Experiments in Theorem Proving and Model Checking for Protocol Verification*

To appear in the Proceedings of FME, March 1996, Oxford

Klaus Havelund[1**] and Natarajan Shankar[2***]

[1] LITP, Institut Blaise Pascal, 4 place Jussieu, 75252 Paris, France
Email: havelund@litp.ibp.fr URL: http://cadillac.ibp.fr:8000/~havelund
[2] Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA
Email: shankar@csl.sri.com URL: http://www.csl.sri.com/~shankar/shankar.html

Abstract. Communication protocols pose interesting and difficult challenges for verification technologies. The state spaces of interesting protocols are either infinite or too large for finite-state verification techniques like model checking and state exploration. Theorem proving is also not effective since the formal correctness proofs of these protocols can be long and complicated. We describe a series of protocol verification experiments culminating in a methodology where theorem proving is used to abstract out the sources of unboundedness in the protocol to yield a skeletal protocol that can be verified using model checking.
Our experiments focus on the Philips bounded retransmission protocol originally studied by Groote and van de Pol and by Helmink, Sellink, and Vaandrager. First, a scaled-down version of the protocol is analyzed using the Murϕ state exploration tool as a debugging aid and then translated into the PVS specification language. The PVS verification of the generalized protocol illustrates the difficulty of using theorem proving to verify infinite-state protocols. Some of this difficulty can be overcome by extracting a finite-state abstraction of the protocol that preserves the property of interest while being amenable to model checking. We compare the performance of Murϕ, SMV, and the PVS model checkers on this reduced protocol.

* Sam Owre (SRI) has assisted with the use of PVS and suggested several improvements to the paper. Sreeranga Rajan (SRI) was instrumental in integrating the mu-calculus model checker (built by Geert Janssen of Eindhoven University of Technology) into PVS. SeungJoon Park of Stanford University implemented the Murϕ-to-PVS translator. David Cyrluk (SRI and Stanford University) sped up parts of the PVS equality decision procedure. Ken McMillan (Cadence Labs) suggested that we examine forward reachability as a way of obtaining efficiency from the PVS model checker. We are also grateful to John Rushby (SRI) for facilitating Klaus Havelund's visit to SRI, and to Therese Hardin (LITP) for providing a stimulating environment at LITP in Paris.

** Supported by a European Community HCM grant, with origin institution being DIKU, Institute of Computer Science, University of Copenhagen, Denmark.

*** Supported by NSF Grant CCR-930044 and by ARPA through NASA Ames Research Center under Contract NASA-NAG-2-891 (ARPA Order A721).

1 Introduction

Communication protocols are an important class of concurrent algorithms that pose a difficult challenge for existing verification technologies [11]. Tools based on model checking and state exploration are effective and widely used for protocol verification, but many real-life protocols are not finite state and cannot be fully analyzed by these methods. In these instances, verification techniques based on theorem proving can be applied, but these have the disadvantage that they are not automatic and the verification effort involved can be substantial. In this paper, we examine the relative efficacy of finite-state and theorem proving approaches to verification when applied to a non-academic example of a communication protocol. We show how it is possible to combine the two techniques to create a useful methodology for protocol verification.

The specific protocol we examine is the socalled bounded retransmission protocol (BRP) from Philips Electronics. This variant of the alternating bit protocol transmits files each consisting of a sequence of individual messages. File transmission is aborted if any message in the file remains unacknowledged after a fixed number of retransmissions. This protocol has already been verified by researchers at Philips and CWI [10] using the Coq proof checker [5] using the framework of Lynch and Tuttle's I/O automata [16]. Their hand proof effort occupied two man-months, and it took them three man-months to mechanize this proof using Coq. The protocol has also been formalized in the process algebra μCRL and similarly checked using Coq [9]. This proof also required a serious amount of effort.

The interesting question therefore is whether this verification effort can be dramatically reduced, perhaps by using a combination of finite state and theorem proving techniques. To explore this question, we first consider a scaled-down version of the protocol BRP-M and show that it can be quickly analyzed and debugged using the Murϕ state exploration tool from Stanford University. This Murϕ specification can be converted into PVS [20] using a mechanical translator. The PVS description of the protocol is then generalized to the full protocol BRP-PVS, and the main safety property of the protocol is proved in a conventional manner as an invariant. Our initial PVS proof attempt of BRP-PVS took about three months to develop and verify, and this is roughly similar to the Philips/CWI effort. By employing a more finely tuned proof methodology and by taking further advantage of the automation provided by PVS, the verification has been redone in about one man-month.

Since this level of effort is still very large, we investigate techniques for reducing the verification effort without compromising the generality of the protocol. The PVS theorem prover has recently been extended with mu-calculus based model checking [21] so it is natural to ask whether model checking can somehow be applied to BRP-PVS. We answer this question in the affirmative by using theorem proving to construct a property-preserving finite-state abstraction BRP-mu that can be verified using model checking. There are three sources of unboundedness in the state space of the protocol: the message data, the retransmission bound, and the file length. Each of these sources of unboundedness can be eliminated by means of abstraction. The correspondence between BRP-PVS and BRP-mu is verified using PVS. The resulting finite-state protocol BRP-mu can be verified using a model checking or state exploration tool. We have successfully applied and can compare the PVS model checker [12], Murϕ [18], and SMV [17] on this example. The model checking part is automatic, but our initial attempts with the PVS model checker were unsuccessful until the mu-calculus definition of invariance was revised to compute fixpoints differently.

The general lesson from this is that the correctness of communication protocols is primarily control-sensitive, and the most effective verification approach is to use theorem proving to abstract out the control skeleton which can then be verified by finite-state techniques. We believe that the above verification paradigm can be generalized to apply to other protocols of industrial relevance. The main contribution of the paper is a mechanized methodology for industrial-strength protocol verification where:

1. A scaled-down version of the protocol is debugged using state exploration
2. This scaled-down version is generalized to recover the full version of the protocol for verification using theorem proving
3. Theorem proving is used to abstract out a finite-state protocol whose correctness (when established by model checking) implies the correctness of original protocol.

The work reported here is very much in progress. The effort saved by combining theorem proving and model checking is still quite modest at this point, but we believe that such savings can indeed be achieved through a more aggressive application of our proposed methodology.

The main difference between our work and previous work is that we develop a mechanized verification methodology for communication protocols where theorem proving is used to compute finite abstractions that can be verified by model checking. The closest related work is obviously the earlier verification of Helmink, Sellink, and Vaandrager [10]. The bulk of their verification is in proving various invariance properties but their main result is a refinement argument showing that one I/O automaton specification implements another more abstract one. We have employed a formalization that is closer to the state-transition model of Unity [3] and TLA [14]. By superposing the abstract and concrete state machines, we reduce the refinement demonstration to that of invariance. While the manual effort required by both their proof and by our initial proof attempt with BRP-PVS is comparable, PVS seems to provide greater and more efficient automation in the verification process particularly through the use of highly optimized rewriting and BDDs [6]. Our use of the abstracted protocol BRP-mu yields a simplification in the proof and a valuable technique for other protocol correctness proofs.

Groote's and van de Pol's specification [9] in μCRL is notationally compact, but their computer aided verification in Coq requires detailed encoding, and the resulting Coq description is fairly large and their verification is not mechanized to the degree achieved in PVS.

Müller and Nipkow [19] use a clever abstraction for reducing the alternating bit protocol to an infinite-state system with only a finite number of reachable states. Most finite-state model checking tools cannot cope with potentially infinite but reachably finite state spaces and therefore cannot exploit such an abstraction. Cardell-Oliver [2] has used the HOL proof checking system [8] to verify the sliding window protocol. It would be an interesting challenge to obtain a finite-state abstraction of the sliding window protocol.

Lam and Udaya Shankar [13] present a systematic method of projecting images of protocols by applying stepwise refinement to the protocol with respect to the property being verified. Their abstractions preserve the property so that protocol M has property P if and only if the abstract protocol M' has the property P. Our abstractions only preserve the property in one direction, i.e., if the abstract protocol M' has property P' (which in our case need not be P), then the concrete protocol M has property P. This means that we have much more freedom in our choice of abstractions, and in particular, we can introduce more nondeterminism into the abstract protocol. Some of the specific abstractions proposed here cannot be obtained by Lam and Shankar's technique. We do not provide a systematic method for obtaining abstractions; this is a topic for future research. The theoretical ideas underlying our use of abstraction have been previously studied [4, 7, 15].

2 The Bounded Retransmission Protocol

The bounded retransmission protocol developed at Philips Research Laboratory communicates messages from a *producer* to a *consumer* over an unreliable physical medium that can lose messages. The protocol is a nontrivial extension of the alternating bit protocol [1] that uses timeouts and aborts transmission following a bounded number of retransmission attempts. The *environment* to the protocol consists of the producer and the consumer. The black box view of the system is that it accepts requests REQ(f) from the producer to transmit the file f. When transmission of a file has been either completed or aborted, the producer receives a confirmation CONF(c), where c is either OK, NOT_OK, or DONT_KNOW, respectively indicating that the file was successfully transmitted, aborted, or that the last message in the file was not acknowledged but might have been received by the consumer. The consumer either receives an IND_ERR signal indicating that the file transmission was aborted, or an IND(m, i) signal where m is the message and i is either FIRST, LAST, or INCOMPLETE corresponding to the first, last, or an intermediate message in the file.

The protocol consists of a *sender* program at the producer side; a *receiver* program at the consumer side, and two channels (one-place buffers): a message channel K, and an acknowledgment channel L. Both channels are unreliable in that they can lose messages or acknowledgments. The protocol is pictured in figure 1. The sender sends each message over the channel K and then waits for an acknowledgment on channel L. If there is no acknowledgment, the sender times out and retransmits the message. There is a fixed upper bound on the number of such retransmissions.

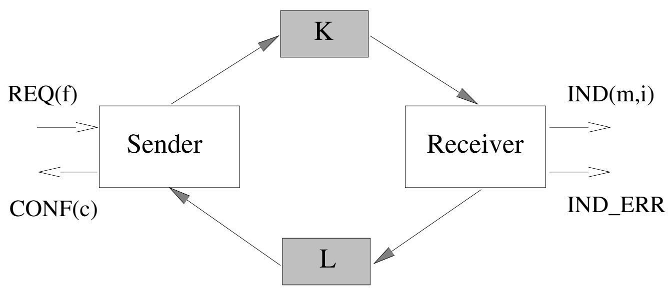

Fig. 1. The BRP Protocol

The protocol uses three timers to deal with loss of messages and acknowledgments. A timer has a fixed period T of time associated. When it is *set*, a *timeout* occurs T time units or more later. The first timer in the sender is used to detect the loss of a message or an acknowledgement. It is used as follows: when the sender sends a new message over K, timer 1 is set. The time associated with this timer exceeds the time it takes from when a message has been sent over K until the corresponding acknowledgment is received over L. If an acknowledgment comes back within this time, the timer is *cleared* (and the next message is sent). If not, a timeout occurs whereupon the message is retransmitted, and the timer is set again. When the retransmission bound has been reached, the sender aborts transmission and confirms that the transmission failed. Either it confirms CONF(NOT_OK) or it confirms CONF(DONT_KNOW). Two other timers are used to bring the sender and the receiver back in synchrony after a file transmission has been aborted by the sender. We do not model the real-time aspects of the protocol but instead represent the timers by timer events. For example, the timeout event which is supposed to detect message loss is instead defined to occur when a message is lost. This simplification is also present in Helmink, *et al* [10].

The receiver may also retransmit acknowledgments. This happens when the receiver gets a message that it has already received once. The receiver distinguishes between an old message and a new message via the alternating bit (the toggle) which is part of the message.

We can now examine the behavior of the protocol for the case when no messages or acknowedgments are lost. The sender sends each individual message in the file to the receiver over the channel K in the form (first,last,toggle,m). If the producer signals REQ(f) where f is $[m_1, m_2, m_3]$, then the sender first sends the tuple (true,false,toggle,m_1) on channel K. Upon receipt of this message, the receiver signals IND(m_1,FIRST) to the consumer and sends an acknowledgement on channel L. The sender then sends the second message as (false,false,$\neg$ toggle,m_2). The receiver correspondingly signals IND(m_2,INCOMPLETE) and acknowledges receipt. The sender then sends the last message as (false,true,toggle,m_3). The receiver now signals IND(m_3,LAST) and acknowledges receipt. The sender on receiving this acknowledgment signals CONF(OK) to confirm successful transmission to the producer.

3 State Exploration Using Murϕ

In this section we present the formulation of the protocol and its correctness criteria in Murϕ [18], a state exploration tool for finite state transition systems. Murϕ uses a program model that is similar to Unity [3]. A Murϕ program has three components: a declaration of the global variables, a description of the initial state, and a collection of transition rules. Each transition rule is a guarded command that consists of a boolean guard expression over the global variables, and a deterministic statement that changes the global variables. Transition rules can include `Assert` statements that terminate execution when the asserted condition is falsified. We use Murϕ as an effective debugging tool for testing invariance assertions.

An execution of a Murϕ program is obtained by repeatedly *(1) arbitrarily selecting one of the transition rules where the boolean guard is true in the current state; (2) executing the statement of the chosen transition rule.* The statement is executed atomically: no other transition rules are executed in parallel. Thus state transitions are interleaving and processes communicate via shared variables. The notion of process is not formally supported, but may be thought of as a subset of the transition rules. The Murϕ verifier tries to explore all reachable states in order to ensure that all `Assert` statements hold. If a violation is detected, Murϕ generates a violating trace.

Programming the Protocol. The protocol we have described in the previous section takes three parameters: the kind of data transmitted, the size of files (how many messages in each file), and finally the number of retransmissions (max) performed by the sender before it aborts file transmission. It turns out that when we choose the data domain to be finite, bound the size of files, and choose some fixed value for max, then the protocol state space is bounded and it can be formulated and verified in Murϕ. We therefore choose the data domain to be boolean, fix the size of files at 3, and set the retransmission limit to 2:

Const	Type	Var
`last : 3;`	`Data : boolean;`	`file : File;`
`max  : 2;`	`File : Array[1..last] Of Data;`	`head : 1..last+1;`
		`rn   : 0..max;`

A file is modeled as an array of data, of length 3, where the `head` variable points to the current message being sent, exceeding the domain of the array if the file is empty. The variable `rn` contains the current number of retransmissions tried. Recall that channel `K` carries tuples containing a `first`, `last`, `toggle` and `data` field:

```
Type
  Msg : Record
          first,last,toggle : boolean;
          data : Data
        End;
```

```
Var
  K : Msg;
  K_full : boolean;
```

When the sender sends a message, it just writes to `K`, and when the receiver reads this message it just reads `K`. The flag `K_full` is raised when some message has been written to `K`, i.e., the sender sets it to true when writing to `K`, and the receiver sets it to false when reading from `K`. Similar flags are used for other variables that play the role of "channels." The sender control is managed via a sender program counter `spc` of type `Spc`:

```
Type
  Spc : Enum{WR, SF, WA, SC, WT2};
```

```
Var
  spc : Spc;
```

When, for example, the sender is waiting for a new file, `spc` is `WF`. We only explain a few of the rules in the Murϕ specification. Assume that the environment writes new files to a `req` channel. The following sender rule reads a new file to be transmitted from the `req` channel into the `file` variable.

```
Rule "read_req"
  spc = WR & req_full ==>
    req_full := false; file := req; head := 1; spc := SF;
    verify_REQ(req)
End;
```

The rule is named `read_req`. The precondition requires that the sender's program counter is `WR`: "wait for a request". Also, the `req_full` flag must be true – the environment must have written some file into `req`. The `head` is set to point to the first message in the new file, and the program counter is set to `SF`: "send the file." The `verify_REQ` procedure contains part of the abstract specification of the protocol. The call will verify certain assertions in the abstract protocol, and if they fail to hold, program execution will terminate. The abstract protocol thus "polices" the behavior of the protocol; this will be explained in the next section. Note that the "policing" only takes place for so called *external* transition rules: the ones that model externally visible events according to a black box view. Communications on channels `K` and `L` are, for example, not external.

The next rule models how the sender sends a message to the receiver on `K`:

```
Rule "write_K"
  spc = SF & !K_full ==>
    K_full := true; K.first := sfirst; K.last := (head = last);
    K.toggle:= stoggle; K.data := file[head];
    spc:= WA; rn := rn+1; stimer1_on := true;
End;
```

The program counter is updated so that the sender now waits for an acknowledgment to arrive. The number `rn` of retransmissions is incremented, and finally, a timer `stimer1` is set: if an acknowledgment does not arrive within a certain time period, a timeout will occur.

There are other sender rules, and there are similar rules for the receiver, seventeen in total. The complete program is included in appendix B.

The Correctness Criteria. The abstract protocol specification is written as part of the Murϕ program and uses its own local variables. For example, the following three variables are declared:

```
Var
  abusy : boolean; afile : File; ahead : 1..last+1;
```

The `abusy` variable is true whenever a file is being transmitted. The variables `afile` and `ahead` will together at any time model which message the abstract protocol is prepared to transmit (by an `IND` action).

Now we are ready to define the abstract protocol. We do this in terms of a collection of procedures, one for each of the four external activities of the protocol: `REQ`, `IND`, `IND_ERR`, and `CONF`. For example, we saw previously the call `verify_REQ(req)`. This procedure is defined as follows:

```
Procedure verify_REQ(f:File);
Begin
  Assert !abusy;
  abusy := true; afile := f; ahead := 1;
End;
```

So these procedures are supposed to model the same external behavior as the protocol, ignoring channels `K` and `L`. Note how the `Assert` statements function as "pre-conditions": when the protocol calls the procedure, the `Assert` condition is evaluated, and if it evaluates to false, the Murϕ verification terminates, printing the trace of states that leads to the falsified assertion. In the above case, `abusy` must be false. The complete specification is included in appendix A. When we verified the protocol, no such error states occurred: the downscaled protocol was verified "correct" in 784 seconds on a Sun Sparc station.

4 Theorem Proving: Proving Safety with PVS

In the previous section, we verified a scaled down finite-state version of the protocol using Murϕ. The advantage of the verification was that it was automatic. In this section, we shall describe the use of PVS to construct a full-scale proof for the complete infinite state protocol. In order to obtain an infinite state protocol in PVS we apply a Murϕ-to-PVS translator. That is, we apply this translator to the Murϕ program in the previous section, and get a corresponding PVS specification. This is still a finite-state specification and to obtain the infinite state specification, we manually modify a few of the PVS declarations.

Applying the translator to the Murϕ program yields two PVS theories. The first one (`protocol`) contains the protocol itself. The second theory (`protocol_safe`) contains the statement of the correctness criteria. This correctness criteria was implicit in the Murϕ program: whenever an `Assert` evaluated to false, the verifier would terminate. We have to find a way of modeling this in the PVS-specification.

The Protocol. The Protocol (obtained by the translation, and some modification) is represented as a theory in PVS, a portion of which is shown below.

```
protocol : THEORY
BEGIN
  last     : posnat
  max      : posnat
  Data     : NONEMPTY_TYPE
  Position : TYPE = {i: int | 1 <= i AND i <= last+1}
  File     : TYPE = ARRAY[Position -> Data]
  Msg      : TYPE = [# first,last,toggle : bool, data: Data #]
  Spc      : TYPE = {WR, SF, WA, SC, WT2}
  State    : TYPE = [# file  : File, head  : Position, spc : Spc ...
                       afile : File, ahead : Position, SAFE : bool #]
  ...
END protocol
```

The theory contains definitions of constants, types, functions, relations, etc. We have modified the first three declarations (i.e., `last`, `max`, and `Data`) in order to get an inifinite state protocol. The Murϕ state is modeled as a record. The additional state component `SAFE` captures the effect of an `Assert` command. It is initially `TRUE`, and can only be affected by the `Assert` commands in the abstract specification which checks the externally observable behaviour of the protocol.

We first look at the PVS definition of the abstract specification. Murϕ transitions are translated into PVS as functions over state. The abstract specification is defined as a function which takes an input state and the external action and returns an output state which is the same as the input state except that the abstract state variables may have changed and the `SAFE` variable may have been falsified as a result of a failed `Assert`. The action argument is constructed as an element of the abstract datatype `Action` shown below. This datatype definition uses two enumerated types `Conf` and `IndT` and has four constructors: `REQ`, `IND`, `IND_ERR`, and `CONF`. There is a recognizer corresponding to each constructor, and some of the constructors have associated accessors, e.g., the accessor `req` corresponds to the constructor `REQ`.

```
Conf : TYPE = {OK,NOT_OK,DONT_KNOW}
IndT : TYPE = {FIRST,INCOMPLETE,LAST}

Action : DATATYPE
  BEGIN
    REQ(req:File):REQ?
    IND(data:Data,ind:IndT):IND?
    IND_ERR:IND_ERR?
    CONF(conf:Conf):CONF?
  END Action
```

With this type, the abstract protocol can be defined as the function `automaton` shown below. The body is a case-expression over the constructors of the action data type. In the `REQ(f)` case, `f` is bound, the `SAFE` variable is set to denote the value of the condition `NOT abusy(st)` — exactly the `Assert` condition in the Murϕ program. The state is then "updated" corresponding to the Murϕ assignment statements where sequencing is enforced by the `LET` construct.

```
automaton(st:State,action:Action):State =
  CASES action OF
    REQ(f) :
      LET
        st = st WITH [SAFE := NOT abusy(st)],
        st = st WITH [abusy := TRUE],
        st = st WITH [afile := f],
        st = st WITH [ahead := 1]
      IN st,
    IND(d,i) : ...
    IND_ERR  : ...
    CONF(c)  : ...
  ENDCASES
```

The Murϕ transition rule `read_req` is translated as the function `Rule_read_req` shown below.

```
Rule_read_req(st:State):State =
  IF spc(st) = WR AND req_full(st) THEN
    LET
      st = st WITH [req_full := FALSE],
      st = st WITH [file := req(st)],
      st = st WITH [head := 1],
      st = st WITH [spc  := SF],
      st = automaton(st,REQ(req(st)))
    IN st
  ELSE
    st
  ENDIF
```

Another perhaps more interesting rule is:

```
Rule_write_K(st:State):State =
  IF spc(st) = SF AND NOT K_full(st) THEN
    LET
      st = st WITH [K_full := TRUE],
      st = st WITH [(K)(first) := sfirst(st)],
      st = st WITH [(K)(last) := head(st) = last],
      st = st WITH [(K)(toggle) := stoggle(st)],
      st = st WITH [(K)(data) := file(st)(head(st))],
      st = st WITH [spc := WA],
      st = st WITH [rn := incr_rn(rn(st))],
      st = st WITH [stimer1_on := TRUE]
    IN st
  ELSE
    st
  ENDIF
```

We have seen that rules are modeled as functions of type `[State -> State]` since they are deterministic. The program which features nondeterminism in the selection of the transition rule is modeled with the help of a transition relation `transition : [[State,State] -> bool]` between states. That is, `transition(s1,s2)` holds between two states `s1` and `s2`, if one of the rules can bring us from state `s1` to `s2`. The relation is defined as follows:

```
transition(st1,st2:State):bool =
  ...
  OR st2 = Rule_read_req(st1)
  OR st2 = Rule_write_K(st1)
  ...
```

A program is a predicate that holds of a sequence of states `aa` if and only if the initial state of `aa` is `startstate` and each pair of successive states is related by the `transition` relation.

```
Behavior : TYPE = sequence[State]

program(aa:Behavior):bool =
  aa(0) = startstate
    AND
  FORALL (n:nat): transition(aa(n),aa(n+1))
```

Sequences are infinite lists of elements, represented as functions of type: `[nat -> State]`. The protocol theory contains a definition of the `startstate` (a particular record not shown).

The Correctness Criteria in PVS. We mentioned earlier that the Murϕ-to-PVS translator generates two theories: `protocol` above, and the theory `protocol_safe` containing the correctness criteria to be proved. The theory defines two predicates and a theorem: the correctness criteria. The predicate `invariant` takes as argument a predicate `p` on states (`pred[State]` is short for the function space [State -> bool]). It returns TRUE if for all execution traces aa of the program: the predicate p holds in every position of that trace. Note that the program referred to here is the one defined in protocol.

```
protocol_safe : THEORY
BEGIN
  IMPORTING protocol

  invariant(p:pred[State]):bool =
    FORALL (aa:Behavior):
      program(aa) IMPLIES
        FORALL (n:nat): p(aa(n))

  safe(st:State):bool =
    SAFE(st)

  correct : THEOREM invariant(safe)
END protocol_safe
```

The safety property we want to verify for each reachable state is defined by the predicate safe, which again states that the variable SAFE evaluates to TRUE. The correctness criteria is defined by the formula named correct. Obviously, the invariant safe in formula correct needs to be greatly strengthened in order to be provable, and this invariant strengthening is the real challenge of the proof.

57 invariants were needed in the proof. Their formulation in Murϕ is included in appendix C. In a second proof attempt, several proof optimizations led to a significant reduction in effort. In comparison with the work of Helmink, *et al* [10], the invariants used are roughly the same but their invariants were discovered by hand in advance of a mechanical proof, whereas we used the PVS proof to guide the discovery of invariants. Rerunning the proof takes 5 hours.

The verification of the protocol has been automated as far as possible by defining a set of tactics (occupying five pages). This level of automation could be achieved primarily because of the flexibility afforded by the PVS decision procedures.

5 Abstraction: Reduction to Finite State Using PVS

In recent work [21] a Boolean mu-calculus model checker [12] has been integrated into PVS as a decision procedure. This integration uses a relational mu-calculus (quantified Boolean formulas with least and greatest fixpoints of monotone Boolean predicate transformers) as a medium for communicating between PVS and the model checker for the Boolean mu-calculus. In this section we shall see how to apply this integration to our protocol.

The mu-calculus of a given state type can be formalized within the higher-order logic by defining least and greatest fixpoint operators for monotone predicate transformers. More usefully, the temporal operators of the branching time logic CTL can be defined using this mu-calculus. In particular, we can define the

CTL `AG` operator which represents the modality: "for every path, and for every state along that path" of CTL. The assertion `AG(i,n)(p)`, where `i` is the initialization predicate and `n` is the next-state relation, holds if `p` holds in all reachable states; the latter notion is defined in terms of the least fixpoint operator `mu`.

When the state type is finite, i.e., constructed inductively from the booleans and scalar types using records, tuples, or arrays over subranges, the PVS mu-calculus (and the corresponding CTL) can be translated into the Boolean mu-calculus and model checking can be used as a decision procedure for this fragment, using just boolean variables and BDDs. The PVS proof command `model-check` carries out these verification steps automatically. The resulting model checker by itself has few advantages over a conventional model checker. The main advantage is when it is combined with theorem proving to exploit the use of abstraction to reduce unbounded state spaces to finite ones. Abstraction is well studied in the literature [4, 7, 15], but the reasoning is usually carried out informally.

In order to prove an `AG` property, there is a simple way to define an abstraction, as shown in [4] and recalled in [21]. Suppose we are given a concrete, possibly infinite state, system S_c (like our protocol) defined by a state type, an initialization predicate and a next-state transition relation over the state. That is: $S_c = \langle \Sigma_c, I_c, N_c \rangle$, where $I_c : \Sigma_c \to \mathcal{B}$ and $N_c : (\Sigma_c \times \Sigma_c) \to \mathcal{B}$. Suppose further that we want to verify the property $p_c : \Sigma_c \to \mathcal{B}$ about the concrete system, that is, $AG(I_c, N_c)(p_c)$. We define an abstract system: $S_a = \langle \Sigma_a, I_a, N_a \rangle$ and an abstract property p_a, together with an abstraction mapping $h : \Sigma_c \to \Sigma_a$. We then show that the abstract system satisfies the abstract property, and that the mapping preserves the initialization predicate, the next-state relation and the property. This is expressed by the following theorem which has been proved using PVS:

Theorem 1. *In order to prove $AG(I_c, N_c)(p_c)$ it is sufficient to prove that:*

1. $\forall s : \Sigma_c.\ I_c(s) \Rightarrow I_a(h(s))$
2. $\forall s_1, s_2 : \Sigma_c^r.\ N_c(s_1, s_2) \Rightarrow N_a(h(s_1), h(s_2))$ *where Σ_c^r denotes all the concrete states reachable from an initial state via the next-state relation.*
3. $\forall s : \Sigma_c.\ p_a(h(s)) \Rightarrow p_c(s)$
4. $AG(I_a, N_a)(p_a)$

We saw that there were three sources of unboundedness in BRP: the file size, the message data, and the retransmission bound. These can be abstracted away to obtain a property-preserving abstraction. The main trick is that since we are dealing with ACTL properties, i.e., those that involve only universal path quantification, it is okay to introduce additional nondeterminism at the abstract level. In particular, the size of the as yet untransmitted portion of the file can be abstracted to one of `NONE`, `ONE`, or `MANY`, the message data is irrelevant and can be eliminated, and the retransmission bound can be replaced by a nondeterministic choice between the continuation and termination of retransmission. As a result, the type `Data` is removed. A file is no longer an array of `Data` but an element of the enumerated type {`NONE,ONE,MANY`}, indicating whether the file has no elements, one element or more than one element. The constant `max` is removed: there is no longer an upper bound for the number of retransmissions. This is possible since we just require that whenever the concrete protocol can make a transition, the abstract protocol must also be able to, and surely, if we remove the upper bound, then this is guaranteed. The counter `rn` itself that counts the number of retransmissions is turned into a single boolean: it is true only if a message has been sent at least once. The file component (`afile`) in the `automaton` is just a boolean, indicating whether it is empty or not: whether there is more to send or not.

```
abstract_protocol : THEORY
  BEGIN
    File  : TYPE = {NONE,ONE,MANY}
    Msg   : TYPE = [# first: bool, last: bool, toggle: bool #]
    State : TYPE = [# file : File, rn : bool, afile : bool, ... #]
```

The `automaton` specification also changes, but we omit the details of this new automaton specification. Concerning the program itself, in the abstract version of f.ex. the transition rule `Rule_write_K`, the update of `K(last)` has been changed from `K(last) := (head(st)=last)` to `K(last) := (file(st)=ONE)`. The update of `rn` has changed, and the update of `K(data)` has now disappeared.

The abstraction mapping between the concrete state type and the abstract state type is then given by the function `abs` defined below.

```
abstraction : THEORY
BEGIN
  A  : THEORY = abstract_protocol
  C  : THEORY = protocol

  abs(s:C.State):A.State =
    (# file  := IF    head(s)=nil  THEN NONE
                ELSIF head(s)=last THEN ONE ELSE MANY ENDIF,
       rn    := rn(s)/=0,
       afile := ahead(s)/=nil, ... #)
  ...
END abstraction
```

Theorem 1 can now be used to prove the invariant asserted in the theorem `concrete_correct` shown below.

```
concrete_correct : THEOREM
  AG(C.initial,C.transition)(safe)
```

Recall that Theorem 1 was defined in the context of a concrete system S_c (`protocol` in our case), an abstract system S_a (`abstract_protocol`) and an abstraction mapping h (`abs`) between the two. Theorem 1 tells us that it is sufficient to prove the propositions corresponding to items 1–4 in the theorem. The first and third of these are easily proved by the `grind` proof strategy of PVS. The fourth is proved by the `model-check` proof strategy of PVS. The second proof obligation is the only nontrivial one and it is shown below as `transition_preserves`.

```
transition_preserves : PROPOSITION
  FORALL (s1,s2:C.State):
    (I(s1) AND I(s2) AND C.transition(s1,s2))
      IMPLIES
    A.transition(abs(s1),abs(s2))
```

The proof obligation `transition_preserves` only requires the abstract system to simulate a step of the concrete system when a concrete invariant `I` holds of the initial and final states of the transition. Indeed, we do require the use of a specific concrete invariant in justifying the abstraction, and this is currently proved directly in PVS without the aid of abstraction. We, in fact, reuse this invariant from the proof of BRP-PVS in the Section 4.

It took two weeks to define the abstract protocol and the abstraction mapping, and to carry out the proof of the main lemma: `transition_preserves`. It takes two hours to rerun the proof of this lemma. However, this lemma again uses a number of invariants about the concrete protocol; we need 45 of the 57 invariants invented in the original invariant proof in Section 4. Considering the extra two-week effort in carrying out this abstraction proof, one may conclude that no essential work effort has been saved by doing the abstraction with model checking in comparison with the pure invariant proof. We claim, however, that the abstraction gives a good intuition about the functioning of the protocol, since it focuses on control. Also, it is likely that many of the prior invariance proofs can also be proved via abstraction and model checking.

The final proof obligation corresponding to item 4 of theorem 1 is proved by means of model checking. Using the PVS command `model-check`, this lemma is proved in 36.7 minutes. Our initial experiences in applying the PVS model checker to this example were not satisfactory. This led us to reformulate the abstract protocol in Murϕ and in the CTL model checker SMV, in order to compare execution times. In Murϕ, the verification took 28.51 seconds. The main source of efficiency in Murϕ with respect to this example is that it uses an explicit (not symbolic) representation of the state space and only explores the reachable states. Our initial definition of the `AG` operator was such that it explored all those states that could potentially lead to a state violating the invariant in order to check if these states included the start state. Obviously, many more states were explored in this manner.

When we apply the SMV model checker [17] to the example, it takes 2.5 hours to verify the abstract protocol. When SMV is used with the option '-f', only the 'forward' reachable states are examined, and this gives an impressive improvement in efficiency taking only 24 seconds — an improvement factor of nearly 400.

6 Observations

As a general remark, we want to stress a basic result of the work: to provide a nontrivial example/methodology for reasoning about protocols using theorem proving and model checking. Our framework is adequate for deeper studies of the problems we have encountered, such as the identification of suitable invariants and the design of useful finite-state abstractions.

We have found it useful and productive to employ a state exploration tool such as Murϕ as a prelude to full theorem proving. Murϕ was also useful for checking putative invariants.

In [10] a refinement between the protocol and an abstract protocol was defined and verified. Our contribution has been to reformulate refinement as a safety property by superposing the implementation and specification of the protocol. This technique seems simple and yet useful.

The infinite state PVS specification initially took three man-months to verify. This is comparable to the work in Coq [10] where they were starting from a hand-written proof. We believe though that our proof is more automated than the Coq proof: for any invariant proof, once we had identified (and included as assumptions) which other sub-invariants it depended on, the proof was automatic using a specially designed tactic based on decision procedures. With our improved approach of strengthening invariants on the fly, we were able to reduce the proof effort to four weeks.

Our use of abstraction yields a better understanding of the protocol since it extracts out the relevant control skeleton. We hope that a deeper study will reveal systematic techniques for obtaining such abstractions.

References

1. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260, 261, May 1969.
2. Rachel Mary Cardell-Oliver. The formal verification of hard real-time systems. Technical Report 255, University of Cambridge Computer Laboratory, 1992.
3. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.
4. E.M. Clark, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
5. C. Cornes, J. Courant, J.C. Filliatre, G. Huet, P. Manoury, C Paulin-Mohring, C. Munoz, C. Murthy, C. Parent, A. Saibi, and B. Werner. The Coq proof assistant reference manual, version 5.10. Technical report, INRIA, Rocquencourt, France, February 1995. This version is newer than the version used to verify the BRP-protocol in [10].
6. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Com-*

puter Science, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

7. Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 561–581, 1994.
8. M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, Dordrecht, The Netherlands, 1988.
9. J. F. Groote and J. C. van de Pol. A bounded retransmission protocol for large packets. A case study in computer checked verification. Logic Group Preprint Series 100, Utrecht University, 1993.
10. L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, Centrum voor Wiskunde en Informatica (CWI), Computer Science/Department of Software Technology, March 1994.
11. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.
12. G. Janssen. ROBDD software. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.
13. Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Trans. on S.W. Engg*, SE-10(4):325–342, July 1984.
14. L. Lamport. The Temporal Logic of Actions. Technical report, Digital Equipment Corporation (DEC) Systems Research Center, Palo Alto, California, USA, April 1994.
15. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.
16. N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth Annual Symposium on Principles of Distributed Computing, New York*, pages 137–151. ACM Press, 1987.
17. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.
18. R. Melton, D.L. Dill, and C. Norris Ip. Murphi annotated reference manual, version 2.6. Technical report, Stanford University, Palo Alto, California, USA, November 1993. Written by C. Norris Ip.
19. O. Müller and T. Nipkow. Combining model checking and deduction for i/o-automata. Technical University of Munich. Draft manuscript, 1995.
20. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
21. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In *Computer-Aided Verification (CAV) 1995, Liege, Belgium, Lecture Notes in Computer Science, Volume 939*, pages 84–97. Springer Verlag, July 1995.

A Specification Automaton In Murϕ

Note that certain constants and types are defined in the implementation automaton below.

```
Var
  abusy  : boolean; afile : File;
  ahead  : 1..last+1;
  afirst : boolean; aerror : boolean;

Procedure verify_REQ(f:File);
Begin Assert !abusy;
  abusy := true; afile := f; ahead := 1;
End;

Procedure verify_IND(d:Data;i:IndT);
Begin Assert
        abusy & !aerror & ahead != nil &
        d = afile[ahead] &
        i = (ahead=last ? LAST :
                (afirst ? FIRST :
                          INCOMPLETE));
  afirst := (ahead=last);
  ahead := ahead + 1;
End;

Procedure verify_IND_ERR();
Begin Assert aerror;
  afirst := true; aerror := false;
End;

Procedure verify_CONF(c:Conf);
Begin Assert
         abusy & !aerror &
         (c=OK -> ahead=nil) &
         (c=DONT_KNOW  ->
           (ahead=nil|ahead=last)) &
         (c=NOT_OK  -> ahead != nil);
  abusy := false; aerror := !afirst;
  ahead := nil;
End;
```

B Implementation Automaton in Murϕ

```
Type
  Data : boolean;
  Msg  : Record
           first, last, toggle : boolean;
           data : Data
         End;
  Spc  : Enum{WR,SF,WA,SC,WT2};
  Conf : Enum{OK,NOT_OK,DONT_KNOW};
  File : Array [1..last] Of Data;
  Rpc  : Enum{WF,SI,SA,RTS,NOK};
  IndT : Enum{FIRST,INCOMPLETE,LAST};

Const
  max  : 2; last : 3; nil : last+1;

Var
  K : Msg; K_full : boolean; L : boolean;
  req : File; req_full : boolean;
  conf : Conf; conf_full : boolean;
  spc : Spc; sfirst : boolean;
  stoggle : boolean; file : File;
  head : 1..last+1; rn : 0..max;
  stimer1_on : boolean;
  stimer1_enabled : boolean;
  stimer2_on : boolean;
  stimer2_enabled : boolean;
  ind_data : Data; ind_indication : IndT;
  ind_full : boolean; ind_error : boolean;
  rpc : Rpc; rfirst : boolean;
  rtoggle : boolean; ctoggle : boolean;
  msg : Msg;
  rtimer_on : boolean;
  rtimer_enabled : boolean;

Ruleset d1 : Data; d2 : Data; d3 : Data Do
  Rule "write_req" !req_full ==>
    req_full := true;
    req[1] := d1; req[2] := d2; req[3] := d3;
  End;
End;

Rule "read_conf" conf_full ==>
  conf_full := false;
End;

Rule "read_ind" ind_full ==>
  ind_full := false;
End;

Rule "read_ind_error" ind_error ==>
  ind_error := false;
End;

Rule "lose_msg" K_full ==>
  K_full := false;
  stimer1_enabled := true;
End;

Rule "lose_ack" L ==>
  L := false; stimer1_enabled := true;
End;

Rule "read_req" spc = WR & req_full ==>
  req_full := false;
  For i := 1 To last Do file[i] := req[i] End;
  head := 1; spc := SF;
  verify_REQ(req);
End;

Rule "write_K" spc = SF & !K_full ==>
  K_full := true; K.first := sfirst;
  K.last := (head = last);
  K.toggle:= stoggle; K.data := file[head];
  spc:= WA; rn := rn+1; stimer1_on := true;
```

```
End;

Rule "read_L" spc = WA & L ==>
  L := false;
  If head = last Then
    spc := SC;
  Else
    spc := SF;
  End;
  sfirst := (head = last);
  If !(head = last) Then rn := 0 End;
  head := head + 1; stoggle := !stoggle;
  stimer1_on := false;
  stimer1_enabled := false;
End;

Rule "write_conf" spc = SC & !conf_full ==>
  conf_full := true;
  If head = nil Then conf := OK
  Elsif head = last & rn != 0 Then
    conf := DONT_KNOW;
  Else conf := NOT_OK; End;
  If head = nil Then spc := WR; Else
    spc := WT2;
    sfirst := true; stoggle := !stoggle;
    stimer2_on := true;
    rtimer_enabled := true;
  End;
  head := nil; rn := 0;
  verify_CONF(conf);
End;

Rule "stimer1" stimer1_on & stimer1_enabled
  ==>
  If rn = max Then
    spc := SC;
  Else
    spc := SF;
  End;
  stimer1_on := false;
  stimer1_enabled := false;
End;

Rule "stimer2" stimer2_on & stimer2_enabled
  ==>
  spc := WR; stimer2_on := false;
  stimer2_enabled := false;
End;

Rule "read_K" rpc = WF & K_full ==>
  K_full := false;
  msg.first := K.first; msg.last := K.last;
  msg.toggle := K.toggle; msg.data := K.data;
  If !ctoggle | (msg.toggle = rtoggle) Then
      rpc := SI; rtimer_on := false;
      rtimer_enabled := false;
  Else
    rpc := RTS;
  End;
End;

Rule "write_ind" rpc = SI & !ind_full ==>
  ind_full := true; ind_data := msg.data;
  If msg.last Then ind_indication := LAST;
  Elsif msg.first Then
    ind_indication := FIRST;
  Else ind_indication := INCOMPLETE;
  End;
  rpc := SA; rfirst := msg.last;
  ctoggle := true; rtoggle := !msg.toggle;
  verify_IND(ind_data,ind_indication);
End;

Rule "write_L" (rpc = SA | rpc = RTS) & !L
  ==>
  L := true;
  If rpc = SA Then rtimer_on := true; End;
  rpc := WF;
End;

Rule "write_ind_error" rpc = NOK & !ind_error
  ==>
  ind_error := true; rpc := WF; rfirst := true;
  rtimer_on := true; stimer2_enabled := true;
  verify_IND_ERR();
End;

Rule "rtimer" rtimer_on & rtimer_enabled ==>
  ctoggle := false;
  If !rfirst Then
    rpc := NOK; rtimer_on := false;
  Else
    stimer2_enabled := true;
  End;
  rtimer_enabled := false;
End;
```

C Invariants

```
Invariant "ref_abusy"
  abusy = (spc = SF | spc = WA | spc = SC);

Invariant "ref_afirst"
  afirst = rfirst;

Invariant "ref_aerror"
  aerror = ((rpc = NOK) |
           (spc = WT2 & rtimer_on & !rfirst));

Invariant "ref_afile"
  Forall i:1..last Do afile[i]=file[i] Endforall;

Invariant "ref_ahead"
  ahead =
   ((spc = WT2 |
    (ctoggle -> stoggle=rtoggle)) ? head : head+1);

Invariant "spc_1"
  (spc = WR | spc = SF | spc = SC) -> rpc = WF;

Invariant "spc_2"
  (spc = SF & rn = 0) ->
    (ctoggle -> stoggle = rtoggle);

Invariant "spc_3"
  spc = WA -> rn != 0;

Invariant "spc_4"
  spc = WT2 & !rtimer_enabled -> !ctoggle;

Invariant "spc_5"
  spc=SC & head = nil ->
   (ctoggle & stoggle = rtoggle);

Invariant "spc_6"
  spc = SC & head = nil -> rfirst;
```

```
Invariant "spc_7"
  spc = WR -> (ctoggle -> stoggle = rtoggle);

Invariant "spc_8"
  spc = WT2 -> rn = 0;

Invariant "spc_9"
  (spc = SC & rn = 0) ->
    (ctoggle -> stoggle = rtoggle);

Invariant "spc_10"
  spc = SF ->
   ((ctoggle -> stoggle = rtoggle) ->
    sfirst = rfirst);

Invariant "spc_11"
  spc = WR ->
   ((ctoggle -> stoggle = rtoggle) ->
    sfirst = rfirst);

Invariant "spc_12"
  spc = WA ->
   ((ctoggle -> stoggle = rtoggle) ->
    sfirst = rfirst);

Invariant "spc_13"
  spc = SC ->
   ((ctoggle -> stoggle = rtoggle) ->
    sfirst = rfirst);

Invariant "spc_14"
  spc = WT2 -> sfirst;

Invariant "spc_15"
  spc = WA -> head !=  nil;

Invariant "spc_16"
  spc = SF -> head !=  nil;

Invariant "rpc_1"
  (rpc  =  SI | rpc  =  SA | rpc  =  RTS)
    ->
  spc  =  WA;

Invariant "rpc_2"
  rpc  =  NOK -> spc  =  WT2;

Invariant "rpc_3"
  rpc = SI -> msg.last = (head = last);

Invariant "rpc_4"
  rpc = NOK -> !ctoggle;

Invariant "rpc_5"
  rpc = SI -> stoggle = msg.toggle;

Invariant "rpc_6"
  (rpc = SA | rpc = RTS) -> stoggle = msg.toggle;

Invariant "rpc_7"
  (rpc = SA | rpc = RTS) ->
   (ctoggle & msg.toggle !=  rtoggle);

Invariant "rpc_8"
  (rpc = WF | rpc = RTS) -> rtimer_on;

Invariant "rpc_9"
  rpc = SI -> (ctoggle -> msg.toggle = rtoggle);

Invariant "rpc_10"
  rpc = SI -> msg.data = file[head];

Invariant "rpc_11"
  rpc = SI -> rfirst = msg.first;

Invariant "K_1"
  K_full
    ->
  (spc  =  WA & rpc  =  WF & !L);

Invariant "K_2"
  K_full -> K.last = (head = last);

Invariant "K_3"
  K_full -> K.toggle = stoggle;

Invariant "K_4"
  K_full -> K.data = file[head];

Invariant "K_5"
  K_full -> ((ctoggle -> K.toggle = rtoggle) ->
            K.first = rfirst);

Invariant "L_1"
  L
    ->
  (spc  =  WA & rpc  =  WF & !K_full);

Invariant "L_2"
  L -> ctoggle & (!rtoggle) = stoggle;

Invariant "stimer1_1"
  stimer1_enabled
    ->
  (spc  =  WA & rpc  =  WF & !K_full & !L);

Invariant "stimer2_1"
  stimer2_enabled
    ->
  (spc  =  WT2 & rpc  =  WF & !K_full & !L);

Invariant "stimer2_2"
  stimer2_enabled -> rfirst;

Invariant "stimer2_3"
  stimer2_enabled -> rfirst;

Invariant "rtimer_1"
  rtimer_enabled
    ->
  (spc  =  WT2 & rpc  =  WF &
   !K_full & !L & !stimer2_enabled);

Invariant "rn_1"
  rn !=  0 -> spc !=  WR;

Invariant "rn_2"
  (rn !=  0 & ctoggle & stoggle !=  rtoggle) ->
    rfirst = (head = last);

Invariant "head_1"
  head = nil -> (spc = WR | spc = WT2 | spc = SC);

Invariant "safe_1"
  spc = WR -> !abusy;

Invariant "safe_2"
  spc = SC -> abusy;

Invariant "safe_3"
  spc = SC -> !aerror;

Invariant "safe_4"
  spc = SC ->
    (head = nil -> ahead = nil)
      &
    ((head = last & rn !=  0) ->
     (ahead = nil | ahead = last))
      &
    ((head = last & rn = 0) -> ahead !=  nil)
      &
    ((head < last) -> ahead !=  nil);

Invariant "safe_5"
  rpc = SI -> abusy;

Invariant "safe_6"
  rpc = SI -> !aerror;

Invariant "safe_7"
  rpc = SI -> ahead !=  nil;

Invariant "safe_8"
  rpc = SI -> msg.data = afile[ahead];

Invariant "safe_9"
  rpc = SI ->
   ((msg.last = (ahead = last)) &
    (msg.first = afirst));

Invariant "safe_10"
  rpc = NOK -> aerror;
```

This article was processed using the LaTeX macro package with LLNCS style