Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools*

Yifei Dong, Xiaoqun Du, Y.S. Ramakrishna**, C.R. Ramakrishnan, I.V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky***, Eugene W. Stark, and David S. Warren

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

Abstract. The i-protocol, an optimized sliding-window protocol for GNU UUCP, came to our attention two years ago 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 repeated this verification effort with five widely used model checkers, namely, COSPAN, Murφ, SMV, SPIN, and XMC. It is our contention that the i-protocol makes for a particularly compelling case study in protocol verification and for a formidable benchmark of verification-tool performance, for the following reasons: 1) The i-protocol can be used to gauge a tool's ability to detect and diagnose livelock errors. 2) The size of the i-protocol's state space grows exponentially in the window size, and the entirety of this state space must be searched to verify that the protocol, with the livelock error eliminated, is deadlock- or livelock-free. 3) The i-protocol is an asynchronous, low-level software system equipped with a number of optimizations aimed at minimizing control-message and retransmission overhead. It lacks the regular structure that is often present in hardware designs. In this sense, it provides any verification tool with a vigorous test of its analysis capabilities.

1 Introduction

Model checking [CGP99] is a verification technique aimed at determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in real-life systems. An interesting account of a number of these success stories can be found in [CW96].

In this paper, we report on our experience in using model checking—as provided by six widely used verification tools—to detect and correct a non-trivial livelock in a bidirectional sliding-window protocol. The tools in question are the Concurrency Factory [CLSS96], COSPAN [HHK96], Murφ [Dil96],

* Research supported in part by NSF Grants CCR-9505562 and CCR-9705998, and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

** Currently at: Sun Microsystems, Mountain View, CA 94043, USA.

*** Currently at: Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia, PA 19104, USA.

SMV [CMCHG96], Spin [HP96], and XMC [RRR+97], each of which supports some variety of model checking.

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 is part of a 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 rather 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 Stark, 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 this abstraction of the protocol in VPL (Value Passing Language), the textual specification language of the Concurrency Factory specification and verification toolset.

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 reestablished. 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 acknowledgements are handled. The livelock error was fixed independently by Ian Taylor, the i-protocol's original developer, in GNU UUCP version 1.05.

We repeated our model-checking-based verification of the i-protocol with the COSPAN, Murφ, Spin, SMV, and XMC verification tools, so that we could draw some comparisons between these tools on a real-life protocol. The i-protocol is particularly compelling as a case study in protocol verification and as a verification-tool performance benchmark 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 will need to be searched to verify that the protocol, with the livelock error eliminated, is deadlock- or livelock-free. Finally, the i-protocol is an asynchronous, low-level software system equipped with a number of optimizations aimed at minimizing control-message and retransmission overhead. It lacks the regular structure that is often present in hardware designs. In this sense, it provides any verification tool with a vigorous test of its analysis capabilities.

Our experimental results show that the special-purpose cycle-detection algorithms of Spin and COSPAN can be used to significant advantage to check for livelocks in complex systems like the i-protocol. SMV exhibited excellent memory-usage performance on all runs of window size 1, but failed to complete in a reasonable amount of time on any run of window size 2. This can most likely be attributed to exponential blowup in the BDD representation for window sizes greater than 1. Murφ and XMC performed the best on the i-protocol. In the case of Murφ this is due to the low-level nature of its specification language (guarded commands) and the succinct manner in which system states are encoded. XMC's strong performance is a consequence of the efficiency of the underlying tabled logic programming system, XSB [XSB97], and our use of partial evaluation to specialize the logical formula capturing livelock to the i-protocol's behavior. Our model-checking results are described more fully in Section 5 (see Table 1).

Although the Concurrency 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 CPU time usage was in general significantly higher in comparison with the other model checkers. The new release of the Factory, planned for January 1999, uses a more sophisticated scheme for encoding value-passing behavior of processes. We expect its performance to be on par with the other tools.

In related work, [CCA96,Cor96] benchmark the performance of a variety of model checkers (including SMV and Spin) on Ada tasking programs. The major differences between our study and theirs is in the application domain (a real-life communication protocol vs. a suite of concurrency analysis benchmark programs) and in the type of properties considered (livelock vs. reachability).

The remainder of the paper is organized as follows. Section 2 describes the salient features of the tools used in this case study. Section 3 gives a detailed account of the i-protocol, with an emphasis on how we modeled the protocol for verification purposes. Section 4 describes the livelock that we discovered, and shows how a small change to the protocol effectively eliminates this form of livelock. Section 5 summarizes the results of our model-checking experiments, and offers a comparison of the tools' performance. Section 6 contains our concluding remarks.

We have constructed a web site (`http://www.cs.sunysb.edu/~lmc/iproto/`) to serve as a central repository for our results. The site contains the source code of version 1.04 of the i-protocol, the patch to the C code to fix the livelock error, the encoding of the protocol in each of the input languages of the six tools, and various performance statistics generated by our benchmarking activity. For each tool, these include the number of states explored, number of transitions traversed, CPU time usage, and memory usage (see Table 1).

2 The Verification Tools

In this section, we describe the most salient features of the tools we used in our analysis of the i-protocol.

2.1 The Concurrency Factory

In the context of our case study, the main features of the Concurrency Factory [CLSS96] are its textual specification language, VPL, and its local model checker for the modal mu-calculus [RS97]. VPL-supported data structures include integers of limited size and arrays and records composed of such integers. A system specification in VPL is a tree-like hierarchy of subsystems. A subsystem is either a *network* or a *process*. A network consists of a collection of subsystems running in parallel and communicating with each other through typed channels. Simple statements of VPL are assignments of arithmetic or boolean expressions to variables, and input/output operations on channels. Complex statements include sequential composition, `if-then-else`, `while-do`, and nondeterministic choice in the form of the `select` statement.

LMC, the Factory's local model checker, computes in an on-the-fly fashion the product of a graph representation of the formula to be checked with the labeled transition system (guaranteed to be finite-state) underlying the VPL program. The number of nodes of the product graph explored by LMC is further minimized through the use of partial-order reduction. This technique eliminates from consideration those portions of the state space resulting from redundant interleavings of independent events. LMC is also equipped with diagnostic facilities that allows the user to request that the contents of the depth-first search stack be displayed whenever a certain "significant event" occurs (e.g. when the search first encounters a state at which a logical variable is determined to be either true or false) and to play interactive games for the full modal mu-calculus.

2.2 COSPAN

COSPAN [HHK96] is a model checker for synchronous systems based on the theory of ω-automata. 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 consists of the automata language containment test $\mathcal{L}(P) \subset \mathcal{L}(T)$. P is typically given as the synchronous parallel composition of component processes, specified as ω-automata. Asynchronous composition can be modeled through nondeterministic delay in the components.

Language containment can be checked in COSPAN using either a symbolic (BDD-based) algorithm or an explicit state-enumeration algorithm. Both algorithms are "on-the-fly." COSPAN also supports a notion of "recur edge" and can check whether in every execution of the system the recur edge occurs infinitely often. We used this facility to detect livelock in the i-protocol.

Systems can be specified in COSPAN using the S/R language, which supports nondeterministic, conditional (i.e., `if-then-else`) variable assignments; variables of type bounded integer, enumerated, Boolean, and pointer; arrays and records; and integer and bit-vector arithmetic. Modular hierarchy, scoping, parallel and sequential execution, homomorphism declaration and general ω-automaton fairness are also available. COSPAN also provides an error tracing facility that allows the user to back-reference from the error track to the S/R source.

2.3 Murφ

The Murφ verification system consists of the Murφ compiler and the Murφ description language. The Murφ compiler generates a special-purpose verifier from a Murφ description. The Murφ description language uses a set of iterated guarded commands, like Chandy and Misra's Unity language [CM88]. A Murφ description consists of constant and type declarations, variable declarations, procedure declarations, rule definitions, a description of the start state, and a collection of invariants. Each rule is a guarded command consisting of a condition and an action. The condition is a boolean expression and the action is a sequence of statements. An invariant is a boolean expression that is desired to be true in every state. When an invariant is violated, an error message and error trace are generated.

Murφ is able to verify liveness specifications written in a subset of Linear Time Temporal Logic (LTL). Liveness specifications are expressed using keywords `ALWAYS`, `EVENTUALLY`, and `UNTIL`, and are checked under the assumption that every rule is weak-fair (unless declared otherwise). We used this facility of Murφ to encode and check for livelock in the i-protocol.

2.4 SMV

SMV [CMCHG96] is an automatic tool for model checking CTL formulas. CTL can also be used to specify simple fairness constraints. The transition relation of the system to be verified is represented implicitly by boolean formulas, and implemented by BDDs. This allows SMV to verify models having more than 10^{20} states. SMV also has a diagnostic facility that produces a counterexample when a formula is not true.

An SMV program can be viewed as a system of simultaneous equations whose solution determines the next state. Asynchronous systems, such as the i-protocol, are modeled by defining a set of parallel processes whose actions are interleaved arbitrarily in the execution of the program. As in Murφ liveness specifications, such as absence of livelock, are given in a form of temporal logic (CTL).

2.5 SPIN

SPIN [HP96] is a model checker for asynchronous systems specified in the language PROMELA. Safety and liveness properties are formulated using 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 explosion, with an efficient approximation method based on bitstate hashing. With a careful choice of hashing functions, the probability of an exhaustive proof remains very high.

Besides being able to specify correctness properties in LTL, the PROMELA specification language includes two types of labels that can be used to define two complementary types of liveness properties: acceptance and progress. We used SPIN's ability to check for this latter type of formula to detect livelock in the i-protocol.

PROMELA is a nondeterministic guarded-command language with influences from Hoare's CSP and the language C. PROMELA includes support for data structures, interrupts, bracketing of code sections for atomic execution, the dynamic creation of concurrent processes, and a variety of synchronous and asynchronous message passing primitives. Message passing is via channels with arbitrary numbers of message parameters.

2.6 XMC

XMC [RRR+97] is a model checker for a value-passing process calculus and the modal mu-calculus. It is written in under 200 lines of XSB tabled Prolog code. XSB [XSB97] 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.

Systems to be verified in XMC are encoded in the XL language, a value-passing language similar in many ways to Milner's CCS. A distinguishing feature of XL is its support for a generalized process prefix operator, which allows arbitrary Prolog terms to appear as prefixes. This construct allows the XL programmer to take advantage of XSB's substantial data-structuring facilities to describe sequential computation on values.

Properties such as the possibility of livelock are expressed as modal mu-calculus formulas. The encoding of the semantics of the mu-calculus in XMC can be specialized [JGS93] with respect to a given formula. For the livelock formula used in the verification of the i-protocol, specialization yields a logic program that implements an efficient cycle-detection algorithm, and leads to improved performance.

3 Modeling the i-Protocol

In this section we give a detailed account of the i-protocol, describing how we modeled it for verification purposes. To allow the reader to follow our description of the protocol more closely, we provide (VPL-style) pseudo-code in Appendix A and intermittently refer to line numbers from the pseudo-code.

The i-protocol is a sliding window protocol, but with some optimizations, to be described later, aimed at reducing the acknowledgement and retransmission traffic. The window size, among other "steady-state" protocol parameters, such as data packet size, line quality and error handling parameters, timeout values, acknowledgement high watermarks, and data and message buffer sizes, is decided at the parameter negotiation stage during connection set-up. Since we are concerned with the data transfer properties of the protocol, we do not model the stages involved in connection set-up, parameter negotiation, error and line-quality 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 as consisting of two halves — a sender half that sends data packets to, and receives acknowledgements from, the receiver half on the other side, and a receiver half that receives data packets from, and sends acknowledgements to, the sender half on the other side. To allow for communication latency, the sender can send several packets without waiting for acknowledgements. 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 UUCP, uses a fixed value of $SEQ = 32$, and is intended for window sizes up to, but not exceeding, 16. As discussed below, however, this bound is not essential, and using a sequence space of SEQ, a window size of up to $\lfloor SEQ/2 \rfloor$ can be supported.

To cut down on the acknowledgement traffic, the receiver can piggyback its acknowledgements 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 acknowledgements. 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 SEQ. 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 acknowledgements 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[1].

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

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

upper layer. The variable *lack* records the sequence number up to which an acknowledgement, either explicit (via an ACK) or implicit (via a piggybacked acknowledgement 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).[2] 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* 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 (lines 2–10) consists of busy waiting for one of the following events to occur, and taking appropriate actions as described:

(E1): a packet arrival (line 31) over the communication link (lower layer interface): the packet is first checked for header checksum errors (line 32), and silently discarded (line 33) if it has a header error. Otherwise, if the piggybacked acknowledgement is for a sequence number in the sender's active window, this is used to update (lines 93–96) *rack*. This subsumes the handling of explicit ACK packets (line 39). If the received packet is a NAK for a sequence number in the sender's active window, the requested DATA packet is resent (lines 100–106). If the received packet is a DATA packet, its data checksum is first verified (line 45). If the data is found corrupted, and the packet's sequence number is in the receiver's active window, it has not been previously received, and has not been negatively acknowledged since the previous timeout, then a NAK is sent for that sequence number (lines 47–54). 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 (line 59). 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 (lines 61–66). At each point, *recseq* is appropriately incremented, thus shifting up the active window (lines 58 and 62).

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 (lines 67–73). 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 (lines 74–75), and NAKs generated for all "earlier" missing packets, for which a NAK has not been sent since the last timeout (lines 76–87).

[2] Henceforth, unless explicitly specified otherwise, we shall assume that all arithmetic is modulo SEQ.

(E2): a user request (line 14) to send a new message (upper layer interface): The sender first checks (line 15) if there is an opening in its active window (i.e., that the active window size is less than W). If there is an opening, 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 (lines 22–27). 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 (line 17) for the arrival of a new packet (see (E1) above), or (line 19) for the occurrence of a timeout (see (E3) below).
(E3): a timeout (line 118): The *nakd* buffer is first cleared (line 119), signaling that fresh NAKS may need to be sent out. If there is no packet in the receive buffer (from the lower interface), then the receiver sends a NAK for the "earliest" missing sequence number ($recseq + 1$) in its active window (lines 123–128). Further, the sender resends the "oldest" message (if one exists in its active window), for which it has not received an acknowledgement from the remote (lines 129–134). If, on the other hand, there is a packet available from the lower interface (line 121), we follow (E1) above.

The medium is modeled, in the usual manner, as a lossy FIFO buffer, one for each direction of communication. Each packet has a data and header checksum field, which are nondeterministically reset by the medium to model corruption of the data or header.

Our model of the i-protocol was derived from the C-code of the implementation, and involved a number of abstractions aimed at reducing the protocol's state space. One such abstraction reduces the message sequence space from a fixed value of $SEQ = 32$ (a defined constant in the GNU implementation) to the value $2W$ when using a window size of W. Indeed, with a sequence space of $SEQ = 32$, a system consisting of just the receiver half of the protocol on one side and the sender half of the protocol on the other, connected by a single-buffer communication medium in either direction, would have an estimated state space of about 2.7×10^{14}, even with a window size of 1. In actuality, though, many of these configurations are observationally equivalent [Mil89] to one another, and by using a sequence space of $2W$, this number can be reduced. For instance, for the case $W = 1$, the estimated state space shrinks dramatically to about 1.6×10^7, a reduction by almost a factor of 10^7.

4 Livelock Error

The livelock error detected first using the Concurrency Factory, and subsequently using COSPAN, Murφ, SMV, SPIN and XMC, is illustrated in Figure 1 for the case of $W = 2$, medium buffer capacity 1, and assuming that one side acts as sender and the other as 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, and sends NAK1 and resends DATA1. These (and all subsequent packets) are correctly delivered by the medium. Meanwhile, the receiver also times out, but finding the messages, NAK1, 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 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.

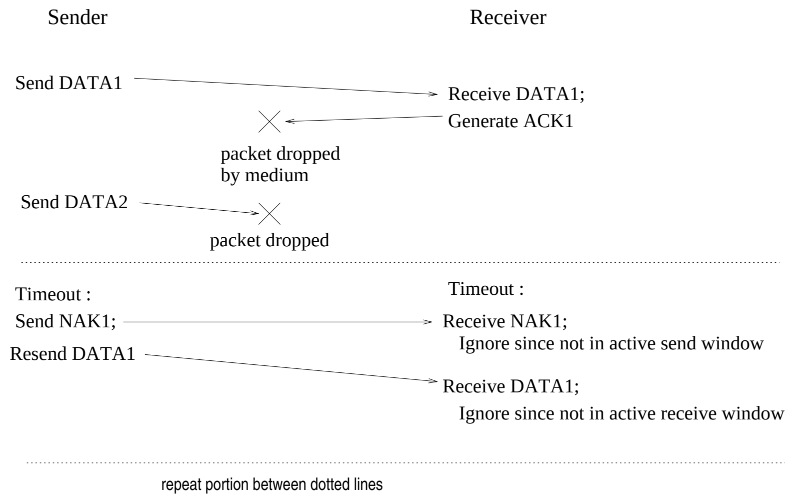

Fig. 1: An error scenario illustrating a livelock in the original version of the i-protocol.

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. (The fix appears as lines 107–114 in the VPL listing of the protocol.) With this fix the model checker was unable to find any livelocks in the protocol.

5 Model-Checking Results

As discussed in the Introduction, the i-protocol makes for a formidable case study for verification tools, and forms the basis for an interesting comparative study. Table 1 contains the performance data obtained by applying COSPAN (version 8.15), Murφ (version 3.0), SMV (version 2.4), SPIN (version 2.9.7), and XMC to the i-protocol. Results are given for $W=1$ and $W=2$, with the livelock error present (~fixed) and not present (fixed), and with a medium that can only drop messages (mini) versus one that can also corrupt messages (full). All results were obtained on an SGI IP25 Challenge machine with 16 MIPS R10000 processors and 3GB of main memory. Each individual execution of a verification tool, however, was carried out on a single processor with 1.9GB of available main memory.

A few comments about Table 1 are in order. On some runs, memory was exhausted before the verification effort could complete. This is indicated in the "Completed?" column. The timing figures given in the table are "wall-clock" time rather than cpu time. This makes a difference in exactly one instance, $W=2$/full/fixed for XMC, where 4.7GBytes of virtual memory are used. In this case, the wall-clock time is perceptively higher than the cpu time. Some table entries are left blank. This is because the corresponding data was unavailable because the tool does not provide it (e.g., the number of transitions, in the case of SMV) or because the tool failed to terminate on the run in question. The number of states reported by SMV is the total number of *reachable* states. The other tools give the number of explored states. Finally, the results for the Concurrency Factory are not included in the table. As mentioned in Section 1, this will change with the new release of the Factory.

As can be gleaned from the results of Table 1, the special-purpose cycle-detection algorithms of SPIN and COSPAN served them well. In particular, these tools were able to complete analysis of several complex versions of the i-protocol, including $W = 2$/mini/˜fixed, $W = 2$/mini/fixed, and $W = 2$/full/˜fixed. The ability to specify atomically executed code sections in SPIN also proved effective, enabling SPIN to complete analysis of the $W=1$/full/fixed version. SPIN, however, ran out of memry for $W=2$/full/fixed, despite the use of partial-order reduction and bitstate hashing (with 98% state-space coverage).

SMV exhibited excellent memory-usage performance on all runs of window size 1, but failed to complete in a reasonable amount of time on any run of window size 2. This is most likely due to an exponential blowup in the BDD representation for window sizes larger than 1. The dynamic variable reordering option of SMV was used on all runs reported in Table 1. Several static variable orderings were also tried, including a "sequential" ordering in which the variables of the sender precede the variables of the sender-to-receiver medium, which precede the variables of the receiver, etc. An "interleaved" ordering, in which the components' variables were strictly interleaved, was also attempted. In all cases, the dynamic reordering significantly outperformed the static ones.

Murφ and XMC performed the best on the i-protocol, completing on all cases of interest. Murφ uniformly exhibited superior memory-usage behavior (over all the other tools), due in part to the low-level nature of its specification language (guarded commands) and the succinct manner it encodes system states. Murφ was also fast. XMC, however, was faster than Murφ for all cases in which the livelock error was present. This is because of the local, top-down nature of XMC's model-checking algorithm (Murφ is a global model checker). Prior experience [RRR$^+$97] indicates that the space requirements of XMC can be reduced through source-level transformations aimed at optimizing the representation of

Version	Tool	Completed?	States	Transitions	Memory(MB)	Time(min:sec)
W=1 mini ~fixed	COSPAN	Yes	63K	204K	4.9	0:41
	Murφ	Yes	3K	8K	0.1	0:01
	SMV	Yes	24.5M		4.0	41:52
	Spin	Yes	425	768	749	0:10
	XMC	Yes	341	571	5	0:01
W=1 mini fixed	COSPAN	Yes	1.5M	5.9M	116	24:21
	Murφ	Yes	7K	19K	0.3	0:06
	SMV	Yes	27.7M		5.3	74:43
	Spin	Yes	322K	1M	774	0:31
	XMC	Yes	3K	12K	78	0:17
W=2 mini ~fixed	COSPAN	Yes	154K	486K	13	1:45
	Murφ	Yes	45K	122K	2	0:21
	SMV	No				
	Spin	Yes	35K	71K	751	0:12
	XMC	Yes	1034	1839	11	0:02
W=2 mini fixed	COSPAN	Yes	11.3M	42.7M	906	619:49
	Murφ	Yes	91K	240K	4	1:37
	SMV	No				
	Spin	Yes	1.9M	6M	905	2:28
	XMC	Yes	20K	74K	475	1:49
W=1 full ~fixed	COSPAN	Yes	116K	345K	9.1	17:03
	Murφ	Yes	54K	205K	2	0:25
	SMV	Yes	425.3M		6.0	201:04
	Spin	Yes	5.2K	10.1K	749	0:11
	XMC	Yes	961	1521	9	0:01
W=1 full fixed	COSPAN	No				
	Murφ	Yes	124K	458K	6	1:57
	SMV	Yes	583.3M		9.8	224:20
	Spin	Yes	12.6M	44.9M	1713	17:50
	XMC	Yes	36K	155K	1051	3:36
W=2 full ~fixed	COSPAN	Yes	194K	562K	15.9	29:40
	Murφ	Yes	1.1M	4M	20	9:43
	SMV	No				
	Spin	Yes	17K	22K	750	0:17
	XMC	Yes	4K	7K	35	0:05
W=2 full fixed	COSPAN	No				
	Murφ	Yes	2.1M	7.7M	89	41:55
	SMV	No				
	Spin	No				
	XMC	Yes	315K	1.33M	4708	47:15

Table 1: Model-checking results.

process terms. Finally, the number of states/transitions explored by XMC is appreciably lower in comparison with the other systems. This is primarily due to XMC's use of lazily evaluated logical variables to represent variables and data structures in the specification, and the fact that XMC treats sequences of pure computation steps as atomic.

6 Conclusions

We have shown how an actual bug in a real-life communications protocol can be detected and eliminated through the use of automatic verification tools supporting model checking. We have also tried to demonstrate the i-protocol's effectiveness as a verification-tool benchmark by conducting a comparative study of the performance of six widely used verification tools in analyzing the original and livelock-free versions of the protocol.

Pertinent future work includes recruiting the actual developers of the model checkers used in this study to encode and analyze the i-protocol. We expect that the performance of each tool will increase under these conditions and it would be interesting to learn what "tricks" the developers employ to attain this improvement.

For completeness, other properties of the i-protocol should be checked besides the absence of livelock, such as deadlock-freedom and eventual message delivery. It would be particularly interesting to apply a tool with deductive reasoning capabilities, such as PVS [ORR$^+$96], to the i-protocol, so that a parameterized version of the protocol (window size, buffer size, etc.) could be analyzed.

Finally, we invite developers of verification tools besides those considered in this case study to try their hand at the i-protocol and report the results to us for posting on the i-protocol web site. This will assist protocol developers and other software engineers interested in pursuing automated verification to make an educated decision about which tool is right for the task at hand.

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.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. To appear.

[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.

[CMCHG96] E. M. Clarke, K. McMillan, S. Campos, and V. Hartonas-GarmHausen. Symbolic model checking. In Alur and Henzinger [AH96], pages 419–422.

[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.

[Dil96] D. L. Dill. The Murφ verification system. In Alur and Henzinger [AH96], pages 390–393.

[HHK96] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Alur and Henzinger [AH96], pages 423–427.

[HP96] G. J. Holzmann and D. Peled. The state of SPIN. In Alur and Henzinger [AH96], pages 385–389.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[ORR$^+$96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [AH96], pages 411–414.

[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.

[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. Petersburgh, January 1986.

[XSB97] XSB. The XSB logic programming system v1.7, 1997. Available by anonymous ftp from `ftp.cs.sunysb.edu`.

A i-Protocol Pseudo-Code

```
{ all additions & subtractions are modulo SEQ }
process i-protocol-unit
  while true do
    select
      call sendmsg
      %
      call getpkt
      %
      call timeout
    endselect
  endwhile
endprocess { i-protocol-unit }

procedure sendmsg
  usr-send?*;
  while (sendseq - rack > W  | sendseq = rack) do
    select
      call getpkt
      %
      call timeout
    endselect
  endwhile
  pak.type := DATA;
  pak.seq := sendseq;
  pak.ack := recseq;
  sendseq := sendseq + 1;
  med-out!pak;
  lack := recseq
endprocedure { sendmsg }

procedure getpkt
  med-in?pak;
  case pak.hck
    false : skip ;
    true :
      call handle-ack;
      case pak.type
        DATA: call handle-data;
        NAK: call handle-nak;
        ACK: skip
      endcase
  endcase
endprocedure { getpkt }
```

```
procedure handle-data
  if  ~ (pak.seq - lack > W  | pak.seq = lack) then
    case pak.dck
      false :
        if ( (pak.seq ∉ (lack, recseq])  &  ~ recbuf[pak.seq]
              &  ~ nakd[pak.seq]) then
          pak.type := NAK;
          pak.ack := recseq;
          nakd[pak.seq] := true ;
          med-out!pak;
          lack := recseq
        endif
      true :
        nakd[pak.seq] := false ;
        if (pak.seq = recseq + 1) then
          recseq := recseq + 1;
          usr-recv!*; { replace by usr-recv!pak.seq for 'in-order delivery' }
          tmp := recseq + 1;
          while recbuf[tmp] do
            recseq := tmp;
            usr-recv!* ; { replace by usr-recv!tmp for 'in-order delivery' }
            recbuf[tmp] := false ;
            tmp := tmp+1
          endwhile
          if  ~ (recseq - lack < W/2) then
            pak.type := ACK;
            pak.ack := recseq;
            pak.seq := recseq;
            med-out!pak;
            lack := recseq
          endif
        else if ( (pak.seq ∉ (lack, recseq])  &  ~ recbuf[pak.seq]) then
          recbuf[pak.seq] := true ;
          tmp := recseq + 1;
          while  ~ (tmp = pak.seq) do
            if  ~ (nakd[tmp]  | recbuf[tmp]) then
              pak.type := NAK;
              pak.seq := tmp;
              pak.ack := recseq;
              nakd[pak.seq] := true ;
              med-out!pak;
              lack := recseq
            endif
            tmp := tmp + 1
          endwhile
        endif
    endcase
  endif
endprocedure { handle-data }
```

```
procedure handle-ack
  if  ~ ( pak.ack = sendseq  | pak.ack - rack > W
          | sendseq - pak.ack > W) then
    rack := pack.ack
  endif
endprocedure { handle-ack }

procedure handle-nak
  if  ~ ( pak.seq = sendseq  | pak.seq - rack > W
          | sendseq - pak.seq > W) then
    pak.type := DATA;
    pak.ack := recseq;
    med-out!pak;
    lack := recseq
  endif
  if FIXED then { bug fix }
    if (pak.seq = sendseq  & rack+1 = sendseq) then
    pak.type := ACK;
    pak.ack := pak.seq := recseq;
    med-out!pak;
    lack := recseq
    endif
  endif { bug fix }
endprocedure { handle-nak }

procedure timeout
  timeout!*;
  nakd := false ; { clear all entries of nakd array }
  select
    call getpkt
    %
    pak.type := NAK;
    pak.seq := recseq + 1;
    pak.ack := recseq;
    nakd[pak.seq] := true ;
    med-out!pak;
    lack := recseq;
    if  ~ (sendseq = rack + 1) then
      pak.type := DATA;
      pak.seq := rack + 1;
      pak.ack := recseq;
      med-out!pak
    endif
  endselect
endprocedure { timeout }
```