq, medium[1].ak) > WINSIZE)))
                    | rack
  asgn seq     -> medium[1].seq ? pop1 | seq
  asgn pop1    := (medium[1].# = full) * (s:1) * ($:beginning) /* "=" is for selvar assignments */
  asgn push0   := (medium[0].# = empty)
                    * ( ($:beginning) * ((s:0)*openwindow+(s:2))
                      + ($:nak_resend) * (s:0,2)
                      + ($:timeout_resend) * (s:0,2) )
  asgn openwindow := ((diff(sendseq, rack) <= WINSIZE) * (sendseq ~= rack))
  /* diff is a macro that computes the difference between its arguments modulo 2*window_size */
  asgn outpty   := nak ? (medium[0].# = empty) * (#:beginning) * (s:2) | data
  asgn outak    := 0 /* sender does not receive data          */
  asgn outseq   := sendseq ? push0 * (#:beginning) * (s:0)
                    | 1 ? push0 * (#:beginning) * (s:2)
                    | (rack+1) mod dbl_WINSIZE ? push0 * (#:timeout_resend)
                    | seq ? push0 * (#:nak_resend)
                    | 0
  kill (s:1,2) * ($:nak_resend,timeout_resend) /* truncate search of state space */
  kill (s:0) * (~openwindow) * ($:beginning)

  trans
  /* transition structure, assigning values to $          */
  beginning /* ready to send or receive          */
  -> beginning: (s:0) * openwindow * (medium[0].# = empty) /* send data          */
  -> nak_resend: /* receive nak and resend data          */
    (medium[1].# = full)
    * (s:1) * (medium[1].hck = 1) * (medium[1].pty = nak)
    * ~( (medium[1].seq = sendseq)
        + (diff(medium[1].seq, rack) > WINSIZE)
        + (diff(sendseq, medium[1].seq) > WINSIZE) )
  -> timeout_resend : /* time out and send nak          */
    (medium[0].# = empty) * (s:2) * (sendseq ~= ((rack+1) mod dbl_WINSIZE))
  -> $: else;

  nak_resend /* resend data after nak received          */
  -> beginning: (s:0) * (medium[0].# = empty)
  -> $: else;

  timeout_resend /* resend data on timeout          */
  -> beginning: (s:0) * (medium[0].# = empty)
  -> $: else
end sender

```

Fig. 4: S/R code for the sender process.

header checksum `medium[1].hck` but ignores the data checksum since, in our model, data flows in only one direction (from the sender to the receiver). The selection variables modeling the checksums are assigned values nondeterministically and their values are used only when the sender or the receiver process read from these variables.

The assignment statements and the transitions in the transition structure of the sender process depend on conditions that test the value of the nondeterministic scheduler `s`. For example, we use the value 0 of `s` to select the data-transmission operation of the sender. Thus all variable updates related to data transmission in the sender depend on the condition `(s:0)`. The sender process is regarded as an *active process* because its transitions can be explicitly selected for execution by the nondeterministic scheduler. On the other hand, the medium processes and the processes representing the upper-layer entities are encoded as *passive processes*; that is, their transitions do not depend on the nondeterministic scheduler. These processes are encoded in such a way that they are ready to synchronize with their communicating partners any time the partners are ready. This reduces the range of values needed by the nondeterministic scheduler, and thus reduces the size of the state space.

Finally, the `kill` statement is used to truncate Cospan’s search of certain parts of the state space. For example, the statement

```
kill (s:0)*(~openwindow)*($:beginning)
```

prevents exploration of those parts of the state space where the condition `(s:0)*(~openwindow)*($:beginning)` is true. This is desired because we do not want to select the data-transmission operation of the sender (`s` has the value 0) when the sender is ready to communicate with the medium (`$` has the value `beginning`), but there is no opening in the sender’s window (`~openwindow`).

4.1.3 Encoding the Livelock Error in S/R

Cospan supports a notion of “recur edge” to specify a system’s acceptance structure. An infinite execution sequence of the system is *accepted* if it contains an infinite number of recur edges. All other infinite execution sequences of the system are reported as errors by Cospan. Note that because Cospan is a model checker for *finite-state* systems, the only source of infinite execution sequences is the presence of cycles in the system state space. Thus, to use Cospan’s recur-edge facility to search for livelocks, we need to ensure that only cycles corresponding to actual livelocks are flagged as errors, and that all other cycles are deemed acceptable. Given that a livelock is a cycle in which no packets are delivered to the upper layer despite the fact that packets are neither dropped nor corrupted, the acceptable or non-livelock cycles are those along which at least one of the following transitions occur: a packet is correctly delivered to

the upper layer (i.e. without corruption and in order), a packet is dropped or corrupted, or a spurious timeout occurs.⁶ Consequently, we designated each of these types of transitions as recur edges in our Cospan model of the protocol. Recall that the underlying cause of livelock errors in the i-protocol is the dropping of packets, and indeed message drops will occur along the execution sequence leading up to the livelock cycle (but not within the cycle itself).

Recur-edge designations are made in the “property file” of a Cospan specification. For example, the designation

```
recur (drop:1)
```

means that the transitions leaving a state in which the value of state variable `drop` is 1 are recur edges. State variable `drop` indicates whether or not a packet is being dropped.

4.1.4 Cospan Run-Time Options

Cospan’s provides both an explicit-state analysis algorithm and a symbolic analysis algorithm. The explicit-state algorithm is the default (no command-line option needed). Cospan can be directed to use the symbolic analysis algorithm by specifying the option “-b” in the command line. In this study, we obtained data on the i-protocol using both algorithms and in each case we used the “-q1” option, which directs Cospan to use its localization reduction techniques. Cospan’s symbolic analysis algorithm and localization reduction techniques are designed to combat state-space explosion. Section 6, which presents our various model-checking results for the i-protocol case study, compares the performance of Cospan’s symbolic and explicit-state algorithms.

4.2 Murφ

The Murφ verification system [Dil96] consists of the Murφ compiler and the Murφ description language. The Murφ compiler generates a special-purpose verifier (C++ program) from a Murφ description. The Murφ description language, inspired by Chandy and Misra’s Unity [CM88], is based on a collection of guarded commands (condition/action rules) that are executed repeatedly in an infinite loop.

Verification in Murφ is based on explicit-state enumeration. A Murφ state is an assignment of values to all of the global variables of the description. A Murφ verifier performs breadth-first search in the state graph determined by a Murφ description, storing all the states it encounters in a large hash table. The user can alternatively specify as a command-line option that depth-first search be used.

⁶ Timeouts are nondeterministic events in our Cospan model of the i-protocol. Therefore, the possibility of spurious timeouts exists, meaning that a timeout occurs even though other (non-timeout) transitions are available for execution.

4.2.1 The Mur φ Specification Language

A Mur φ specification consists of constant and type declarations, variable declarations, procedure declarations, rule definitions, and a description of the start state. Data types of the language include subranges, enumerated types, arrays, and records.

A Mur φ program models a state transition system. A state represents a valuation of global variables whose initial values are set by the `STARTSTATE` block. Transitions are defined by a set of guarded-command rules where the guard is a boolean expression of the global variables and the command is a block of sequential statements updating the global variables. Temporary variables can be declared locally in the command portion of a rule but they are not part of the state. A rule is enabled at a state when its guard evaluates to true in the state. The target state of a transition is computed by executing the command of an enabled rule. When multiple rules are enabled, one of them is nondeterministically chosen for the transition.

There is no parallel composition in Mur φ . To specify concurrent systems, the programmer needs to declare the global variables as the union of the state variables of the conceptual subsystems. Depending on how the global variables are handled, Mur φ can model either synchronous or asynchronous systems. For synchronous systems, each rule updates all the variables; for asynchronous systems, a rule only updates the variables corresponding to a subsystem. Communication between subsystems is implemented by accessing shared variables.

Mur φ provides three mechanisms for verifying programs written in the Mur φ description language.

- **Deadlock Detection:** Mur φ can be directed to search for deadlock states, i.e., a state with no successors.
- **Assertion and Error Checking:** Explicit `assert` and `error` statements can be placed in Mur φ rules, facilitating the checking of a variety of state properties including invariants. If an `error` statement is executed or an `assert` statement is violated, the verifier halts.
- **Liveness Verification:** Mur φ supports a subset of Linear Time Temporal Logic (LTL) for the specification of liveness and safety properties. Six types of temporal formulas and their conjunctions can be checked by Mur φ version 2.7L.⁷ By default, liveness formulas are checked under the assumption that every rule is weak-fair,⁸ unless it is declared to be not fair (“`RULE UNFAIR`”). If a liveness specification is vi-

⁷ Verification of liveness properties was removed from subsequent releases of Mur φ due to complications in integrating this feature with symmetry reduction, a state-space management technique. Since, in the case of Mur φ , we found it essential to encode livelock in the i-protocol using liveness properties, we used version 2.7L of the tool instead of later releases.

⁸ A rule is weak-fair if it is executed infinitely often when enabled infinitely often.

olated, Mur φ produces a trace leading to the cause of the violation.

4.2.2 Modeling the i-Protocol in Mur φ

Using a set of guarded-command rules for each process, we modeled the i-protocol in Mur φ as the parallel composition of four conceptual processes: a sender, receiver, and two channels connecting the sender and receiver. The modeling is illustrated in Figure 5 for the sender process. The basic control structure of the sender is that of a top-level control loop inside of which are three nondeterministic alternatives: send a data packet, receive a packet, and timeout. This structure is captured in the Mur φ code by rules `sender send message`, `sender getpkt`, and `sender timeout`, respectively. Rules `sender getpkt 1` and `sender timeout 1` are “continuations” of `sender getpkt` and `sender timeout`, respectively.

4.2.3 Encoding the Livelock Error in Mur φ

To check for livelock in the Mur φ specification of the i-protocol we used LTL model checking. Specifically, we encoded the property of livelock freedom as the LTL formula

`ALWAYS EVENTUALLY action = progress`

where `action` is a variable recording the nature of the action taken by each executed rule. When the value of `action` is set to `progress`, the corresponding rule represents a progress step, in the sense of a recur edge in *Cospan* (Section 4.1.3) or a progress-labeled statement in *Spin* (Section 4.3.3). The LTL formula for livelock freedom thus directs Mur φ to check for progress loops.

4.2.4 Mur φ Run-Time Options

Verifying a Mur φ specification is a three-step process: generate a C++ program from the Mur φ input file, compile the program using `gcc`, and then run the resulting executable. Things work in a similar manner for *Spin*; see Section 4.3.4. In our experiments, we compiled the C++ program at optimization level 4. The command-line options we used for the executable are as follows:

`verifier -pn -tv -m<XX>`

where `verifier` is the name of the executable, `-pn` instructs Mur φ not to report state counts periodically (to save time), `-tv` causes an error trace to be printed, and `-m<XX>` means allocate `XX` MB of memory for the hash table. For smaller runs, we used the `-k<YY>` option which specifies memory usage in kilobytes instead of megabytes.

4.3 Spin

Spin [Hol97] is an explicit-state model checker for asynchronous systems specified in the language *PROMELA*.

```

VAR
  -- Channels
  DataChan  : Channel;
  AckChan   : Channel;

  -- Sender states (All started with "S" to
  -- be distinguished from receiver states.)
  SState    : SenderState;
  SSendSeq  : SequenceNumber;
  SRack     : SequenceNumber;
  SSeq      : SequenceNumber;

RULE "sender send message"
  SState = S_START
    & openwindow(SSendSeq, SRack)
    & isempty(DataChan)
==>
BEGIN
  action := tau;
  output(DataChan, DATA, SSendSeq, 0);
  SSendSeq := inc(SSendSeq);
END;

RULE "sender getpkt"
  SState = S_START
    & isfull(AckChan)
==>
VAR
  pak : Packet;
  hck : Checksum;
  dck : Checksum;
BEGIN
  input(AckChan, hck, dck, pak);
  IF hck = 1 THEN
    ASSERT(pak.pty = ACK | pak.pty = NAK);
    IF in_open_interval(pak.ack, SSendSeq, SRack)
    THEN
      SRack := pak.ack;
    ENDIF;
    IF pak.pty = NAK THEN
      IF in_open_interval(pak.seq, SSendSeq, SRack)
      THEN
        SSeq := pak.seq;
        SState := S_GETPKT_1;
      ENDIF;
    ENDIF;
  ENDIF;

  action := tau;
ELSE
  action := progress;
ENDIF;
END;

RULE "sender getpkt 1" -- output
  SState = S_GETPKT_1
    & isempty(DataChan)
==>
BEGIN
  output(DataChan, DATA, SSeq, 0);
  SState := S_START;
  CLEAR SSeq;
  action := tau;
END;

RULE "sender timeout"
  SState = S_START
    & isempty(DataChan)
==>
BEGIN
  output(DataChan, NAK, 1, 0);
  IF openwindow(SSendSeq, SRack) THEN
    -- arbitrary timeout as progress
    action := progress;
  ELSE
    action := tau;
  END;
  IF SSendSeq != inc(SRack) THEN
    SState := S_TIMEOUT_1;
  ELSE
    SState := S_START;
  END;
END;

RULE "sender timeout 1"
  SState = S_TIMEOUT_1
    & isempty(DataChan)
==>
BEGIN
  output(DataChan, DATA, inc(SRack), 0);
  SState := S_START;
  action := tau;
END;

```

Fig. 5: Mur ϕ code for the sender process.

Safety and liveness properties are formulated using LTL, and LTL model checking is performed on-the-fly and with partial-order reduction, if specified by the user. Moreover, model checking can be done in a conventional exhaustive manner, or, when this proves to be impossible due to state-space explosion, using an efficient approximation method known as bitstate hashing. Spin also offers a lossless compression option, which reduces memory requirements at the expense of additional running

time. The current version of Spin also supports a number of optimization techniques [Hol99] including dead variable elimination and statement merging. Statement merging is an optimization technique aimed at merging a sequence of statements, each of which represents an internal computation of some component, into a single atomic step. This technique, aimed at managing the size of the system state space, reduces the number of control

points in a system component as well as the amount of interleaving in the overall system.

4.3.1 The PROMELA Specification Language

PROMELA is a nondeterministic guarded-command language with influences from Hoare’s CSP and the language C. PROMELA includes support for control flow, data structures, interrupts, bracketing of code sections for atomic execution, dynamic creation of concurrent processes, and a variety of synchronous and asynchronous message-passing primitives. Message passing is via bounded message channels; the maximum number of messages any particular channel can contain is included in the channel’s declaration.

The statements of a PROMELA specification may be labeled. A statement label uniquely identifies the control state of the process in which it is defined. Moreover, PROMELA supports three special kinds of labels: progress, acceptance, and end-state. A label is identified as a progress label by prefixing the string “progress” to it; similarly, for the other kinds of labels. Progress and acceptance labels are used to define two complementary types of liveness properties, while an end-state label marks a control state as a valid termination point (as opposed to e.g. a statement at which the system deadlocks). We shall focus further discussion on progress labels, as these are what we used in our case study.

4.3.2 Modeling the i-Protocol in PROMELA

We modeled the i-protocol in PROMELA as the parallel composition of a number of processes, including the sender, receiver, and two medium processes. To illustrate our encoding, Figure 6 contains the PROMELA code for the sender process (`proctype sender()`). The outermost control structure in this code is a do-loop, capturing the infinite execution of the sender. The sender has three alternatives within the loop:

1. Transmit data if there is an opening in the sender’s window.
2. Receive a packet from the medium and then process the packet.
3. Time out, send a negative acknowledgment, and then re-send the data.

The choice as to which alternative will be executed during an iteration of the do-loop is nondeterministic. In particular, the third alternative (sender timeout) may be chosen when the first alternative (sender transmits data) is also enabled; this occurs precisely when the third alternative is chosen and there is an opening in the sender’s window. This is an example of a “spurious” timeout discussed above in the context of our S/R encoding of the protocol. Recalling that discussion, such an event should be considered “progress.” Consequently, the `skip` statement preceded by the `openwindow` guard in the third

alternative of the sender is marked with a progress label (`progress_t1`).

4.3.3 Encoding the Livelock Error in PROMELA

Statements bearing progress labels play the role of recur edges in *Cospan* (see Section 4.1.3). That is, *Spin* treats any infinite execution sequence that does not infinitely often encounter a progress-labeled statement as an (liveness) error. Such errors are referred to as *non-progress loops*, in reference to the fact that *Spin*, like *Cospan* (and the other model checkers considered in this case study), is a model checker for finite-state systems. As such, infinite execution sequences arise only in the context of cycles or loops through the state space.

To use *Spin* to check for livelocks in the i-protocol, it sufficed to annotate our PROMELA specification of the protocol with progress labels in a manner virtually identical to our use of recur-edge labelings in our S/R specification. As such, any non-progress loops reported by *Spin* can be seen to correspond to livelocks in the protocol.

4.3.4 Spin Run-Time Options

Verifying a PROMELA program using *Spin* is a three-step process. One first runs *Spin* itself to generate a C program called `pan.c` from the input PROMELA file. Next, a C compiler such as `gcc` is used to compile `pan.c` into an executable binary called `pan`. Finally, `pan` is executed. Run-time options or parameters may be specified at each of these stages, with the ones for compilation of `pan.c` and execution of `pan` being the most important. We discuss below some of the key options we used during our analysis of the i-protocol using *Spin*.

To compile `pan.c`, the following command-line directive was used:

```
gcc -w -o pan -DMEMCNT=31 -DNP pan.c
```

The interesting command-line options here are `-DMEMCNT`, which specifies the memory-usage limit (2^{31} bytes, in this case); and `-DNP`, which enables non-progress loop detection. Documentation on other options can be found within the *Spin* web site [Spi]. One option that we did not use is `-DNOREDUCE`, which disables partial-order reduction. Therefore, in all *Spin* verification runs reported in this paper, partial-order reduction was in effect.

To run the executable `pan`, a command-line directive of the following form was used:

```
./pan -n -l -c1 -m10000 -w19
```

The interesting options in this case are `-l`, which enables searching for non-progress loops; `-c1`, which means “stop on first error”; `-mN`, specifying a maximum search depth of N ; ⁹ and `-wN`, meaning “use a hash table having 2^N entries”.

⁹ In *Spin*, when the search-depth limit is reached, an error is reported and the state-space search is truncated.

```

active proctype sender()
{
    byte sendseq=1; /* next sequence number to use when sending */
    byte rack;      /* last sequence number acked by remote */
    byte pty;       /* packet type */
    byte seq;       /* packet sequence number */
    byte ak;        /* ack field in packet */
    bool hck, dck; /* header and data checksum */
    byte tmp;

#define openwindow (diff(sendseq, rack) <= HWIN && sendseq != rack)

    do
    :: /* opening in remote window */
        openwindow ->
            s2m!data(sendseq, 0); /* send data packet */
            inc(sendseq)          /* update sendseq */

    :: /* read packet from medium */
        m2s?pty(seq, ak, hck, dck) ->
            if
            :: hck -> /* good header */
                assert(pty == ack || pty == nak);
                if
                :: (!(ak == sendseq)
                    || diff(ak, rack) > HWIN
                    || diff(sendseq, ak) > HWIN) ->
                    rack = ak
                :: else
                    fi;
                if
                :: pty == nak ->
                    if
                    :: (!(seq == sendseq)
                        || diff(seq, rack) > HWIN
                        || diff(sendseq, seq) > HWIN) ->
                        s2m!data(seq, 0)
                    :: else
                        fi
                :: else
                    fi
                :: else /* bad header */
                    fi

            :: /* timeout */
                s2m!nak(1, 0);
                if
                :: openwindow ->
                    progress_t1: skip
                :: else
                    fi;
                if
                :: (sendseq != (rack+1)%WIN) ->
                    s2m!data((rack+1)%WIN, 0)
                :: else
                    fi
            od
    }
}

```

Fig. 6: PROMELA code of the sender process.

The values we used for N in the `-mN` and `-wN` options varied depending on the number of states in the protocol. As explained in Section 6, for each tool considered in this paper, we obtained model-checking results on a number of different versions of the protocol, some with the livelock error present (buggy versions) and others with this error absent (fixed versions). Moreover, for each tool, we subjected the specification of the protocol to various abstraction techniques, successively making the specification more and more abstract. As a consequence, the size of the protocol’s state space varied considerably from run to run.

To account for the varying state-space sizes, we varied the value of N in the option `-mN` from 10^2 to 10^6 , the smaller values for versions that were either buggy and/or more abstract, and the larger values for versions that were either fixed and/or less abstract. We similarly varied the value of N in option `-wN` from 11 to 25. The best value of N for this option was such that 2^N is approximately the number of states. Finally, we note that the `-c1` option (stop on the first error) only had an impact performance-wise on the buggy versions of the protocol.

4.4 XMC

XMC [RRR⁺97] is a model checker for systems specified in XL, a value-passing process calculus, and properties specified in the modal mu-calculus. Besides the core model checker, XMC is equipped with a type checker for XL, an XL compiler that translates an XL specification into an efficient representation of the system’s global automaton, and a graphical user interface written in TCL/TK.

XMC’s model checker is written in under 200 lines of XSB tabled Prolog code. XSB [XSB99] is a logic-programming system developed at SUNY Stony Brook that extends Prolog-style SLD resolution with tabled resolution. The principal merits of this extension are that XSB terminates on programs having finite models, avoids redundant subcomputations, and computes the well-founded model of normal logic programs.

4.4.1 The XL Specification Language

Systems to be verified in XMC are encoded in XL, a value-passing language similar in many ways to Milner’s CCS [Mil89]. An XL program is a collection of process definitions whose headers are process identifiers and bodies are XL expressions. The following CCS operators can be used in an XL expression: prefix, nondeterministic choice, parallel composition, input/output actions, relabeling, and restriction. Unlike CCS, which accepts only input/output actions as prefixes, XL uses prefixing to arrive at a general form of sequential composition: arbitrary XL process expressions can be prefixed to other XL process expressions. XL also supports data computation and value passing.

In XL, all variables are declared locally within a process. Data are represented by Prolog terms and computation is implemented by Prolog predicates. There is no explicit loop construct and recursion must be used instead for this purpose.

An XL program specifies a labeled transition system whose states are process expressions and transitions are either input/output actions or internal transitions representing internal computation. XL has a formal semantics [DR99] that closely follows CCS’s semantics plus an extension to accommodate XL’s extended syntax.

XL’s semantics is implemented by the Prolog predicate `trans`: $State \times Action \times State$ which defines the transition relation between process expressions. At first, `trans` was encoded directly from the formal semantics. Recently an optimizing compiler [DR99] for XL has been developed. The compiler efficiently translates XL specifications into a compact representation of the global automaton. During compilation, several optimizations are applied. These include elimination of dead variables from state expressions, partial evaluation of transition rules by pre-computing expressions in global transitions rules, and statement merging, where consecutive internal transitions are merged into a single atomic step. These optimizations permit the size of the global automaton to be reduced automatically, without relying on user annotations.

XMC verifies properties of XL programs expressed as modal mu-calculus formulas, a highly expressive temporal logic. In addition to `trans`, the core of the system consists of the predicate `models`: $State \times Formula$ which verifies whether a state represented by a process term models a given modal mu-calculus formula. The `models` predicate directly encodes the semantic equations for the modal mu-calculus. The encoding exploits the capabilities of the XSB logic programming system to compute least fixed points very efficiently. Greatest fixed-point computations are encoded as the negation of their dual least fixed points. This encoding reduces model checking to logic-program query evaluation. Furthermore, the goal-directed query evaluation mechanism of the XSB logic programming system yields a local, on-the-fly model checker.

4.4.2 Modeling the i-Protocol in XL

We modeled the i-protocol in XL as the parallel composition of four processes: the sender, receiver, and two medium processes. To illustrate our encoding, Figure 7 contains the XL code for the sender process. The sender has three nondeterministic top-level alternatives: send a data packet, receive a packet, and timeout. At the end of each alternative is a recursive call to the sender process, which illustrates the manner in which loops may be encoded in XL using recursion.

Each alternative consists of a sequence of (deterministic) statements, including communication, computa-

```

sender(SendSeq, Rack) ::=
  % send data
  (eval(SendSeq-Rack =< hwin), SendSeq \== Rack) o
  out(s2m, packet(data, SendSeq, 0)) o
  eval(NewSendSeq := SendSeq + 1) o
  sender(NewSendSeq, Rack)
#
% get packet
in(m2s, recvpak(packet(Pty, Seq, Ack), Hck, Dck)) o
if( Hck == true
,   if( (Ack \== SendSeq,
        eval(Ack-Rack =< hwin),
        eval(SendSeq-Ack =< hwin))
,     NewRack = Ack
,     NewRack = Rack
) o
if( Pty == nak
,   if( (Seq \== SendSeq,
        eval(Seq-NewRack =< hwin),
        eval(SendSeq-Seq =< hwin))
,     out(s2m, packet(data, Seq, 0))
)
) o
sender(SendSeq, NewRack)
,   sender(SendSeq, Rack)
)
#
% timeout
out(s2m, packet(nak,1,0)) o
if( (eval(SendSeq-Rack =< hwin), SendSeq \== Rack), action(progress)) o
eval(R is Rack+1) o
if( (SendSeq \== R)
,   out(s2m, packet(data, R, 0))
) o
sender(SendSeq, Rack).

```

Fig. 7: XL code for the sender process.

tion, and conditional branching. Communication statements are of the form `in(Channel,Data)` and `out(Channel, Data)` for inputting and outputting `Data` through `Channel`, respectively. Computation statements and branching conditions are written using Prolog predicates or built-in operators of the XSB Prolog system. For example, the user-defined predicate `eval` evaluates its argument, an arithmetic expression, modulo the window size W .

As discussed below, we use XMC’s model checker to check for non-progress loops in order to detect potential livelock in our XL specification of the i-protocol. Because the semantic model of XMC is labeled transition systems, transitions representing progress steps must be explicitly labeled as such (using the statement `action(progress)`) so the model checker can observe such transitions. In the case of the sender, a spurious timeout, i.e., one that occurs when there is an opening in the sender’s window, is considered progress.

4.4.3 Encoding the Livelock Error in XL

We used the following modal mu-calculus formula to specify potential livelock in the i-protocol:

$$\mu X.(\nu Y.(\neg\text{progress})Y) \vee \langle - \rangle X$$

This formula means that there exists a cycle reachable from the start state that does not contain a “progress” action. Thus, just as we did with the other tools, we identify livelock with non-progress loops.

4.4.4 XMC Run-Time Options

XMC, unlike other tools such as `Spin`, dynamically allocates memory as state-space exploration proceeds and does not directly control the search order (which is left to the underlying logic-programming system). As such, no run-time options are needed when using the tool.

4.5 Modeling Guidelines

We also identified several general guidelines that one should follow when modeling the i-protocol (and protocols in general) using a verification tool. These include minimizing the number of variables in a specification and maximizing atomicity; both techniques aim to keep state-space sizes at manageable levels. Below we briefly summarize how these guidelines are applied in the context of the different tools deployed in this case study.

- **Cospan:** In *Cospan* a state is a valuation of state variables. Thus one should use as few state variables as possible. In particular, a state variable should not be used if a selection variable can be used instead. Relatedly, one should also attempt to minimize the number of values in the range of each type of state variable.

One technique that can be used to achieve this effect is to atomicize internal transitions that lead to states in which one or more state variables take on new values, thereby avoiding the need to explicitly maintain these values in the range of the state variables in question. To illustrate, suppose that x is a state variable having the finite range type $0..3$, and suppose that we have the transition sequence $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$. That is, from the state in which x assumes a value of 0 there is a transition to a state in which x assumes the value of 1, and so forth and so on. Furthermore, suppose that the state in which x assumes the value of 1 is an internal state; i.e., $0 \rightarrow 1$ and $1 \rightarrow 2$ are internal transitions. Then we should atomicize the transitions $0 \rightarrow 1$ and $1 \rightarrow 2$ into one single transition, so that only the values $(0, 2, 3)$ need to be maintained in the range of x .

The use of passive processes rather than active processes whenever possible is also recommended, as this will reduce the required number of values in the range of the nondeterministic scheduler.

Another useful modeling guideline for *Cospan* is the use of the S/R `kill` statement in order to truncate *Cospan*'s search of the state space, as described in Section 4.1.

- **Mur φ :** In *Mur φ* states are determined by the value of global variables. Thus one should minimize the number of global variables. When a variable is live only during the execution of a single rule, it should be declared as a variable local to the rule.

Rules in *Mur φ* are executed atomically. There is no limit on how many statements can appear in a single *Mur φ* rule. As long as the behavior of the model is not affected, i.e. the operations performed by these statements are independent of the operations in the other rules, the programmer should put as many statements in one rule as possible.

Another modeling guideline for *Mur φ* is to reset dead variables. Consider a variable that is live across sev-

eral rules after which its value is not used; i.e. it becomes dead. In this case, the *Mur φ* designers suggest that the programmer use the `CLEAR` statement to set the dead variable to a default value, thereby avoiding duplicate counting of the same system state.

Mur φ also has several built-in space-reduction methods, such as symmetry reduction, reversible rules, and repetition constructors. However, we did not find occasion to use these techniques in our case study.

- **Spin:** In *Spin* local variables of processes are part of system states. Therefore when modeling a system in *Spin*, one should minimize both the number of global variables and local variables. *Spin* also supports optimization techniques that allow automatic merging of sequences of statements into atomic steps and dead-variable elimination.
- **XMC:** In *XMC* one should also minimize the number of variables used. *XMC*'s optimizing compiler performs automatic merging of sequences of transitions into atomic steps and dead-variable elimination.

4.6 Comparison of the Tools

As can be gleaned from the above discussion, the tools used in our case study follow different approaches to modeling concurrent systems, specifying properties, and checking models for properties. Table 1 summarizes the main features of each of the four tools.

As another point of comparison, it is worthwhile to consider the work involved in extracting a model of the i-protocol from the original C code using the modeling languages of the four tools. Actually, the first model of the protocol we produced was specified using VPL, the input language of the Concurrency Factory, and took a Post-Doc six months of full-time effort. Subsequent versions were easier to produce as we became more familiar with the protocol and ways of modeling it.

The i-protocol was probably easiest to model in the Concurrency Factory and *Spin*, as both of these tools are targeted toward asynchronous systems, and their input languages provide data structuring, complex control statements and typed communication channels. As a result, the Concurrency Factory and *Spin* models of the sender and receiver processes can be regarded as more or less direct translations of the protocol's C code.

Mur φ and *XMC* are also designed for verifying asynchronous systems, but modeling the i-protocol in these tools requires more effort: in *Mur φ* the protocol's behavior must be captured using nondeterministic guarded commands, and in *XMC* using a set of CCS-like expressions. The modeling of the i-protocol in *Cospan* requires yet the most effort, as the input to *Cospan* is essentially a collection of finite-state machines; moreover, *Cospan* is designed for the verification of synchronous systems, with extensions for asynchronous systems.

	Cospan	Mur φ	Spin	XMC
Syntactic building block				
process	✓		✓	✓
transition rule		✓		
Basic statements				
assignment	✓	✓	✓	✓
sequence		✓	✓	✓
nondeterministic branching			✓	✓
channel input/output			✓	✓
iteration		✓	✓	
recursion				✓
Semantic model				
automaton	✓		✓	
transition system		✓		✓
State representation				
valuation of variables	✓	✓	✓	✓
control points			✓	✓
channel status			✓	
Modeling of nondeterminism				
interleaving processes			✓	✓
nondeterministic branching statement			✓	✓
nondeterministic assignment	✓			
multiple enabled rules		✓		
Modeling of concurrency				
syntactic support	✓		✓	✓
simulation		✓		
synchronous system	✓	✓		
asynchronous system		✓	✓	✓
Modeling of communication				
channels			✓	✓
shared variables	✓	✓		
Property specification				
deadlock	✓	✓	✓	✓
good/bad cycle	✓		✓	
assertion		✓	✓	
temporal logic formula		LTL	LTL	mu-calc.

Table 1: Tool comparison.

5 Abstraction Techniques

A main point of this case study is to demonstrate the utility of abstraction techniques on verifying complex systems like the i-protocol. For each tool considered, we started with a specification of the i-protocol that was culled directly from the C code. We then identified four behavior-preserving abstraction steps that we applied to each of the original specifications to obtain successively more abstract versions. The techniques we deploy were first used on the i-protocol in [Hol99]. That paper considered the effect of these abstractions on the performance of Spin only.

The four steps applied to the four original specifications yielded a total of 18 specifications, including the originals. The number would have been 20 except for the fact that the first abstraction step was not applicable to the original Mur φ and XMC specifications. We now de-

scribe the abstraction steps we took in terms of what we call *abstraction levels*.

Level 0: This level corresponds to two of the four original specifications of the protocol, namely the ones for Cospan and Spin. The underlying principle in producing all of the original specifications was that they be as faithful to the C code as possible. This provided us with a common ground on which to derive an initial specification in the input language of each tool. The initial Cospan and Spin specifications differ from the initial Mur φ and XMC specifications in that the former use source and sink processes to explicitly model upper-layer entities. This point is discussed further in the context of abstraction level 1.

Level 1: A *source* process is a process whose sole purpose is to generate a sequence of predetermined messages, while a *sink* process merely consumes such messages [Hol98]. Our level-0 specifications of the i-protocol for Cospan and Spin used source and sink

processes to explicitly model the protocol’s upper-layer entities. The level-1 abstraction eliminated source and sink processes from these specifications. As described in [Hol98], this can be done without changing the behavior of the model in question as follows. Consider the source process. Its purpose was to model the producer entity of the upper layer and it did this by generating a sequence of messages (data packets) to be consumed by the protocol’s sender process. At abstraction level 1, the producer is deleted and the sender process itself generates the data packets it sends out on the medium connecting it with the receiver process. The correctness of this abstraction critically depends on the fact that the sender was the *only* process reading messages from the producer. In this case, it is unambiguously clear which process should assume the responsibility for generating data packets after the producer is deleted, namely, the sender process. The situation is exactly analogous for the deletion of the sink process.

The initial Mur ϕ and XMC specifications, which we place at abstraction level 1, did not contain source and sink processes as these tools support the specification of “open systems”. This allows one to model the interaction between a system and its environment implicitly. To better understand this form of modeling, consider, for example, the XL specification of the i-protocol. XL, the input language of XMC, is based on process algebra. As such, to model, say, the interaction between the sender process of the protocol and the producer entity at the upper level, it suffices to equip the sender with a port called `in` linking it with the producer, which is considered an external entity to the specification and never specified explicitly. The formal semantics of process algebra is such that the model’s behavior will include state transitions corresponding to receiver-producer interactions. Moreover, such interactions are observable at the “top level” of the specification, meaning that they can appear in mu-calculus formulas to be model checked by the system.

Level 2: The C code of the i-protocol’s sender and receiver processes was written in such a way that the main control structure consisted of a pair of nested do-loops; moreover, several of the actions taken by the protocol in the inner loop were also options in the outer loop (e.g. receive a packet from the medium, timeout). Since all four initial specifications were faithful to the C code, they too were structured in this manner. While in no way implying that there was anything unreasonable about the C code—in all likelihood, it reflected the author’s view of the control logic—specifications structured in this way tended to yield state-space sizes that were larger than need be. The main purpose of the level-2 abstraction was to replace the nested-loops control structure with a flattened, single do-loop control structure. This also pre-

sented some opportunities, e.g. in the *Cospan* model, to group together sequences of local computations into atomically executable units, further reducing the size of the state space.

Level 3: In this abstraction, the header checksum field and the data checksum field are deleted from the model. In particular, the state transitions in the medium processes corresponding to the transmission of packets with bad checksums are removed. The removal of the transitions from the medium processes modeling the transmission of packets with bad *header* checksums does not affect the behavior of the sender or the receiver, since such packets were simply discarded (without changing control state) by these processes anyway. On the other hand, the removal of the transitions from the medium processes corresponding to a bad *data* checksum is compensated for by adding a transition to the receiver process that models the receipt of a packet with a bad data checksum. This newly added transition can be nondeterministically selected by the receiver while waiting for a packet from the medium. Such a transition is not added to the sender process since, in our model of the i-protocol, data flow is unidirectional from the sender to the receiver; thus the sender never receives a data packet.

Level 4: In this abstraction, the state transitions modeling the dropping of messages are moved from the medium processes to the sender and receiver processes. Specifically, at level 4, after receiving a message, the receiver or the sender may simply choose to ignore it.

6 Model-Checking Results

We have obtained model-checking results for eight versions of the i-protocol, obtained by adjusting the values of three parameters:

1. A window size of either one or two is used. We distinguish these two cases as `1--` and `2--`.
2. The possibility of message corruption by the communication media is either captured in the model or not (although in either case the media can still potentially drop messages). We distinguish these two cases as `-f-` (*full*) and `-m-` (*mini*), respectively.
3. The patch identified in Section 3 to prevent the livelock error is enabled or disabled. We distinguish these two cases as `--f` (*fixed*) and `--n` (*non-fixed*), respectively.

The eight cases can then be identified as `1mf`, `1mn`, `2mn`, `2mf`, `1ff`, `1fn`, `2fn`, and `2ff`. Clearly, case `1mn` will be the easiest, in terms of state-space complexity, and `2ff` the most difficult. This naming scheme for the cases was first proposed in [Hol99].

As suggested in [Hol99], it is also useful to partition the eight cases into two groups: the *non-fixed* cases (1mn, 2mn, 1fn, 2fn) and the *fixed* cases (1mf, 2mf, 1ff, 2ff). The reason for this distinction is that for the fixed cases, the protocol will be livelock-free and determining this fact will require an exhaustive search of the protocol’s entire state space. For the non-fixed cases, we allow the tools to halt execution as soon as the livelock error is detected. Although tool performance in these cases is somewhat unpredictable due to uncontrollable variations in search order, they serve to illustrate the benefits of *local model checking*, where the state-space search is goal-directed and guided by the structure of the formula being checked.

For each of the eight cases considered above, we present performance data for four tools: *Cospan* (version 8.23.120), *Murφ* (version 2.7L), *Spin* (version 3.3.4), and *XMC* (version 1.0). Moreover, results are given for each of the five abstraction levels defined in Section 5. All told, this adds up to 144 runs of one verification tool or another.¹⁰ All results were obtained on a Sun Ultra-Enterprise workstation with two 336 MHz CPUs and 2 GB of RAM.

Performance data for the Concurrency Factory verification tool suite is not provided. Although the Factory was the tool we first used to detect and diagnose livelock in the i-protocol, and it was able to do this for both window sizes 1 and 2, its performance turned out not to be competitive with the other model checkers.

The results for each tool are presented in Tables 2-5, respectively. A word of warning: for identical models, encoded equivalently in the input language of each tool, we would expect the number of reachable states explored by each tool to be approximately the same. As can be seen in the tables, this is not the case. Since tool performance is directly related to model complexity, it would be incorrect to draw conclusions about relative tool performance from these data. Rather, the differences in performance are more closely related to differences in model complexity, the latter of which can be reconciled to a large degree using the abstraction techniques investigated here.

We organize our discussion of the data by abstraction level. Consider first abstraction level 0, where, for each tool, we encoded the protocol in a manner as faithful as possible to the C code. As discussed in Section 5, only the original *Cospan* and *Spin* encodings utilize source and sink processes, explicitly modeling the upper-layer entities. Since the abstraction taking us from level 0 to level 1 removes such processes and is therefore applicable only to the original *Cospan* and *Spin* encodings, we present level-0 performance data for these tools only; As noted earlier (Section 4.1), we obtained results for *Cospan* using both *Cospan*’s symbolic analysis algorithm and explicit-state algorithm. In Table 2, the data from *Cospan*’s symbolic analysis algorithm is presented. When the sym-

bolic analysis algorithm is used, *Cospan* does not provide information on the number of transitions contained in the model. The data from *Cospan*’s explicit-state algorithm is presented in Table 6. We present the data from *Cospan*’s symbolic analysis algorithm first, which reflects an intentional bias on our part toward *Cospan*’s symbolic analysis algorithm. This is because this algorithm scales better than the explicit-state one in the presence of fast-growing state spaces.

A trend established at abstraction level 0 and maintained at the higher abstraction levels is that resource consumption (time and space) for all four tools is almost always significantly lower in the non-fixed cases. This is because we allow the tools to terminate the state-space search as soon as a livelock is detected, a form of local model checking. In contrast, a global model checker, by definition, requires construction of the full state space in advance of the actual verification. In this case, there would be little difference in performance between the non-fixed and fixed cases.

At abstraction level 1, the impact of removing source and sink processes on *Cospan* and *Spin*’s performance is evident. The data indicates that for *Spin* this abstraction reduces the state-space size, and hence the memory usage and running times, by a factor of about two. For *Cospan*, however, only a nominal reduction in state-space size is witnessed. This discrepancy can be attributed to the fact that the processes representing the upper-layer entities are encoded as synchronous “passive processes” in the *Cospan* model and as asynchronous processes in the *Spin* model. In a synchronous model, passive processes add very little to the size of the global state space. As discussed below, another opportunity to remove asynchronous source/sink processes from the *Spin* model presents itself at abstraction level 4. This again results in an appreciable decrease in model complexity.

Abstraction level 2, which consisted of flattening the sender and receiver’s control-loop structure and merging control states, has a significant effect on all tools. Reductions in state-space sizes range from approximately 2 to 11.

The abstraction at level 3 consisted of removing header and data checksums from the model. This again leads to marked reductions in the size of the i-protocol’s state space. The data given in Table 3 shows that *Spin* benefits the most from this abstraction, with the size of the state space going down by a factor of about 10. In *Murφ* the state space is reduced by a factor of 6, while in *XMC* it is reduced by a factor of 3. *Cospan* is the least sensitive to this abstraction, because the header and data checksums are modeled as selection variables, whose values are not stored in the global states. In particular, for the mini versions, the header and data checksums are always set to true, and *Cospan* is thus able to automatically eliminate the selection variables representing the checksums. Thus the level-3 abstraction has no effect on the size of the state space for the mini versions in *Cospan*.

¹⁰ There would have been a total of 160 runs but, for reasons explained in Section 5, the level-0 abstraction was not applicable to the *Murφ* and *XMC* specifications.

Version	Abstraction Level	States	Memory(MB)	Time(sec)
1mn	0	14.4K	3.8	6.8
	1	13.9K	3.8	4.4
	2	1039	3.2	0.5
	3	1039	3.2	0.6
	4	626	3.2	0.5
2mn	0	97.0K	10.4	127.0
	1	93.2K	14.4	83.2
	2	33.7K	6.9	13.2
	3	33.7K	6.9	15.9
	4	24.9K	5.3	4.4
1fn	0	92.2K	7.4	16.3
	1	91.6K	7.4	17.3
	2	1780	3.2	0.7
	3	1590	3.2	0.7
	4	1087	3.2	0.6
2fn	0	858.3K	25.8	310.8
	1	847.9K	25.3	300.1
	2	353.7K	15.9	116.9
	3	290.7K	15.3	91.1
	4	239.9K	10.6	32.7
1mf	0	24.2K	5.3	11.5
	1	23.6K	5.3	12.2
	2	1956	3.2	0.7
	3	1956	3.2	0.7
	4	1393	3.2	0.5
2mf	0	142.1K	14.9	174.0
	1	137.1K	14.8	159.0
	2	55.8K	9.5	55.7
	3	55.8K	9.5	35.6
	4	45.2K	9.0	11.9
1ff	0	163.6K	8.5	31.8
	1	162.5K	8.0	26.9
	2	3169	3.2	0.8
	3	2730	3.2	0.8
	4	1963	3.2	0.6
2ff	0	1.34M	31.7	627.2
	1	1.32M	34.7	588.3
	2	578.6K	18.1	184.5
	3	448.2K	17.1	125.8
	4	354.0K	15.3	55.0

Table 2: Results for Cospan

Abstraction level 4 moves the modeling of dropped messages from the medium processes to the sender and receiver processes. For the *Spin* model this reduces the two medium processes to source-sink processes, which can now be removed without loss of generality. Since there are more interprocess synchronizations in the model after this abstraction (i.e. the choice of nondeterministically dropping a message is made later in an execution sequence), the size of the state space is reduced. Indeed, *Spin*, *Cospan*, and *Mur ϕ* all exhibit significantly smaller state spaces. The size of the state space for *XMC*, however, increases by about 20% (24.7K states to 29.5K states). Additional experimentation revealed that the increase is due to the presence of an externally observ-

able transition in the level-4 specification that prevents *XMC*'s optimizing compiler from performing transition merging in the context of this transition.

The results show that each of the model checkers we considered can be used to find the error in the protocol within plausible resource limits. They also illustrate convincingly that abstraction techniques can significantly improve the performance of any verification tool.

In modeling the *i*-protocol, we consulted with the inventors of each of the tools we used in the case study in order to optimize our chances of producing the most efficient model as possible (in terms of state-space size). We also made every effort to ensure that the models are consistent across the four tools. This includes the use

Version	Abstraction Level	States	Transitions	Memory(MB)	Time(sec)
1mn	0	4073	8578	0.6	0.3
	1	1990	4091	0.3	0.1
	2	1023	1985	0.2	0.06
	3	706	1360	0.1	0.04
	4	251	546	0.1	0.02
2mn	0	22.8K	49.7K	3.0	1.6
	1	9185	19.0K	1.1	0.6
	2	1826	2526	0.3	0.08
	3	976	1481	0.2	0.04
	4	500	1105	0.1	0.02
1fn	0	851	1311	0.2	0.1
	1	611	877	0.2	0.1
	2	10.7K	23.1K	0.9	0.52
	3	1590	3727	0.2	0.1
	4	328	709	0.1	0.01
2fn	0	23.4K	43.3K	3.3	1.6
	1	4516	7962	0.6	0.2
	2	7966	17.8K	0.8	0.39
	3	3216	6957	0.3	0.18
	4	380	759	0.1	0.01
1mf	0	126.7K	383.5K	11.8	10.8
	1	62.5K	186.1K	5.1	4.5
	2	28.6K	79.1K	1.8	1.8
	3	23.6K	65.7K	1.6	1.3
	4	1565	3.8K	0.2	0.9
2mf	0	126.7K	383.5K	11.8	10.8
	1	1565	3.8K	0.1	0.1
	0	797.3K	2.33M	76.4	66.0
	1	459.6K	1.35M	37.7	35.9
	2	165.8K	450.4K	11.0	10.0
1ff	0	2.13M	7.10M	180.3	189.9
	1	971.7K	3.08M	75.4	70.7
	2	392.7K	646.6K	21.4	24.5
	3	27.8K	78.8K	1.8	1.8
	4	1654	4.0K	0.1	0.1
2ff	0	18.8M	62.1M	1708.4	1770.9
	1	9.48M	30.2M	752.8	773.5
	2	3.20M	8.91M	194.9	200.4
	3	272.5K	758.6K	17.6	16.6
	4	8950	21.3K	0.6	0.4

Table 3: Results for Spin

of identical sets of control points and variables whenever possible. However, the data still shows that there are differences in the number of states computed by the different tools.

The variation in the state counts can be largely explained by differences in modeling formalisms and optimization techniques supported by each tool. The input language S/R of *Cospan* is a data-flow language, and *Cospan* is a tool for synchronous systems. Thus abstractions such as removing source and sink processes and merging control states have relatively less effect on the state counts computed by *Cospan*. Furthermore, in order to capture interleaving in a *Cospan* model, a non-deterministic scheduler must be introduced into the S/R

program for the i-protocol, which significantly increases the state count reported by *Cospan*.

The other three tools, *Spin*, *Murφ* and *XMC*, are all tools for asynchronous systems. *XMC* contains an optimizing compiler which performs partial evaluation of transition rules and statement merging. This leads to a low state count for *XMC* at abstraction level 1. On the other hand, *XMC* is based on labeled transition systems and the livelock property is expressed as a modal mu-calculus formula that refers to output labels representing certain actions performed by the i-protocol. Thus, additional output transitions with labels must be introduced into the *XMC* model. Each such transition introduces an additional local control point in the component where

Version	Abstraction Level	States	Transitions	Memory(MB)	Time(sec)
1mn	1	2104	6092	0.1	0.18
	2	380	1216	0.1	0.04
	3	380	1216	0.1	0.03
	4	158	376	0.1	0.02
2mn	1	27.1K	80.2K	0.5	2.25
	2	2800	8896	0.1	0.25
	3	2800	8896	0.1	0.25
	4	1100	2572	0.1	0.11
1fn	1	18.7K	71.8K	0.3	1.61
	2	2192	9188	0.1	0.23
	3	464	1608	0.1	0.05
	4	190	492	0.1	0.03
2fn	1	348.6K	1.32M	5.6	37.31
	2	28.5K	117.3K	0.5	2.58
	3	5136	17.0K	0.1	0.54
	4	1956	4644	0.1	0.25
1mf	1	4068	16.5K	0.2	0.64
	2	856	3998	0.1	0.16
	3	856	3998	0.1	0.16
	4	304	1100	0.1	0.07
2mf	1	47.7K	20.0K	2.0	8.38
	2	5456	25.3K	0.2	1.05
	3	5456	25.3K	0.2	0.93
	4	1820	6464	0.1	0.36
1ff	1	39.3K	181.4K	1.7	7.19
	2	5868	31.7K	0.2	1.02
	3	996	4930	0.1	0.22
	4	352	1378	0.1	0.09
2ff	1	636.0K	2.94M	26.8	126.32
	2	58.5K	311.4K	2.5	11.32
	3	9016	43.4K	0.4	2.13
	4	3000	10.9K	0.1	0.74

Table 4: Results for Mur φ

the transition is located, which in turn leads to an increase in the number of reachable global states. The percentage of increase depends on the total number of control points in the components, as well as the amount of additional interleaving the output transitions introduce. For systems with a small number of control points, the effect of these additional output transitions is more pronounced. This explains why the state count computed by XMC is larger than those computed by Spin and Mur φ at abstraction level 4.

Spin supports automatic statement-merging as one of its optimization techniques. In contrast, the Mur φ model of the i-protocol is a collection of condition/action rules and the relatively primitive nature of this modeling notation allows certain optimizations, such as a form of statement merging, to be performed by hand. At earlier levels of abstraction, our models of the i-protocol tend to be more complex, and the effect of the hand-optimizations in Mur φ is more obvious. Thus, at earlier levels of abstraction, the state counts computed by Mur φ are lower than those computed by Spin. At abstraction

level 4, however, the state counts computed by Spin and Mur φ are very close.

6.1 Symbolic vs. Explicit-State Model Checking in Cospan

The Cospan data presented in Table 2 were obtained using command-line options “-bq1” the first of which causes Cospan to invoke its BDD-based symbolic analysis routine. Table 6 documents Cospan’s performance at abstraction levels 3 and 4 with only the “q1” option in effect. The absence of the “-b” option will cause the tool to use explicit-state checking. The following observations are noteworthy. First, unlike in the symbolic case, Cospan reports the number of state-space transitions in the explicit-state case. Secondly, the explicit-state algorithm produces exactly the same state count as those output by the symbolic algorithm in the fixed cases. Thirdly, the explicit-state algorithm uses noticeably less space than the symbolic algorithm in all but one case (2ff, abstraction level 3).

Version	Abstraction Level	States	Transitions	Memory(MB)	Time(sec)
1mn	1	436	733	0.6	0.13
	2	112	206	0.6	0.04
	3	112	206	0.6	0.03
	4	163	243	0.4	0.04
2mn	1	1993	3283	1.3	0.52
	2	301	562	0.7	0.07
	3	309	577	0.7	0.06
	4	1348	2304	0.8	0.28
1fn	1	238	341	0.5	0.04
	2	206	375	0.6	0.04
	3	122	216	0.6	0.02
	4	174	254	0.4	0.03
2fn	1	792	1146	0.9	0.14
	2	615	1049	0.8	0.12
	3	349	617	0.7	0.08
	4	1470	2426	1.0	0.31
1mf	1	4512	15.3K	7.2	3.73
	2	584	1564	1.2	0.34
	3	584	1564	1.2	0.31
	4	884	1912	1.4	0.37
2mf	1	31.4K	106.4K	57.8	33.97
	2	14.7K	39.5K	14.5	10.34
	3	15.2K	40.9K	15.3	10.44
	4	19.9K	43.2K	16.7	11.40
1ff	1	14.0K	52.2K	23.0	12.87
	2	2070	6058	2.7	1.22
	3	886	2416	1.5	0.47
	4	1262	2744	1.7	0.57
2ff	1	106.5K	396.8K	190.0	137.27
	2	57.0K	167.1K	58.2	49.49
	3	24.7K	68.9K	25.9	18.53
	4	29.5K	64.3K	25.4	18.87

Table 5: Results for XMC

Version	Abstraction Level	States	Transitions	Memory(MB)	Time(sec)
1mn	3	229	830	0.2	0.1
	4	150	501	0.2	0.1
2mn	3	936	3426	0.2	0.3
	4	679	2309	0.2	0.2
1fn	3	266	972	0.2	0.1
	4	176	601	0.2	0.1
2fn	3	1794	6245	0.2	1.3
	4	1246	4118	0.2	0.9
1mf	3	1956	8875	0.2	0.8
	4	1393	5843	0.2	0.6
2mf	3	55.8K	252.1K	2.4	27.4
	4	45.2K	198.2K	2.0	22.3
1ff	3	2730	11.9K	2.4	2.4
	4	1963	7897	0.2	1.7
2ff	3	448.2K	1.90M	18.2	437.8
	4	354.0K	1.44M	14.4	346.0

Table 6: Results from Cospan’s explicit state-enumeration algorithm (abstraction levels 3 and 4).

Finally, the explicit-state algorithm is faster in all eight non-fixed cases and one other case (2mf, abstraction level 3). This is because the BDD-based algorithm—which constructs a symbolic representation of the entire state space thereby making it a global algorithm—suffers from significantly more overhead in the non-fixed cases where the verification can be halted early. On the other hand, the explicit-state algorithm is significantly slower than the symbolic algorithm in the 2ff case (by a factor of more than 6 at abstraction level 4). This indicates that the performance of the symbolic algorithm is likely to scale better than the explicit-state one in the presence of fast-growing state spaces.

7 Conclusions

We have presented a case study on explicit-state model checking based on the i-protocol, an optimized sliding-window protocol for GNU uucp. The study involved the application of four verification tools, *viz.* *Cospan*, *Mur φ* , *Spin*, and *XMC*, each of which supports some form of explicit-state model checking. In the process of conducting the study, we identified tool-specific modeling guidelines and run-time options, as well as tool-agnostic modeling abstractions. Collectively, these techniques allowed us to effectively and efficiently model check the i-protocol, despite the protocol’s inherent complexity. We moreover believe that many of these techniques will be applicable to a broad class of protocols.

In terms of future work, other properties of the i-protocol should be checked besides the absence of livelock, such as deadlock freedom and eventual message delivery. It would also be interesting to conduct a *parametric* analysis of livelock in the i-protocol: the results we have presented are limited to fixed window sizes of 1 and 2, mainly to avoid state-space explosion. By making the window size a parameter of the analysis, the question of livelock in the i-protocol could be answered for all possible window sizes. Note that such an analysis would need to take into account the constraint that $W \leq \lfloor SEQ/2 \rfloor$, where W is the window size and SEQ is the size of the message sequence space, if the basic abstraction of the protocol’s message sequence space described in Section 2 is used.

Acknowledgments We are grateful to the anonymous referees for their valuable comments and suggestions; their feedback greatly improved the quality of the paper. We would also like to thank Bob Kurshan for assisting us in applying *Cospan* to the i-protocol. Finally, thanks to Y.S. (Ramki) Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Oleg Sokolsky, Gene Stark and David S. Warren for their valuable contributions to the i-protocol case study. Ramki, in particular, produced the initial working VPL specification of the protocol and paved the way for many of the results found in this paper.

References

- [AH96] R. Alur and T. A. Henzinger, editors. *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, July 1996. Springer-Verlag.
- [CCA96] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. Experimental design for comparing static concurrency analysis techniques. Technical Report 96-084, Computer Science Department, University of Massachusetts at Amherst, 1996.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
- [CLSS96] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In Alur and Henzinger [AH96], pages 398–401.
- [CM88] K. M. Chandy and J. Misra. *Parallel Program Design — A Foundation*. Addison-Wesley, 1988.
- [Cor96] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.
- [CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.
- [D⁺00] Y. Dong et al. i-Protocol Case Study Web Site. <http://www.cs.sunysb.edu/~lmc/iproto/>, 2000.
- [DDR⁺99] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I.V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-Protocol: A comparative study of verification tools. In *Tools and Algorithms for the Construction and Analysis of Algorithms (TACAS '99)*, Lecture Notes in Computer Science, Amsterdam, March 1999. Springer-Verlag.
- [Dil96] D. L. Dill. The Mur φ verification system. In Alur and Henzinger [AH96], pages 390–393.
- [DR99] Y. Dong and C.R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proceedings of FORTE/PSTV '99*, 1999.
- [EC81] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings of the Seventh International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, Berlin, 1981. Springer-Verlag.
- [HD93] A. Hu and D. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In C. Courcoubetis, editor, *Computer Aided Verification (CAV '93)*, volume 693 of *Lecture Notes in Computer Science*, pages 3–14, Elounda, Greece, June 1993. Springer-Verlag.
- [HHK96] R. H. Hardin, Z. Har’El, and R. P. Kurshan. COSPAN. In Alur and Henzinger [AH96], pages 423–427.
- [Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

- [Hol98] G. J. Holzmann. Designing executable abstractions. In *Proceedings of Workshop on Formal Methods in Software Practice*, Clearwater Beach, FL, March 1998. ACM Press.
- [Hol99] G. J. Holzmann. The engineering of a model checker: The Gnu i-protocol case study revisited. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
- [Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
- [QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.
- [RRR⁺97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.
- [RS97] Y. S. Ramakrishna and S. A. Smolka. Partial-order reduction in the weak modal mu-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR '97)*, volume 1243 of *Lecture Notes in Computer Science*, Warsaw, Poland, July 1997. Springer-Verlag.
- [Spi] Spin Web Site. <http://netlib.bell-labs.com/netlib/spin/whatispin.html>.
- [Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B*. Elsevier Science Publishers, 1990.
- [VW86] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)*, pages 332–344, Cambridge, Massachusetts, June 1986. Computer Society Press.
- [Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, St. Petersburg, January 1986.
- [XSB99] XSB. The XSB logic programming system v2.01, 1999. Available by anonymous ftp from <ftp.cs.sunysb.edu>.