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

Y.S. Ramakrishna
Sun Microsystems Laboratories
Mountain View, CA 94043, USA

Oleg Sokolsky
Computer Command and Control Company
Philadelphia, PA 19103, USA

Xiaoqun Du, Scott A. Smolka, Eugene W. Stark
Department of Computer Science, SUNY at Stony Brook
Stony Brook, NY 11794–4400, USA

Abstract

We report on our efforts to detect, locate, and correct a non-trivial livelock in the i-protocol, a bidirectional sliding-window protocol employed in the publicly available GNU UUCP file transfer utility. Four widely used verification tools were chosen for this purpose, namely, the Concurrency Factory, SPIN, COSPAN, and SMV. As such, this paper serves both as a case study in protocol verification and as a comparative tool study.

1 Introduction

Model checking [CE81, QS82, VW86, Kur94] 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 four 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], SPIN [HP96], COSPAN [HHK96], and SMV [CMCHG96], 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, such as telephone 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

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

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, formalizing it in a Value Passing Language (VPL). VPL is the textual specification language of the Concurrency Factory and is derived from Milner's $\mathcal{M}_0$ [Mil89].

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 (or virtual circuit) 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 diagnostic facility that accompanies the Factory's model checker, 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.

The verification was particularly challenging because of the large theoretical state space of the protocol. Indeed, most exhaustive verifiers, or global model checkers, would be inadequate for this verification, since the state space of the protocol, even for a small window size of 2, is approximately 4×10^{17}. The Concurrency Factory's model checker was, however, able to detect and diagnose the livelock, since it is a *local* model checker. In effect, it exploits the structure of the formula expressing the property of interest to guide and limit its search to only the "relevant" portions of the global state space.

In addition, some judicious abstractions were used to reduce the size of the state space, retaining in our model only as much detail as appeared necessary for finding the bug. While many of these abstractions have been verified only in an informal sense, we describe how we verified formally the soundness of a certain data abstraction step that reduced the state space of the protocol by several orders of magnitude. This involved reducing the message sequence number space from 32 — the constant defined in the protocol's C code — to $2W$, where W is the window size. This abstraction is shown to preserve the truth of all modal mu-calculus formulæ, and, considering again the case of $W = 2$, reduces the size of the state space to approximately 1.5×10^{12}.

We repeated our model-checking-based verification of the i-protocol with the SPIN, COSPAN, and SMV 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 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. Although the Concurrency Factory was able to detect and diagnose livelock in the i-protocol for both window sizes 1 and 2, the livelock property must be explicitly encoded as a temporal logic formula; this adds a potential growth factor in the state space to be searched, namely, the size of the formula. 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. Our model-checking results are described more fully in Section 5.

The remainder of the paper is organized as follows. Section 2 provides an overview of the four tools that we used on the i-protocol. Section 3 gives a detailed account of the i-protocol, and discusses how we modeled it in each of the tools. Section 4 presents the abstraction technique we used on the protocol's message sequence number space and gives a sketch of its soundness proof. Section 5 summarizes the results of our model-checking experiments, and offers a comparison of the tools' performance on the protocol. This section also includes a scenario illustrating the livelock in the protocol that we discovered, and describes how a small change in the protocol effectively eliminates this form of livelock. Section 6 contains our concluding remarks.

The complete VPL, PROMELA, S/R, and SMV source listings of the i-protocol are available from `http://www.cs.sunysb.edu/~vicdu/ProtiSpec`. (PROMELA is SPIN's input language and S/R is COSPAN's input language.)

2 The Verification Tools

In this section, we briefly describe each of the tools we used in our analysis of the i-protocol.

2.1 The Concurrency Factory

The Concurrency Factory [CLSS96] is an integrated toolset for the specification, verification, and implementation of concurrent and distributed real-time systems such as communication protocols and process control systems. The main features of the Factory are: a *graphical user interface* that allows the non-expert to design and simulate concurrent systems using GCCS, a graphical process algebra; a *textual user interface* for VPL; a local model checker with partial-order reduction [RS97]

for the modal mu-calculus, a very expressive temporal logic; a local model checker for a real-time extension of the modal mu-calculus [SS95]; and a *graphical compiler* that transforms GCCS and VPL specifications into executable Facile [TLK96] code.

VPL is well suited for specifying concurrent systems with non-trivial control structure and data structures, and which involve value passing (as opposed to pure synchronization). 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.

VPL programs can be verified using LMC, the Concurrency Factory's local model checker for the propositional modal mu-calculus [RS97]. LMC computes 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. This "product graph" is constructed on-the-fly: each product graph node has an associated boolean value (representing the value of a subformula in an LTS state) and the value of a node is computed based on the values of its successors. Nodes are never constructed unless their values are needed to determine the value of a previously constructed node.

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 a diagnostic facility that allows the user to request that the contents of the depth-first search stack be displayed whenever a certain "significant event" occurs—for instance, when the search first encounters a state at which a logical variable is determined to be either true or false. This technique can be used to great effect, for instance, to discover execution paths leading to deadlocked states, or to livelock cycles.

2.2 SPIN

SPIN [HP96] is a model checker for asynchronous systems specified in the language PROMELA. Safety and liveness properties are formulated using linear-time temporal logic (LTL), and model checking is carried out using the automata-theoretic approach; i.e., by checking the non-emptiness of the intersection of Büchi automata (automata over infinite words), one for the system model and one for the formula.

Model checking in SPIN is performed on-the-fly—meaning that the state space of the model intersected with the formula automaton need only be built up to the point where the non-emptiness of the resulting automaton can be proven—and with partial-order reduction. The computation of the intersection can either 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. In the syntax of LTL, an acceptance property corresponds to a formula of the type $\Box\Diamond p$, where p is a user-defined accepting state. The violation of a progress property corresponds to a formula of the form $\Diamond\Box\neg p$, where p is a user-defined progress state. We will use 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, 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.

2.3 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, each 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 algorithms. Both algorithms are "on-the-fly." Like SPIN, COSPAN implements a kind of non-progress cycle-detection algorithm; instead of "progress state", COSPAN employs a notion of "recur edge." We use this facility of COSPAN 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.

2.4 SMV

SMV [CMCHG96] is an automatic tool for model checking CTL formulas with 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.

SMV's input language allows users to specify synchronous or asynchronous, detailed deterministic or abstract nondeterministic finite state machines. The language provides modular hierarchical

descriptions, reuse of components, and parameterization so that multiple instances of module can use different data values. Variables can be declared local, and the type of a variable may be boolean, enumerated, or integer subrange.

An SMV program can be viewed as a system of simultaneous equations whose solution determines the next state. Asynchronous systems can be modeled by defining a set of parallel processes whose actions are interleaved arbitrarily in the execution of the program.

Fairness constraints can be specified as arbitrary sets of states in the model described by a temporal logic formula. A path in the model is considered fair with respect to a set of fairness constraints if each constraint is true infinitely often along the path. Verification can be restricted to fair paths.

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. Our description is somewhat VPL-oriented, and we provide from time to time line numbers from the VPL source of the i-protocol (available on-line at `http://www.cs.sunysb.edu/~vicdu/ProtiSpec`), to allow the reader to follow the source code more closely.

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 we show in Section 4, 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 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 4–8) consists of busy waiting for one of the following events to occur, and taking appropriate actions as described:

(E1): a packet arrival (line 32) over the communication link (lower layer interface): the packet is first checked for header checksum errors (line 33), and silently discarded (line 34) 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 94–95) *rack*. This subsumes the handling of explicit ACK packets (line 40). If the received packet is a NAK for a sequence number in the sender's active window, the requested DATA packet is resent (lines 100–105). If the received packet is a DATA packet, its data checksum is first verified (line 47). 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 49–55). 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 60). 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 62–67). At each point, *recseq* is appropriately incremented, thus shifting up the active window (lines 59 and 63).

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

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

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 68–73). 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 75–76), and NAKs generated for all "earlier" missing packets, for which a NAK has not been sent since the last timeout (lines 77–87).

(E2): a user request (line 15) to send a new message (upper layer interface): The sender first checks (line 16) 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 23–28). 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 18) for the arrival of a new packet (see (E1) above), or (line 20) for the occurrence of a timeout (see (E3) below).

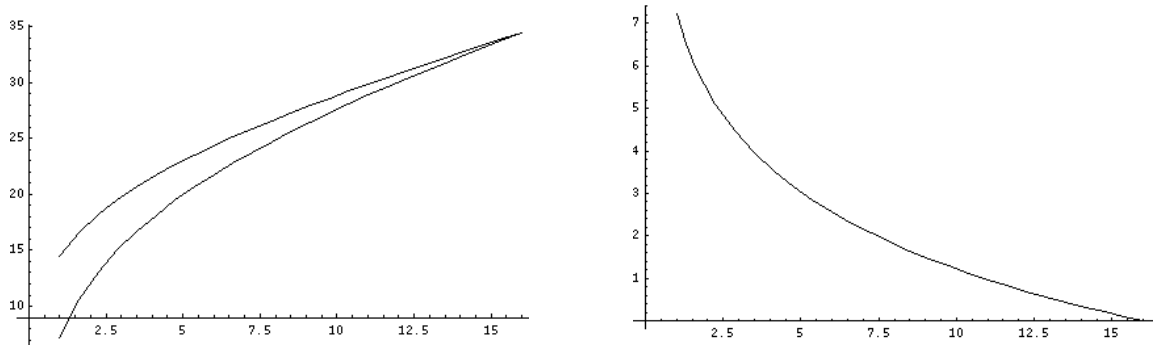
(E3): a timeout (line 114): The *nakd* buffer is first cleared (line 118), signaling that fresh NAKS may need to be sent out. If there is no packet in the receive buffer (from the lower interface), then the receiver sends a NAK for the "earliest" missing sequence number ($recseq + 1$) in its active window (lines 123–127). 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 128–133). 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.

4 State Space Reduction and Abstractions

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 "isomorphic" (in a sense made precise below) 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. A rough comparison of the estimated state spaces for the cases with and without this abstraction appears in Figure 1.

An important part of the verification is ensuring that this abstraction step is *sound*, in the sense



(a) (b)

W	$\lvert S\rvert$	$\lvert h(S)\rvert$	$O\left(\frac{\lvert S\rvert}{\lvert h(S)\rvert}\right)$
1	2.749×10^{14}	1.638×10^{7}	10^{7}
2	3.862×10^{17}	1.473×10^{12}	10^{5}
3	5.609×10^{19}	2.437×10^{15}	10^{4}
4	3.069×10^{21}	7.493×10^{17}	10^{4}

Figure 1: State reduction using the abstraction described in the text. Plot (a) above shows the growth in state space for the abstracted and unabstracted systems as window size is increased from 1 to 16. Plot (b) gives corresponding change in the state space reduction factor. Observe that, in both plots, the y-axes are on a logarithmic scale (base 10).

that, at least for the properties of interest, the abstract model satisfies a property if and only if the implementation does. We provide a proof sketch of this fact, for $W \leq \lfloor SEQ/2 \rfloor$, by exhibiting a bisimulation between the concrete model and the abstract model. Because two bisimilar processes satisfy the same set of modal mu-calculus formulæ [Sti87], we are ensured that any property proved of the abstraction holds also for the concrete model.

We first show how a *labeled transition system* (LTS) can be derived from a given VPL program. The *states* of the LTS are the reachable assignments of values to variables from their respective (finite) domains. For each VPL process, we assume that the set of variables also includes a program counter. The execution of program statements constitute *transitions* between these states. Transitions not involving communication are labeled by the internal action τ; for example, consider `x1 := x2 + x3`. The poststate of this transition is the assignment of the transition's prestate, with the value for `x1` updated by `x2 + x3`. Transitions corresponding to communication statements such as `chan!x` and `chan?x` are handled in a fashion similar to Milner's translation of value-passing CCS to basic CCS [Mil89].

LTSs are composed (in parallel) using an interleaving semantics with usual CCS synchronization over shared ports. The set of variables of the resulting LTS is the disjoint union of the variables of the component LTSs, and the "program counter" for the composite LTS is a vector of constituent LTS program counters.

Suppose $L = \langle S, s_0, T \rangle$ is an LTS, with S the set of states, s_0 an initial state, and $T \subseteq S \times Act \times S$

the set of action-labeled transitions. An *abstraction mapping* from L to an LTS $L' = \langle S', s'_0, T' \rangle$ is a function $h : S \to S'$ with the property that the binary relation $\{(s, h(s)) : s \in S\}$ is a (strong) bisimulation. We say that L' is an *abstraction* of L just in case there exists an abstraction mapping from L to L'.

Let $L_{\rm ip}$ denote the LTS associated with the i-protocol VPL code, and $L'_{\rm ip}$ the LTS associated the VPL code in the special case that $SEQ = 2W$. We show that, for $SEQ \geq 2W$, the LTS $L'_{\rm ip}$ is an abstraction of $L_{\rm ip}$, and hence the two LTSs satisfy the same set of modal mu-calculus formulæ. The abstraction mapping $\mathcal{H}$ we used to demonstrate that $L'_{\rm ip}$ is an abstraction of $L_{\rm ip}$ is defined as follows: for each state of $L_{\rm ip}$, consisting simply of an assignment a of values to variables, the state $\mathcal{H}(a)$ of $L'_{\rm ip}$ is characterized by the conditions:

$\mathcal{H}(a)(v) = a(v) \bmod 2W$ for $v \in \{sendseq, lack, recseq, lack, tmp, pak.seq, pak.ack\}$
$\mathcal{H}(a)(v[i + a(lack) \bmod 2W]) = a(v[i + a(lack) \bmod SEQ])$ for $i \in 2W, v \in \{recbuf, nakd\}$
$\mathcal{H}(a)(v) = a(v)$ for all other v

In particular, the program counter is left unchanged by $\mathcal{H}$. The second clause above describes how the values of the array variables $recbuf, nakd$, which are vectors of dimension SEQ in $L_{\rm ip}$, relate to the $2W$-dimensional values of these same array variables in $L'_{\rm ip}$.

The proof that $\mathcal{H}$ is an abstraction mapping from $L_{\rm ip}$ to $L'_{\rm ip}$ consists of showing that the relation $\mathcal{B} = \{ (s, \mathcal{H}(s)) \mid s \in S \}$ is a bisimulation. Specifically, given a state s and the corresponding abstract state $\mathcal{H}(s)$, for any transition from s to r in $L_{\rm ip}$ there is an identically labeled transition from $\mathcal{H}(s)$ to $\mathcal{H}(r)$ in $L'_{\rm ip}$, and conversely, for any transition from $\mathcal{H}(s)$ to r' in $L'_{\rm ip}$ there is an identically labeled transition from s to r in $L_{\rm ip}$, such that $r' = \mathcal{H}(r)$. The proof is by exhaustive analysis of the VPL code. Since the program counter is left unchanged by $\mathcal{H}$, the work may be organized as a collection of independent cases, each of which concerns the execution of the same program step by both $L_{\rm ip}$ and $L'_{\rm ip}$.

5 Model Checking Results

The livelock error detected by the Concurrency Factory's local model checker, and subsequently by SPIN, COSPAN and SMV, is illustrated in Figure 2 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.

The error was detected by checking explicitly for the presence of livelock cycles using the

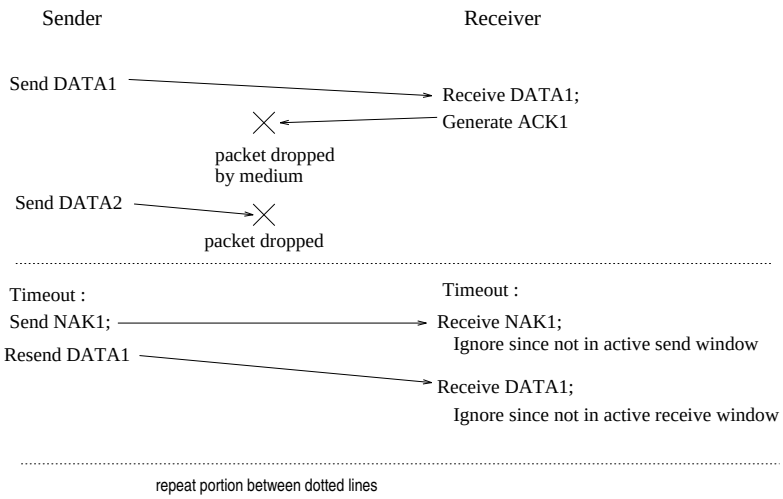


Figure 2: An error scenario illustrating a livelock in the original version of the i-protocol. The error trace was obtained using the Factory's local model checker.

following modal mu-calculus formula:

$$\mu X \cdot \langle - \rangle X \vee \langle \text{Lsend} \rangle \, \mu Y \cdot [\text{Rrecv}]\mathsf{false} \wedge (\langle - \rangle Y \vee \nu Z \cdot [\text{Rrecv}]\mathsf{false} \wedge (\langle \tau \rangle Z \vee \langle \text{Ltimeout} \rangle Z))$$

Lsend corresponds to a user-level send action by the sender side, Ltimeout to a timeout action on the sender side, Rrecv to a user-level receive action by the receiver side, τ to any internal action of the composite system, and $-$ to any action (either internal or visible) of the composite system. Informally, the formula is satisfied by any (looping) execution in which there is an Lsend action that is never followed by an Rrecv action, and after a certain point (inside the loop) the only visible action is Ltimeout.

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 106–113 in the VPL listing of the protocol.) With this fix the model checker was unable to find any livelocks in the protocol.

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 the Concurrency Factory (LMC), SPIN (version 2.9.7), COSPAN (version 8.15), and SMV (version 2.4) 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 1.8GB of available main memory.

As can be gleamed from the results of Table 1, the special-purpose cycle-detection algorithms

Version	Tool	Completed?	States	Transitions	Memory(MB)	Time(min:sec)
W=1 mini ~fixed	COSPAN	Yes	63K	204K	4.9	0:41
	LMC	Yes	171K	276K	41	70:30
	SPIN	Yes	425	768	749	0:10
	SMV	Yes	24.5M	-[a]	4.0	41:52
W=1 mini fixed	COSPAN	Yes	1.5M	5.9M	116	24:21
	LMC	Yes	4.0M	6.8M	958	29.5 hrs
	SPIN	Yes	962K	2.6M	820	1:02
	SMV	Yes	27.7M	-	5.3	74:43
W=2 mini ~fixed	COSPAN	Yes	154K	486K	13	1:45
	LMC	Yes	534K	880K	118	349:31
	SPIN	Yes	35K	71K	751	0:12
	SMV	No			>28	>35 hrs
W=2 mini fixed	COSPAN	Yes	11.3M	42.7M	906	619:49
	LMC	No				
	SPIN	Yes	7.2M	17.4M	1789	6:23
	SMV	No				
W=1 full ~fixed	COSPAN	Yes	116K	345K	9.1	17:03
	LMC	No[b]			1846	$\simeq$30 hrs
	SPIN	Yes	6.4K	8.1K	749	0:10
	SMV	Yes	425.3M	-	6.0	201:04
W=1 full fixed	COSPAN	No[c]			>520	>48hrs
	LMC	No				
	SPIN	Yes[d]	27.3M	63M	1650	26:20
	SMV	Yes	583.3M	-	9.8	224:20
W=2 full ~fixed	COSPAN	Yes	194K	562K	15.9	29:40
	LMC	No				
	SPIN	Yes	17K	22K	750	0:17
	SMV	No				
W=2 full fixed	COSPAN	No				
	LMC	No				
	SPIN	No[e]	29.7M	48.3M	1878	20:03
	SMV	No				

Table 1: Model checking results.

[a] No information about the number of transitions is made available by SMV. Also, the number of states reported by SMV is the number of *reachable* states.

[b] The process was terminated because it ran out of memory.

[c] The process was killed after 2 days.

[d] Bit-state analysis used. Coverage of state space better than 98%.

[e] The process ran out of memory. It used bit-state analysis with coverage better than 98%.

of SPIN and COSPAN served them well. In particular, these tools were able to complete analysis of more complex versions of the i-protocol, including $W = 2$/mini/~fixed, $W = 2$/full/fixed, and $W = 1$/full/~fixed. SPIN's bitstate hashing technique also proved effective, enabling SPIN to be the only tool to complete analysis of the $W = 1$/full/fixed version (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.

For the versions of the protocol for which the Concurrency Factory completed, it exhibited very competitive memory usage, but comparatively poor time usage. We attribute this mainly to two factors. First, the use of an explicit modal mu-calculus formula to encode livelock introduces approximately a factor of 10 into the run times, as nodes in the state space explored by the Factory are global-state/logical-variable pairs. Also, the current version of the Factory does not provide direct support for value passing but instead employs a standard but rather inefficient encoding of value passing in a basic process calculus. This latter situation is expected to be remedied in the next release of the Concurrency Factory, in which an explicit environment structure is used to keep track of variable bindings.

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 performing a comparative study of the performance of four widely used verification tools in analyzing the original and livelock-free versions of the protocol.

Pertinent future work includes the application of other verification tools to the i-protocol, and the creation of a website that contains the tool input descriptions of the i-protocol along with all relevant performance data. 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.

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.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[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.

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

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[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.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[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.

[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.

[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.

[SS95] O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification.* American Mathematical Society, 1995.

[Sti87] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–347, 1987.

[TLK96] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *Proceedings of the Seventh International Conference on Concurrency Theory (CONCUR '96)*, Vol. 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer-Verlag, 1996.

[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*, pages 184–192, St. Petersburgh, January 1986.