Experiments in Theorem Proving and Model Checking for Protocol Verification

K. Havelund[1]
LITP, Institut Blaise Pascal
4, place Jussieu
75252 Paris Cedex 05 France
havelund@litp.ibp.fr
Phone: +33 (1)44.27.40.55

N. Shankar[2]
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
Phone: +1 (415) 859-5272

September 20, 1995

[1]Supported by a European Community HCM grant, with origin institution being DIKU, Institute of Computer Science, University of Copenhagen, Denmark.
[2]Supported by NSF Grant CCR-930044 and by ARPA through NASA Ames Research Center under Contract NASA-NAG-2-891 (ARPA Order A721).

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 Helmink, Sellink, and Vandraager. 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. A large part 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.

Contents

Chapter 1

Introduction

Communication protocols are an important class of concurrent algorithms that pose a difficult challenge for existing verification technologies [13]. 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 an effective and general methodology for protocol verification.

The specific protocol we examine is the bounded retransmission protocol (BRP) from Philips [11]. 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 using the Coq proof checker [5] using the framework of Lynch and Tuttle's I/O automata [18]. Their hand proof effort occupied two man-months, and it took them three man-months to mechanize this proof using Coq. 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 answer 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 [23] 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 [24] 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 [14], Murϕ [20], and SMV [19] on this example. The PVS proof of the abstraction mapping took two man-weeks. 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 work 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 not dramatic at this point, but we believe that a dramatic saving can indeed be achieved through a more aggressive application of our proposed methodology. Even as verified, the BRP example demonstrates how the combined use of theorem proving and model checking results in a more insightful and direct verification than can be obtained by either theorem proving or model checking when used in isolation. It also illustrates the value of tightly integrating theorem proving and model checking as has recently been achieved in PVS [24].

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, original verification of Helmink, Sellink, and Vandraager [11]. 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 [16]. 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 considerable simplification in the proof and a valuable technique for other protocol correctness proofs.

Müller and Nipkow [22] 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, however, 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 [10] 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 [15] 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 studied and developed by Clarke, Grumberg, and Long [4], by Dams, Grumberg, and Gerth [7], and by Loiseaux, Graf, Sifakis, Bouajjani, and Bensalem [17].

Chapter 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* of 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: 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 2.1. 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. Communication is asynchronous: K and L are actually unbounded buffers but since there is at most one message active in a channel at any point in time, these are modeled here for simplicity as one-place buffers. This simplification is also present in the work of Helmink, *et al* [11].

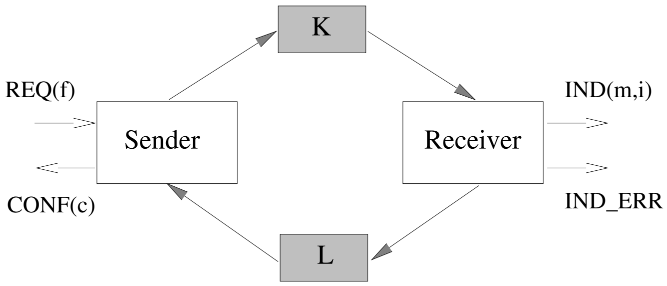

Figure 2.1: The BRP Protocol

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 acknowledgment. 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* [11].

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 acknowledgments are lost. The sender sends each individual message m 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 acknowledgment 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.

One complication in the protocol is that upon abortion of a file transmission, the sender and the receiver must regain synchrony for the next file transmission. If the retransmission bound for a message has been reached without an acknowledgment, then the file transmission is aborted. The sender must then delay for a sufficiently long interval before transmitting the next file so that the receiver's timer can time out and the receiver can hence determine that file transmission has been aborted. Another complication is that when file transmission is aborted, then it is possible that the next message on channel K (namely, the first message of a new file) may not have its alternating bit properly toggled. This is because the receiver may have received the earlier message but the sender did not receive an acknowledgment. To cope with this loss of synchronization, the receiver resets its toggle bit to that of the first message and uses the bit alternation for subsequent messages in the same file.

It is not evident how to formulate the correctness criteria for this protocol. First of all, we limit ourselves to safety properties (that some property always holds), since already these seem challenging enough when trying to do "real" proofs. Liveness properties (that some property eventually holds) are of course of big interest, but they are generally harder to prove. Safety properties are also sometimes called invariants. But even after having restricted ourselves to safety properties, the property to prove is not evident.

One solution is to state some invariant over the variables of the program. Suppose that the variables are x_1,...,x_n, then we could establish some predicate P over these variables, and then show that $P(x_1, \ldots, x_n)$ always holds. It is, however, difficult to come up with a predicate P that gives a good intuitive feeling of correctness.

The solution we shall choose mimics the correctness criteria defined in [11], where the protocol is proved to be a refinement of a more abstract protocol. We shall not prove a refinement, but we shall formulate the correctness criteria in terms of a more abstract protocol, the same as in [11] basically. Instead of proving a refinement, we let the abstract protocol "run in parallel" with the concrete one: for every occurrence of an external communication (REQ, IND, CONF, or IND_ERR) in the concrete protocol, the abstract protocol analyses whether this move is allowed, and a flag is raised in case the "laws" of the abstract protocol are broken.

The abstract protocol only specifies how the events REQ(f), IND(m,i), CONF(c) and IND_ERR can occur, ignoring the details of the channels K and L. That is, it only focuses on the communications with the environment according to the black box view. We shall see below how this is formulated. The parallel composition of the concrete and the abstract protocols has the big advantage that it can be model checked in the finite-state case.

Chapter 3

State Exploration in Murϕ

In this section we present the formulation of the protocol and its correctness criteria in Murϕ [20], a state exploration tool for finite-state transition systems. First we give a short introduction to Murϕ, then we present the protocol and the correctness criteria. The full formal Murϕ program is contained in appendix A.

3.1 Short Introduction to Murϕ

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.

3.2 Programming the Protocol

We shall only present part of the protocol in this section. The full protocol is contained in appendix A.1. First we present some shared declarations, then part of the program for the sender, and finally the abstract specification of the protocol. The receiver is modeled in a style similar to the sender.

3.2.1 Parameters and Shared Variables

The protocol we have described in the previous section can be said to be parameterized with three informations: 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 gives up a file. In fact, it turns out that when we choose the data domain to be finite, and if we let the size of files be limited and choose some value for max, then the protocol gets finite-state and it can be formulated and verified in Murϕ. We choose to let data be booleans, and we choose the size of files to be 3, and we choose the retransmission limit to be 2:

```
Type
  Data : boolean;
Const
  last : 3;
  max  : 2;
```

A file is modeled as an array of data, of size 3, and a variable contains the current number of retransmissions tried:

```
Type
  File : Array [1..last] Of Data;
Var
  rn : 0..max;
```

Recall that channel K carried tuples containing a first, last, toggle and data field. Channel L just carried a signal:

```
Type
  Msg  : Record
           first, last, toggle : boolean;
           data : Data
         End;
Var
  K : Msg;
  K_full : boolean;
  L : boolean;
```

Here three global variables are declared. Both channels are declared as variables. 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`. Hence, the sender sets it to true when writing to `K`, and the receiver sets it to false when reading from `K`. This flag-technique is general and used for variables that play the role of "channels".

3.2.2 The Sender

The sender control is managed via a program counter:

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

When the sender waits for a new file, `spc` equals `WF`. Value `SF`: the sender is ready to send the next message, just waiting for the `K_full` flag to be false. Value `WA`: wait for an acknowledgment to come back. Value `SC`: ready to send a confirmation. Value `WT2`: wait for timer 2 to timeout (which will eventually happen when the receiver has realized that the current file has been cancelled by the sender).

Requests to send files are written by the environment into the variable `req`:

```
Var
  req : File;
  req_full : boolean;
```

When the sender reads this request, it stores the request in it's own local request holder `file`:

```
Var
  file : File;
  head : 1..last+1;
```

The `head` points at any time to the message in the file that is currently being transmitted. In case the pointer exceeds `last`, the file is regarded empty.

The sender program contains two further variables, one indicating whether the current message is the first one, and one indicating the "alternating bit" used to distinguish messages:

```
Var
  sfirst  : boolean;
  stoggle : boolean;
```

Concerning timers, the sender uses two and the receiver uses one. Murϕ is not able to deal with timers explicitly, so they have to be modeled using the existing technology of transition rules and global variables. Also in [11] timers have been modeled, however, we use a different and in our opinion more clear modeling. For each timer, there are two variables. For example for timer 1 (of the sender) we have:

```
Var
  stimer1_on      : boolean;
  stimer1_enabled : boolean;
```

To begin with, these timer variables are false. When a timer is set, the 'on' variable is set to true. When (later) a timeout is allowed to occur, the 'enabled' variable is set to true. As an example, when both variables of timer 1 have been set to true, a timeout from timer 1 is allowed. We will see this illustrated in a moment.

Let us illustrate a few transition rules of the protocol. We focus on the sender. The following rule models how the environment "generates" a new request to the sender:

```
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;
```

The rule is "quantified" over the data of the request: any booleans may be chosen for `d1`, `d2` and `d3`. The rule is called `write_req`. It's pre-condition is that the `req` variable is "empty" (`req_full` is false – '`!`' means negation).

The next rule models how the sender reads this new request:

```
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;
```

The pre-condition requires that the sender's program counter is `WR`: "wait for a request". Also, the `req_full` flag must be true. 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 call of the `verify_REQ` procedure is a call to the abstract specification of the protocol. The call will result in certain assertions to be verified in the abstract protocol, and if they fail to hold, program execution will terminate. The abstract protocol hence "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 models external communications to the surrounding environment. So communications on channels `K` and `L` are for example not external.

The next rule models how the sender sends a message to the receiver: it writes the current data in the file to the `K` variable:

```
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 pre-condition states that a message can only be sent when the sender's program counter is `SF` and the variable `K_full` is false: note that this indicates that the `K` variable is "empty". When the rule is executed, the `K` variable is updated in 5 assignment statements. For example, `K.data` is assigned the value of the current (`head`) element in `file`. 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, timer 1 is set: if an acknowledgment does not arrive within a certain time period, a timeout will occur.

Since timers cannot be modeled directly we have to model them indirectly: we let the cause of the timeout set the 'enabled' variable to true. The cause of timer 1 timeouts is either the loss of a message or an acknowledgment. So we model the loss of a message as an action of the environment as follows:

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

Note how the enabled-variable is set to true. Now, a timeout can occur as modeled by the following rule:

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

The pre-condition of the rules requires that `stimer1_on` as well as `stimer1_enabled` must be true. If the number `rn` of retransmissions has reached the upper limit `max`, then the sender's program counter is set to `SC`: "send a confirmation". Otherwise, it is set to `SF`: try to send the message again.

There are other sender rules, and there are similar rules for the receiver, 17 in total.

3.3 The Correctness Criteria

The abstract protocol specification is written as part of the Murϕ-program and uses its own local types and variables. It can be conceived as an *automaton* in itself, that is triggered by the external events of the protocol. The following variables are used:

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

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). The variable `aerror` becomes true if a confirmation occurs before the last message has been transmitted.

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`.

```
Procedure verify_REQ(f:File);
Begin
  assert !abusy "REQ";
  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)) "IND";
  afirst := (ahead=last);
  ahead := ahead + 1;
End;

Procedure verify_IND_ERR();
Begin
  assert aerror "IND_ERR";
  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) "CONF";
  abusy := false;
  aerror := !afirst;
  ahead := nil;
End;
```

So this abstract protocol is supposed to have the "same behavior" as the protocol, except that there are no internal communications on `K` and `L`. Note how the `assert` statements function as "pre-conditions": when the protocol calls the abstract procedures (as a procedure is called), the `assert` condition in the relevant case is evaluated, and if it evaluates to false, the Murϕ-verification terminates, printing the string in quotes together with the trace of states that lead to the falsified assertion. In this way we have smoothly formulated a refinement relation in the model checker, one of the results of this work.

The abstract protocol works as follows: After a request has been received (`REQ`), the messages are sent (`IND(m,i)`) one by one, the first with `i=FIRST`, the second with `i=INCOMPLETE` and finally the last with `i=LAST` (note the special `?`-syntax for Murϕ if-then-else expressions). Then a confirmation `CONF(OK)` is produced. If an error occurs on the way, a `CONF(NOT_OK)` or `CONF(DONT_KNOW)` may be confirmed instead. In case the confirmed value is `NOT_OK`, the abstract protocol requires there to be more values to be transmitted: some really have been lost. If the confirmed value is `DONT_KNOW`, then either (| means or): all values have in fact transmitted (`ahead=nil=last+1`), or: the last one has not yet been transmitted (`ahead=last`). If a confirmation occurs before the last message has been transmitted, the `aerror` is set to true, and hence an `IND_ERR` action may occur.

As may already be clear from the above, running the Murϕ-verifier means the following: the verifier examines every possible execution trace of the concrete protocol. Each such trace will "include" calls of the abstract procedures whenever an external action is performed. If one of these calls evaluate an `assert` condition to false, the verifier terminates while indicating the error. When we verified the above protocol, no such error states occurred: the protocol was "correct". 450.801 states were explored. Transition rules were fired 1.555.912 times in 784,7 seconds.

Chapter 4

Theorem Proving: Proving Safety with PVS

In the previous chapter, we verified a finite-state version of the protocol. The advantage of the verification was that it was automatic. In this chapter, we shall carry out the full-scale proof for the complete infinite-state protocol. Since Murϕ cannot handle infinite-state programs, we need to carry out this proof in a general purpose theorem prover like PVS. 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 chapter, and get a corresponding PVS specification. This specification is, however, still finite-state since all the Murϕ-declarations have been translated directly. To obtain the infinite-state specification, we modify by-hand a few of the PVS-declarations. It is worthwhile noting that in principle we really only change a few declarations to obtain an infinite-state specification. It should be said, though, that in practice we have made additional modifications, which again have motivated changes to be applied to the translator itself. We will present the resulting PVS-specification and the finite-to-infinite changes needed, since other translator-issues here are irrelevant.

We shall first give a very short introduction to PVS. Then, we describe the PVS-specification obtained by the translation. Finally, we describe the correctness proof. The full formal texts associated with this chapter are contained in appendix B.

4.1 Short Introduction to PVS

PVS (Prototype Verification System) is an environment for writing specifications and developing proofs [23]. It serves as a prototype for exploring new approaches to mechanized formal methods. PVS has been strongly influenced by the observation that theorem proving capabilities can be employed to enrich the type system of a typed logic via the notion of predicate subtypes, and conversely, that this enriched type system facilitates expressive specifications and effective theorem proving. PVS has also been guided by the experience that much of the time and effort in verification is in debugging the initial specification or proof idea. A high bandwidth of interaction is useful at the exploratory level whereas more automated high-level proof strategies are desirable at an advanced stage of proof development. PVS has been used to verify several complex fault-tolerant algorithms, real-time and distributed protocols, and in several other applications. For more information, see SRI's formal methods www-page: `http://www.csl.sri.com/sri-csl-fm.html`.

4.1.1 The Specification Language

The PVS specification language builds on a classical typed higher-order logic. The base types provided by the PVS library consist of booleans, real numbers, rationals, integers, natural numbers, lists, and so forth. The primitive type constructors include those for forming function (e.g., `[nat -> nat]`), record (e.g., `[# a : nat, b : list[nat]#]`) , and tuple types (e.g., `[int, list[nat]]`). The type system of PVS includes *predicate subtypes* that consist of exactly those elements of a given type satisfying a given predicate. So that, for example, the subtype of positive numbers is given by the type {`n : nat | n > 0`}. Predicate subtypes are used to explicitly constrain the domains and ranges of operations in a specification and to define partial functions, e.g., division, as total functions on a specified subtype. In general, type checking with predicate subtypes is undecidable.[1] PVS contains a further useful enrichment to the type system in the form of *dependent* function, record, and tuple constructions where the type of

[1] PVS does have an effective type checker which easily checks for type correctness relative to the simple types. It generates proof obligations corresponding to predicate subtypes. The typical proof obligations can be automatically discharged by the PVS decision procedures. The provability of such proof obligations is the only source of undecidability in the PVS type system so that none of the benefits of decidable type checking are lost.

one component of a compound value depends on the value of another component. PVS terms include for example constants, variables, abstractions (e.g., `(LAMBDA (i : nat): i * i)`), applications (e.g., `mod(i, 5)`), tuple constructions (e.g., `(-5, cons(1, null))`), record constructions (e.g., `(# a := 2, b := cons(1, null) #)`), and record updates (e.g., `r WITH [a := 5, b := cons(3, b(r))]`). Note that the application `a(r)` is used to access the `a` field of record `r`. PVS specifications are packaged as *theories*. Theories can be parametric in types. Type parametricity (or *polymorphism*) is used to capture those concepts or results that can be stated uniformly for all types. PVS also has a facility for automatically generating abstract data type theories (containing recursion and induction schemes).

4.1.2 The Proof Checker

The PVS proof checker is intended to serve as a productive medium for debugging specifications and for constructing readable proofs. The human verifier constructs proofs in PVS by repeatedly simplifying a goal (conjecture) into subgoals using inference rules, until no further subgoals remain. A proof goal in PVS is represented by a sequent. PVS differs from most proof checkers in providing primitive inference rules that are quite powerful, including decision procedures for ground linear arithmetic. The primitive rules also perform steps such as quantifier instantiation, rewriting, beta-reduction, and boolean simplification. PVS has a simple strategy language for combining simple inference steps into more complicated proof strategies. In interactive use, when prompted with a subgoal, the user types in a proof command that either invokes a primitive inference rule or a compound proof strategy. For example, the `skolem` command introduces Skolem constants for universal-strength quantifiers whereas the `inst` command instantiates an existential-strength quantifier with its witness. The `lift-if` command invokes a primitive inference step that moves a block of conditionals nested within one or more sequent formulas to the top level of the formula. The `bddsimp` command invokes a compound propositional simplification decision procedure based on BDD's (Binary Decision Diagrams). Proofs and partial proofs can be saved, edited, and rerun. It is possible to extend and modify specifications during a proof; the finished proof has to be rerun to ensure that such changes are valid.

4.2 Applying a Murϕ-to-PVS Translator, and Becoming Infinite

Applying the translator to the Murϕ program yields two PVS theories. The first one (`protocol`) contains the protocol itself, including the abstract protocol automaton that polices it. 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 full formal PVS theories are contained in appendix B.1. Let us first have a look at `protocol`.

4.2.1 The Protocol in PVS

The Protocol (obtained by the translation) is in PVS represented as a theory (only part of this theory is shown here):

```
protocol : THEORY
BEGIN
  Data : TYPE = bool
  last : posnat = 3
  max  : posnat = 2

  Position : TYPE = {i: int | 1 <= i AND i <= last+1}
  File     : TYPE = ARRAY[Position -> Data]
  Msg      : TYPE = [# first  : bool,
                       last   : bool,
                       toggle : bool,
                       data   : Data #]
  Spc      : TYPE = {WR, SF, WA, SC, WT2}

  ...
END protocol
```

The theory contains definitions of constants, types, and (as we shall see, indicated by ...) functions, relations, etc. The above definitions correspond to the types and constants we highlighted from the Murϕ-program in the previous chapter. The first three lines in the theory defines the type `Data` of data , and the two constants: `last` (number of messages in a file), and `max` (maximal number of retransmissions) . In the Murϕ program these were defined to have certain finite values (`bool`, `3`, `2`), and these values have been translated to the PVS-theory. There is also the definition of the type `File` of files. Note that the keyword `ARRAY` in PVS only has a commentary purpose: the type expression `ARRAY[A->B]` is in fact equivalent to the total function space `[A->B]`. Messages `Msg` that are communicated over the `K` channel are records with 4 fields. The sender's program counter ranges over the enumerated type `Spc`.

It turns out, that if we change the three first definitions by removing the defining right hand sides, we get the infinite-state specification of the "real" protocol that we want. So we change these as follows:

```
Data : NONEMPTY_TYPE
last : posnat
max  : posnat
```

Now we have the infinite-state protocol which we can expose to a full formal verification using the PVS prover. Note that we indicate `Data` to be a nonempty type; as a result we avoid certain proof obligations. Let us explain the remaining contents of the theory. Recall, that the murphi program contained a set of global variables. The state is in PVS modeled as a record. Hence, the type of states is defined as follows (only variables mentioned in the previous chapter are illustrated):

```
State : TYPE =
  [# aerror          : bool,
     afirst          : bool,
     ahead           : Position,
     afile           : File,
     abusy           : bool,
     stimer1_enabled : bool,
     stimer1_on      : bool,
     rn              : Counter,
     head            : Position,
     file            : File,
     stoggle         : bool,
     sfirst          : bool,
     spc             : Spc,
     req_full        : bool,
     req             : File,
     L               : bool,
     K_full          : bool,
     K               : Msg,
     SAFE            : bool,
     ...
  #]
```

The state contains 34 variables in total. The `Counter` type is defined as `Counter: TYPE = {i: int | 0 <= i AND i <= max}`. The variable `SAFE` is initially `TRUE`, and can only be changed by the abstract automaton when this is called to verify an external action of the concrete protocol.

Before turning to the individual transition rules of the protocol, let us illustrate how the abstract protocol automaton is defined. Recall, that in the Murϕ-program it was defined as a collection of procedures. In PVS it is defined as a function over the state. As arguments, it takes a state, and an action (the action performed by the concrete protocol, and which is to be verified by the automaton). As result, it returns a new state, where the abstract variables of the automaton have been changed, and where perhaps (in case the protocol is inconsistent with the automaton), the `SAFE` variable has been set to `FALSE`. The invariant that we later want to verify is exactly, that `SAFE` is always `TRUE`. The type of abstract automaton actions is defined as an abstract data type:

```
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
```

Two further types `IndT` and `Conf` are needed. They contain the values communicated to the environment in case a message is transmitted, respectively when a confirmation is given (see previous chapter). The `DATATYPE` definition is short for a nested theory definition, which itself contains a number of definitions, amongst these the definition `Action : TYPE`, together with the constructors of this type: `REQ`, `IND`, `IND_ERR`, `CONF`. For example, `REQ` has the type `REQ : [File -> Action]`. Also destructors (ex: `req`) and predicates (ex: `REQ? : [Action -> bool]` are defined. With this type, the abstract protocol can now be defined as follows:

```
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 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.

Now to the transition rules. The first two rules correspond to the Murϕ-rules `write_req` and `read_req`:

```
Rule_write_req(st:State,f:File):State =
  IF NOT req_full(st) THEN
    LET
      st = st WITH [req_full := TRUE],
      st = st WITH [req := f]
    IN st
  ELSE
    st
  ENDIF

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
```

Note how the generic quantification over the data values in the Murϕ-program here is modeled as a parameter to the rule. The quantification occurs later. Note also how the read request rule applies the automaton.

Rules are generally modeled as functions of type `[state -> state]`; that is: they are deterministic. Non-determinism is only allowed at the outermost level where, in each execution step, *any* rule with a true precondition may be selected for execution. This decision to model rules as functions is in accordance with the semantics of UNITY [3]. Another choice would be to define rules as relations of type `[[state,state] -> bool]`, as is the case in TLA [16]. This techniques has in fact earlier been applied in PVS [25]. We chose however the functional style since we believed that the theorem prover would deal more efficiently with equalities of the form `new_state = Rule(old_state)` than with general predicates like `Rule(old_state,new_state)`. To say whether this is the case still requires some experiments. In any case, our encoding may be seen as being a special instance of the more general notion of temporal logic [9].

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
```

The behavior of the program is now modeled as 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 =
  (EXISTS (f: File):
     st2 = Rule_write_req(st1,f))
  OR st2 = Rule_lose_msg(st1)
  OR st2 = Rule_read_req(st1)
  OR st2 = Rule_write_K(st1)
  OR st2 = Rule_stimer1(st1)
  ...
```

Note how the non-deterministic generation of input requests to the protocol is modeled by an existential quantification.

To finish the picture, it needs to be said what is a program. The following definitions do exactly that.

```
IMPORTING sequences[State]

Behavior: TYPE = sequence

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

First, the theory of sequences is imported. Sequences are infinite lists of elements, in our case states. The `sequences[t:TYPE]` theory is parameterized with a type `t` and contains the type definition `sequence : TYPE = [nat -> t]`. Hence, the *behavior* of a program is a sequence of states: a total function from natural numbers to states – 0 is mapped to the first state, 1 to the second state, etc.

The protocol theory contains a definition of the `startstate` (a particular record not shown), and with this we can define what programs are. A program is a predicate on behaviors: a behavior satisfies a program (is an execution of the program), if the first state is the startstate, and if each two successive states are related by the transition relation.

4.2.2 The Correctness Criteria i PVS

We mentioned earlier that the Murϕ-to-PVS translator generates two theories: `protocol` above, and then the theory `protocol_safe` containing the correctness criteria to be proved:

```
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)

  invariant : THEOREM invariant(safe)

END protocol_safe
```

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`.

The safety property we want to verify is defined by the predicate `safe`. The correctness criteria is defined by the formula named `invariant`. It expresses that for every state, `st`, reachable during a program execution, `SAFE(st)` evaluates to `TRUE`. The next section describes how we carried out the proof of `correct`.

4.3 The Correctness Proof

We now sketch the correctness proof in PVS. Appendices B.2, B.3, B.4 and B.5 contain listings of the proof tactics (user defined proof commands), the proof scripts (expansion of what was typed during proof sessions), status of lemmas, and dependencies between lemmas.

The statement we wish to prove is the invariant `safe` in formula `invariant`, but obviously this invariant needs to be greatly strengthened in order to be provable. Since we cannot anticipate this strengthening, we state the invariant as `I` below with an open slot `Delta` where the strengthening can be introduced. During a proof we can add lemmas asserting that the auxiliary invariants are implied by `Delta`. Eventually we define Delta to be the conjunction of these auxiliary invariants, and similarly prove each of these auxiliary invariants.

```
Delta : pred[State]
I : pred[State] = Delta & safe
invariant : THEOREM invariant(I)
```

The operator `&` has been "lifted" from applying to booleans into applying to state predicates. This is shown below together with a similar lifting of implication, and a new function `pi`:

```
p,p1,p2 : VAR pred[State]

IMPLIES(p1,p2):bool =
  FORALL (st:State):p1(st) IMPLIES p2(st);

&(p1,p2):pred[State] =
  LAMBDA (st:State): p1(st) AND p2(st)

pi(p):bool =
  p(startstate)
    AND
  FORALL (st1, st2: State):
    (I(st1) AND p(st1) AND transition(st1, st2)) IMPLIES p(st2);
```

The `pi` function expresses the inductive step in proving some sub-invariant `p`, assuming that the invariant `I` holds in the pre-state. Now, to prove `invariant`, we need to prove the lemmas:

```
p_safe  : LEMMA pi(safe)
p_Delta : LEMMA pi(Delta)
```

We first prove `p_safe` in the form of the sequent:

```
  |-------
{1}   pi(safe)
```

This sequent has no antecedent formulas (above the line) and just one consequent formula (below the line) numbered `1`. A sequent represents the implication between the conjunction of the antecedents and the disjunction of the consequents. In the above case the antecedent is `TRUE`. By applying a proof strategy that initiates the induction, discharges the base case, eliminates universal quantification (by introducing the Skolem constants `st1!1` and `st2!1`), expands out the `transition`, and then case-splits according to the program actions, we get 17 sequents, one of which is:

```
{-1}   st2!1 = Rule_read_req(st1!1)
[-2]   Delta(st1!1)
[-3]   safe(st1!1)
  |-------
[1]   safe(st2!1)
```

At this point a pre-defined tactic is applied, which tries to prove the sequent automatically. Our experience is that all such sequents that are provable by syntactic deduction are proved by this tactic. However, when the tactic is applied to a sequent that is not provable solely by the tactic, a reduced sequent is returned which shows what remains to be proven and this gives a hint as to what invariant might be missing in the proof. In the above case such a sequent is returned:

```
{-1}   spc(st1!1) = WR
{-2}   req_full(st1!1)
{-3}   abusy(st1!1)
[-4]   Delta(st1!1)
{-5}   SAFE(st1!1)
  |-------
```

That is, we need to prove that the conjunction of the antecedents is `FALSE` (inconsistent). The only "interesting" variables in the sequent are `spc` and `abusy` so we conclude that `[-1]` and `[-3]` are inconsistent. Now, for each new invariant we add, we need to assume that it is implied by `Delta` (such that we can use it in proving the above sequent) and prove that also this is preserved by the transition relation. In total, the above unsolved sequent makes us add the following declarations:

```
safe_1(st):bool = spc(st)=WR IMPLIES NOT abusy(st)
d_safe_1 : LEMMA Delta IMPLIES safe_1
p_safe_1 : LEMMA pi(safe_1)
```

In PVS, the tree declarations above can be added during a proof (using the `add-declaration` command) without having to stop and restart the proof. Note that applying lemma `d_safe_1` at this point will prove the sequent since the antecedent will then contain `abusy(st1!1)` as well as `NOT abusy(st1!1)` and that evaluates to `FALSE`.

During the proof of `p_safe_1` other variants are again needed, and the same pattern is repeated. The technique allows mutually recursive proofs.

It turns out that some invariants are logical consequences of others, and that for these we can avoid reasoning about the transition relation and just prove the implication. As a matter of fact, our invariant `safe_1` is such a consequence: following [11], we namely define an invariant that describes a relation between the variables of the abstract `automaton` and the variables in the protocol, and `safe_1` is a consequence of this invariant. For example, the following relation for the abstract variable `abusy` is defined and proved to be an invariant:

```
ref_abusy(st):bool =
  abusy(st) = (spc(st) = SF OR spc(st) = WA OR spc(st) = SC)

d_ref_abusy : LEMMA Delta IMPLIES ref_abusy

p_ref_abusy : LEMMA pi(ref_abusy)
```

With this we can define `safe_1` as a consequence that is obviously true, so we don't need to prove `pi(safe_1)`. Hence we change the lemma `p_safe_1` to:

```
p_safe_1 : LEMMA ref_abusy IMPLIES safe_1
```

When all new invariants have been dealt with, the correctness proof can be finished. The `Delta` is now defined as the conjunction of all the new invariants, and we can then prove `p_Delta` using the `pi`-lemmas. Also, all the lemmas of the form `Delta IMPLIES ...` can now be proven. 58 invariants were needed in the proof, yielding 118 lemmas in total. Rerunning the proof takes less than 3 hours. The above proof methodology was in fact applied in the second proof attempt and the whole proof took only a few weeks.

The verification of the protocol has been automated as far as possible by defining a set of tactics (occupying five pages). One can say that the proof is automated "up to" lemmas. That is: if some sequent is provable, then it is proved automatically using a single tactic. If it is not provable, but true in the context of the protocol, we have to identify which extra lemma/invariant is needed. When we have defined this new lemma, it is *included* as an antecedent in the sequent we try to prove, and then again the rest is automatic (proved by a single tactic). The obtained level of automation could be achieved only because of the flexibility provided by the PVS decision procedures.

The far most interesting lemmas are `pi`-lemmas, where a predicate is stated to be preserved by the program, and where we have to prove this over the structure of the program: one sub case for each transition rule. The following strategy, when applied to a `pi`-lemma generates 17 (unproved) sequents, one for each of the 17 transition rules of the protocol:

```
(defstep preserved ()
  (then
    (expand "pi")
    (expand "preserved")
    (spread
      (split)
      ((grind)
       (then
         (skosimp)
         (expand "I")
         (expand "&")
         (flatten)
         (auto-rewrite-theory "protocol" :always? T)
         (auto-rewrite-theory "protocol_safe" :always? T)
         (stop-rewrite "Delta")
         (expand "transition")
         (split)))))
  "..."
  "...")
```

Tactics are written in LISP: the name of the tactic is `preserved`; it has no arguments `()`, and it's body is a sequence (`then`) of commands. Two strings follow the command: the first is an explanation of it's effect which can be obtained via the PVS online help-commands; the second is what is printed when it is activated during a proof.

First, the definition of `pi` is brought into place by `(expand "pi")`. The `(split)` will then generate two subgoals, one for the initial `startstate`, and one for the inductive step (`FORALL ...`). The `startstate` case is proved automatically using the PVS tactic `(grind)`: note that `spread` applies `(grind)` to the first subgoal of the split, and the inner `(then ...)` to the second subgoal – the inductive step. The inductive step is after skolemization further `split` into 17 subgoals, one for each transition rule disjunct in the definition of `transition`. One of these sequents is the one we saw earlier (see page 32), and which is repeated here:

```
{-1}   st2!1 = Rule_read_req(st1!1)
[-2]   Delta(st1!1)
[-3]   safe(st1!1)
  |-------
[1]   safe(st2!1)
```

In fact, we would like PVS to try out itself to prove each such sequent before "giving up" and apply for human interaction for some extra invariant to be discovered. The following tactic tries to prove such a sequent:

```
(defstep action ()
  (then
    (skolem!)
    (replace -1)
    (delete -1)
    (do-rewrite)
    (lift-if)
    (musimp)
    (grind NIL))
  "..."
  "...")
```

In those cases where no extra sub-invariants are needed as assumptions, this tactic proves the sequent corresponding to a transition rule. Now the complete tactic that is applied to `pi`-lemmas is defined as follows:

```
(defstrat pr ()
  (then
    (preserved)
    (action))
  "..."
  "...")
```

This tactic will first apply `(preserved)`, resulting in 17 subgoals, and then apply `(action)` to each of these. Those that are not proved are left to us to prove via invariant strengthening.

4.4 Invariant Checking in Murϕ and Weakest Preconditions

Two further techniques to ease the proof have been applied. These are described below.

4.4.1 Invariant Checking in Murϕ

Whenever a new invariant were added, before proving it in PVS, we model checked it within the Murϕ-program. In this way a certain confidence could be obtained before throwing more costly efforts into the PVS infinite-state proof. The full set of Murϕ invariants is contained in appendix B.6. As an example, the invariants `safe_1` and `ref_abusy` have the following formulation in Murϕ:

```
Invariant "safe1"
  spc=WR -> ! abusy;

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

The Murϕ verifier would (in principle) then follow all execution traces, and in each state verify that each invariant was true. The assertion construct that we saw earlier, and this invariant construct, are both derived syntax based on a uniform unique error construct.

4.4.2 Weakest Pre-conditions

The example proof we have just seen illustrated, shows how a single unproved sequent is returned by the tactic `action` in case the sequent (representing a particular transition rule) it is applied to fails to prove (see page 32). In general, however, several (from 1 to 10) such sequents were returned, and all would be examined to produce a guess for a missing invariant, making the guessing less obvious. This lead to the idea to generate the weakest pre-condition [8] of the relevant transition rule with respect to the invariant to be proven. In fact, in some cases this was done by hand on a piece of paper when inspiration was missing. To ease that process, we experimented with tactics that could automatically generate these weakest pre-conditions. In practice, the goal was to generate *one* unproved sequent instead of many; but reduced as much as possible so that it would become easy to make a guess about what minimal invariant was missing.

Let us first shortly recall the theory of weakest pre-conditions, and then put it into the context of our transition system. Weakest pre-conditions were introduced to give semantics to programming languages [8]. The mechanism was that of predicate transformers producing a weakest pre-condition from a pair consisting of a command and a post-condition. The intuition being that, the weakest pre-condition specifies all the initial states such that executing the command in one of these will result in a state satisfying the post-condition. Assume that R is some post-condition, and that c is some command, then by $wp(c, R)$ we denote the weakest pre-condition generated by the predicate transformer wp. As an example, the weakest pre-condition of an assignment statement is obtained by replacing (in the post-condition) all occurrences of the assigned variable by the right hand side of the assignment:

$$wp(x := e, R) \quad = \quad R\{x \mapsto e\}$$

So for example:

$$\begin{array}{rcl} wp(x := x - 1, x > 0) & = & x - 1 > 0 \\ & = & x > 1 \end{array}$$

Other examples, continued in this pseudo-programming style, are weakest pre-conditions of conditionals, sequential compositions and the skip-command that has no effect on the state:

$$\begin{array}{rcl} wp(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, R) & = & (e \Rightarrow wp(c_1, R)) \wedge (\neg e \Rightarrow wp(c_2, R)) \\ wp(c_1; c_2, R) & = & wp(c_1, wp(c_2, R)) \\ wp(\mathbf{skip}, R) & = & R \end{array}$$

In fact these four rules illustrate very well the rules we need for our purposes since no transition rule contains loop constructs (the only loop is at the outermost level).

As example, consider the following transition rule for timer 1 in the sender (we earlier saw a Murϕ-version of it):

```
Rule_stimer1(st:State):State =
  IF stimer1_on(st) AND stimer1_enabled(st) THEN
    LET
      st = IF rn(st) = max THEN
             LET st = st WITH [spc := SC] IN st
           ELSE
             LET st = st WITH [spc := SF] IN st
           ENDIF,
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF
```

In proving the invariant `p_ref_abusy` from above (page 33), the following sequent is generated (amongst 17) after splitting the transition relation:

```
  p_ref_abusy.2.11 :

  {-1}   st2!1 = Rule_stimer1(st1!1)
  [-2]   Delta(st1!1)
  [-3]   safe(st1!1)
  [-4]   ref_abusy(st1!1)
    |-------
  [1]   ref_abusy(st2!1)
```

Let us first apply the technique we have used before. This sequent is not provable, and applying the tactic `(action)` at this point hence returns two sequents:

```
p_ref_abusy.2.11.1 :

[-1]   Delta(st1!1)
{-2}   SAFE(st1!1)
{-3}   (stimer1_on(st1!1))
{-4}   (stimer1_enabled(st1!1))
{-5}   (rn(st1!1)) = (max)
{-6}   SC = SC
  |-------
{1}   abusy(st1!1)
{2}   spc(st1!1) = SF
{3}   spc(st1!1) = WA
{4}   spc(st1!1) = SC


p_ref_abusy.2.11.2 :

[-1]   Delta(st1!1)
{-2}   SAFE(st1!1)
{-3}   (stimer1_on(st1!1))
{-4}   (stimer1_enabled(st1!1))
{-5}   SF = SF
  |-------
{1}   abusy(st1!1)
{2}   spc(st1!1) = SF
{3}   spc(st1!1) = WA
{4}   spc(st1!1) = SC
{5}   (rn(st1!1)) = (max)
{6}   SC = SF
```

By studying these two sequents, we guess that the missing invariant is:

```
stimer1_1(st):bool =
  stimer1_enabled(st) IMPLIES spc(st) = WA
```

Let us now try to calculate the weakest pre-condition. So we return to the sequent:

```
p_ref_abusy.2.11 :

{-1}   st2!1 = Rule_stimer1(st1!1)
[-2]   Delta(st1!1)
[-3]   safe(st1!1)
[-4]   ref_abusy(st1!1)
  |-------
[1]   ref_abusy(st2!1)
```

The `Rule_stimer1` rule can be regarded as a command, and the formula `ref_abusy(st2!1)` below the line can be regarded as the post-condition. The information needed to prove this sequent is the weakest pre-condition of these two. The reader may check that the following is the weakest pre-condition:

```
(NOT (stimer1_on(st1!1) & stimer1_enabled(st1!1))) IMPLIES
    (abusy(st1!1) = (spc(st1!1) = SF OR
                     spc(st1!1) = WA OR
                     spc(st1!1) = SC))
AND
(stimer1_on(st1!1) & stimer1_enabled(st1!1)) IMPLIES
    abusy(st1!1)
```

It turns out, that we can generate this condition using the tactic:

```
(defstep generate-wp ()
  (then
    (replace*)
    (delete -1)
    (do-rewrite)
    (lift-if)
    (assert))
"..."
"...")
```

That is, the result of applying this tactic to the sequent above is:

```
  p_ref_abusy.2.11 :

  [-1]   Delta(st1!1)
  [-2]   SAFE(st1!1)
  [-3]   abusy(st1!1) =
           (spc(st1!1) = SF OR spc(st1!1) = WA OR spc(st1!1) = SC)
    |-------
  {1}   IF (stimer1_on(st1!1)) AND (stimer1_enabled(st1!1)) THEN
          abusy(st1!1)
        ELSE
          abusy(st1!1) =
            (spc(st1!1) = SF OR spc(st1!1) = WA OR spc(st1!1) = SC)
        ENDIF
```

We see that the `ELSE` branch trivially evaluates to true due to the inductive assumption in `[-3]`. Further applying the builtin tactic `(bddsimp)` — which does propositional simplification using BDD's — yields the one-sequent simplification:

```
p_ref_abusy.2.11 :

[-1]   Delta(st1!1)
[-2]   SAFE(st1!1)
{-3}   (stimer1_on(st1!1))
{-4}   (stimer1_enabled(st1!1))
  |-------
{1}   abusy(st1!1)
{2}   spc(st1!1) = SF
{3}   spc(st1!1) = WA
{4}   spc(st1!1) = SC
```

Now we can more conveniently make our guess about the missing invariant.

Chapter 5

Abstraction and Model Checking in the μ-calculus

In recent work [24] a propositional mu-calculus model checker [14] 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 propositional mu-calculus. In this section we shall see how to apply this integration to our protocol. First, we present the mu-calculus within PVS, then we outline the meta theory behind the proof that we shall carry out using the integration, and finally we apply the theory to our example. The full formal texts associated with this chapter are contained in appendix C.

5.1 Introduction to the μ-calculus within PVS

The theory `mucalculus` defines an operator `mu` that returns the least fixpoint of a monotone predicate transformer.

```
mucalculus[State:TYPE] : THEORY
  BEGIN
     PT : TYPE = [pred[State]->pred[State]]
     ...
     mu(pt:PT):pred[State] = least_fixpoint(pt)
  END mucalculus
```

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.

```
ctl[State:TYPE] : THEORY
BEGIN
  IMPORTING MU@mucalculus

  st,st0 : VAR State
  Q,i,p : VAR pred[State]
  n : VAR pred[[State,State]]

  reachable(i,n):pred[State] =
    mu(LAMBDA Q:
         LAMBDA st:
           i(st) OR
           EXISTS st0:
             Q(st0) AND n(st0,st))

  AG(i,n)(p):bool =
    FORALL st: reachable(i,n)(st) IMPLIES p(st)
END 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 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 propositional mu-calculus and model checking can be used as a decision procedure for this fragment, using just boolean variables and BDD's. 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, 17], but the reasoning is usually carried out informally because the model checkers were not hitherto integrated with theorem provers.

5.2 Abstraction Proofs, the General Case

In order to prove an `AG` property, there is a simple way to define an abstraction, as shown in [4] and recalled in [24]. 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. $AG(I_a, N_a)(p_a)$

Proof

Assume that the properties 1–4 above hold, and consider a trace of the concrete system: we shall prove that every state in this trace satisfies p_c. First we prove that for every state s of the trace, $h(s)$ is reachable in the abstract system via the abstract next-state relation. We do this by induction over the distance of the concrete states in the trace from the initial state in the trace, the initial state having distance 0 and the next-state relation increasing distance by one. That is, first observe the initial state s_0 in the concrete trace (base case). Due to 1, we have that also $h(s_0)$ is initial in the abstract system, hence reachable. As induction step, consider next some concrete state s_1 and concrete state s_2 such that $N_c(s_1, s_2)$ and s_1 has distance n from the initial state. Then due to 2, also $N_a(h(s_1), h(s_2))$; and since s_1 has distance n: $h(s_1)$ is reachable in the abstract system due to the induction hypothesis, and hence also $h(s_2)$ is reachable in the abstract system.

So there exists an abstract trace such that for every state s in our concrete trace, $h(s)$ is in the abstract trace. Now consider such an arbitrary concrete s: due to 4 we have that $p_a(h(s))$, and then due to 3 we then have that $p_c(s)$.

∎

5.3 An Abstraction of the Protocol

In this section we present the finite-state abstraction of the protocol. The full formal PVS theories are contained in appendix C.1.

Types, Constants and Variables:

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 are 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. Finally, we need a special variable in the state `data_is_safe`, which represents the fact that a data value being transmitted is the right one.

```
abstract_protocol : THEORY
BEGIN
  IMPORTING ctl

  File : TYPE = {NONE,ONE,MANY}
  Msg  : TYPE = [# first: bool, last: bool, toggle: bool #]

  State : TYPE =
    [# afile        : bool,
       data_is_safe : bool,
       rn           : bool,
       file         : File,
       req          : File,
       SAFE         : bool,
       ...
    #]

  ...
END abstract_protocol
```

The Automaton:

The `automaton` specification also changes but this is acceptable since Theorem 1 admits an abstract property p_a distinct from the concrete property p_c. The data type of actions changes and becomes:

```
Action : DATATYPE
  BEGIN
    REQ:REQ?
    IND(ind:IndT):IND?
    IND_ERR:IND_ERR?
    CONF(conf:Conf):CONF?
  END Action
```

It has been changed in two ways. First, the `REQ` constructor no longer takes a file as parameter: the automaton no longer cares about this information. Second: data have been removed from indications (`IND`). The automaton itself has also been changed correspondingly.

```
automaton(st:State,action:Action):State =
  CASES action OF
    REQ:
      LET
        st = st WITH [SAFE := NOT abusy(st)],
        st = st WITH [abusy := TRUE],
        st = st WITH [afile := TRUE]
      IN st,
    IND(i):
      LET
        st = st WITH [SAFE :=
          abusy(st) AND NOT aerror(st) AND
          afile(st) AND data_is_safe(st) AND
          (i=FIRST IMPLIES afirst(st)) AND
          (i=INCOMPLETE IMPLIES NOT afirst(st))],
        st = st WITH [afirst := i=LAST],
        st = st WITH [afile  := i/=LAST]
      IN st,
    IND_ERR:
      ...,
    CONF(c):
      ...
  ENDCASES
```

Note that in the `REQ` case, `afile` is set to `TRUE` to indicate that the file now is non-empty. Before, it was set to denote the new file, and the `ahead` was set to point to the first element in this file (`ahead := 1`). In the `IND` case, substantial changes have been made. This case did before look as follows:

```
IND(d,i):
  LET
    st = st WITH [SAFE :=
      abusy(st) AND NOT aerror(st) AND
      ahead(st) /= nil AND
      d = afile(st)(ahead(st)) AND
      i = IF ahead(st) = last THEN
            LAST
          ELSE IF afirst(st) THEN
                 FIRST
               ELSE
                 INCOMPLETE
               ENDIF
          ENDIF],
    st = st WITH [afirst := ahead(st) = last],
    st = st WITH [ahead  := incr_head(ahead(st))]
  IN st,
```

So, for example the concrete test `ahead(st) /= nil` (the file is not empty) has become `afile(st)` in the abstract version. Also, the test `d = afile(st)(ahead(st))` in the concrete has become `data_is_safe`. We shall see how the mapping maps the right information into that variable. Note also how the abstraction allows a `LAST` indication at any time. Observe also how the concrete update of the variable `ahead` has been replaced with an updating of `afile`. Also the updating of `afirst` has been changed.

Transition Rules:

We will only present the change of two rules, namely `Rule_write_K` and `Rule_stimer1`. The first rule is changed as follows:

```
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) := file(st)=ONE],
      st = st WITH [(K)(toggle) := stoggle(st)],
      st = st WITH [spc := WA],
      st = st WITH [rn := TRUE],
      st = st WITH [stimer1_on := TRUE]
    IN st
  ELSE
    st
  ENDIF
```

Here the update of `K(last)` has been changed. Before it was `K(last) := (head(st)=last)`. The update of `rn` has changed, before it was `rn := rn + 1` (in principle). Finally, before there was an update of `K(data)`, which has now disappeared. `Rule_stimer1` has been implemented as two rules in the abstract protocol:

```
Rule_stimer1_QUIT(st:State):State =
  IF stimer1_on(st) AND stimer1_enabled(st) THEN
    LET
      st = st WITH [spc := SC],
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

Rule_stimer1_RETRY(st:State):State =
  IF stimer1_on(st) AND stimer1_enabled(st) THEN
    LET
      st = st WITH [spc := SF],
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF
```

Recall that the behavior of the concrete rule depends on the condition whether the maximum number of retransmissions has been reached, `rn(st)=max`: if it is true, then `spc := SC` (quit and send confirmation) else `spc := SF` (retry and send again). The abstract protocol is just supposed to be able to follow any such transition taken by the concrete one, so a non-deterministic choice between the two possibilities is sufficient, hence two rules. Recall that our choice has been to let each rule be deterministic, only allowing non-determinism in the selection of which rule to be executed at a given moment.

Besides these rules also a corresponding `transition` relation is defined as the standard disjunction of all the rules, and also an `initial` predicate is defined on abstract states.

5.4 The Abstraction Mapping

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_file(h:C.Position):A.File =
    IF h=nil THEN
      NONE
    ELSIF h=last THEN
      ONE
    ELSE
      MANY
    ENDIF

  abs(s:C.State):A.State =
    (#  ...
        K              := (# first  := first(K(s)),
                             last   := last(K(s)),
                             toggle := toggle(K(s)) #),
        req            := IF C.last = 1 THEN ONE ELSE MANY ENDIF,
        file           := abs_file(head(s)),
        rn             := rn(s)/=0,
        msg            := (# first  := first(msg(s)),
                             last   := last(msg(s)),
                             toggle := toggle(msg(s)) #),
        data_is_safe   := rpc(s)=SI IMPLIES
                            data(msg(s))=afile(s)(ahead(s)),
        afile          := ahead(s)/=nil,
        ...
    #)

  ...
END abstraction
```

The two protocols, abstract and concrete, are imported via so called theory instantiations: each instantiation is given a name such that we can distinguish the components of the two theories. The function `abs` is the abstraction function from the concrete state to the abstract state, and is here given in its entirety except for fields that are unchanged; the latter being mapped as `name := name(s)`.

The `file` component containing the current file being processed by the sender depends in the abstract protocol on how many messages there are left in the concrete. The `rn` component is a boolean being true if at least one retransmission has been performed. The `afile` component is also a boolean being true if messages still remain in the `automaton` file.

The `K` component is a reduced record where the data component has been ignored. The `msg` component is the receiver's copy of `K` after reading it, and hence is reduced the same way. The `req` component represents the requests being obtained from outside: in case the concrete protocol files can only contain one message (`last = 1`, the `req` is `ONE`, otherwise `MANY`.

Finally, the `data_is_safe` component represents the condition used in the `automaton` when a message is indicated `IND` to the environment as transmitted: the `rpc(s)` is the program counter of the receiver, and its value `SI` stands for "send indication" – the data indicated to the environment must be the current one in `afile`.

5.5 The Abstraction Proof

We shall now apply theorem 1, recalling that this theorem 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. Appendices C.2, C.3, C.4 and C.5 contain listings of the proof tactics, the proof scripts, status of lemmas, and dependencies between lemmas. The theorem we want to prove is the correctness of the concrete protocol:

```
safety : THEOREM
  AG(initial,transition)(safe)
```

Theorem 1 tells us that it is sufficient to prove this theorem for the abstract protocol (via model checking) plus the following 3 propositions corresponding to items 1–3 in the theorem:

```
init_preserved : PROPOSITION
  A.initial(abs(C.startstate))

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))

property_preserved : PROPOSITION
  FORALL (s:C.State): SAFE(abs(s)) IMPLIES SAFE(s)
```

Note in particular how in proposition `transition_preserves` it is stated that states are reachable: they satisfy the predicate `I`, where `I` is supposed to be an invariant of the concrete protocol. It turns out, that in order to do the proof of this proposition, we need to assume some properties about the infinite-state protocol, and these properties are represented by `I`. So we still have to prove *some* invariants about the concrete protocol, though fewer than in case where no model checking was applied. This is perhaps not surprising if one considers the abstract finite-state protocol basically to deal with control only: the infinite data behavior is left for us to deal with using the deductive capability of PVS, and this necessarily has to take place in the context of the concrete protocol.

It turns out that the invariant `I` needed for the abstraction proof is contained in the invariant we have already proven. Hence, practically speaking, the proof of `transition_preserves` was created by importing the original proof, in which `I` was proven to be an invariant of the concrete protocol, and then defining the needed invariants as consequences of `I`. A special purpose theory was created for this, which is then used in the abstraction proof:

```
protocol_lemmas : THEORY
BEGIN
  IMPORTING protocol_safe

  st : VAR State

  a_spc_1(st):bool =
    spc(st)=SC IMPLIES rpc(st)=WF

  a_safe_4(st):bool =
    (spc(st)=SC AND head(st)=last AND rn(st)/=0)
      IMPLIES
    (ahead(st)=last OR ahead(st)=nil)

  a_safe_7(st):bool =
    rpc(st)=SI IMPLIES ahead(st)/=nil

  a_safe_8(st): bool =
    rpc(st) = SI IMPLIES data(msg(st)) = afile(st)(ahead(st))

  a_safe_9(st):bool =
    rpc(st) = SI IMPLIES
      ((last(msg(st))=(ahead(st)=last)) AND (first(msg(st))=afirst(st)))

  I_spc_1  : LEMMA I IMPLIES a_spc_1
  I_safe_4 : LEMMA I IMPLIES a_safe_4
  I_safe_7 : LEMMA I IMPLIES a_safe_7
  I_safe_8 : LEMMA I IMPLIES a_safe_8
  I_safe_9 : LEMMA I IMPLIES a_safe_9
END protocol_lemmas
```

The proof of the four propositions is now carried out as follows. Propositions `init_preserved` and `property_preserved` are trivial to prove: `(grind)` does it automatically. Proposition `transition_preserves` is the complicated one to prove. First we define two auxiliary functions; each carrying out the subproof needed for a single pair of transition rules: a concrete one `C` and an abstract one `A`.

```
preserves(C:[C.State->C.State],A:[A.State->A.State]):bool =
  FORALL (s1,s2:C.State):
    (I(s1) AND I(s2) AND s2 = C(s1))
      IMPLIES
    abs(s2) = A(abs(s1))

c_preserves(p:pred[C.State])
          (C:[C.State->C.State],A:[A.State->A.State]):bool =
  FORALL (s1,s2:C.State):
    (I(s1) AND I(s2) AND p(s1) AND s2 = C(s1))
      IMPLIES
    abs(s2) = A(abs(s1))
```

The function `preserves` states that: for all reachable concrete states `s1` and `s2`, if `s2` can be reached from `s1` via the rule `C`, then the rule `A` connects the corresponding abstracted states. The rule `c_preserves` (conditional version) takes an extra parameter: a condition `p` that is assumed to hold in the beginning state `s1`. It is shown below how this is used; the following lemmas are examples of lemmas needed to prove `transition_preserves`:

```
write_K_preserves : LEMMA
  preserves(C.Rule_write_K,A.Rule_write_K)

stimer1_preserves1 : LEMMA
  c_preserves(LAMBDA (s:C.State):rn(s)=max)
           (C.Rule_stimer1,A.Rule_stimer1_QUIT)

stimer1_preserves2 : LEMMA
  c_preserves(LAMBDA (s:C.State):rn(s)/=max)
           (C.Rule_stimer1,A.Rule_stimer1_RETRY)
```

The first lemma states that if `C.Rule_write_K` connects two concrete reachable states, then `A.Rule_write_K` connects the corresponding abstracted states. For the concrete rule `C.Rule_stimer1` there are two abstract rules that together simulate it at the abstract level: the rule `A.Rule_stimer1_QUIT` simulates it in the case where the maximum number of retransmissions has been reached (`rn(s)=max`), while the rule `A.Rule_stimer1_RETRY` simulates it in the opposite case – a message is retransmitted.

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 about an hour and a half 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 58 invariants invented in the original invariant proof in chapter 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 proof+model checking instead of 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 (`invariant`) 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. The next chapter presents the results.

Chapter 6

Model Checking the Abstraction in Other Systems

Initial experiences with applying the model checker embedded in PVS to the theorem `invariant` were not satisfactory. In fact, we had a version of the `AG` operator together with an unfortunate formulation of the transition relation that prevented the model checker from returning within reasonable time. This fact combined with the fact that the original protocol was verified in Murϕ within minutes (chapter 3) lead us to try out the example in two other model checkers to see if they were doing better – one based on simple state exploration: Murϕ and one based on BDD's: SMV. That is, in each of these systems we formulated the finite-state abstraction of the protocol, and then verified it's correctness. This chapter reports on our experiences. The full formal texts associated with this chapter are contained in appendix D.

6.1 Murphi

We have already explained how Murϕ works. In principle we hand-translated the abstract PVS protocol into Murϕ, in practice, we modified the Murϕ-program already existing (see appendix A.1 for the already existing Murϕ-program). The full formal new Murϕ-program is contained in appendix D.1. Some of this new program is shown here:

```
Type
  File : Enum{NONE,ONE,MANY};
  Msg  : Record first, last, toggle : boolean End;
Var
  K : Msg;
  afile  : boolean;
  data_is_safe : boolean;
  req : File;
  file : File;
  rn : boolean;
...

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

Murϕ explored 22.432 states, with 74.626 rules fired in 28,51 seconds. This is very fast, and compared with the initial mu-calculus results it revealed an interesting issue: the simple techniques behind Murϕ were superior to a BDD-based model checker. Murϕ has the characteristics that it uses an explicit (not symbolic) representation of the state space and only explores the reachable states, while the mu-calculus checker explores all states, if not guided in a way that we later discovered.

6.2 SMV

The SMV system [19] is a tool for checking finite-state systems against specifications in the temporal logic CTL. It is based on (O)BDD's. We regard it as the "800 pound gorilla" we can't get around (as Shankar puts it) within model checking, hence it's importance as a test-ground for our example is obvious.

The system is designed to allow the description of asynchronous systems as well as synchronous systems. Our protocol is asynchronous in the sense that in each step exactly one action with a satisfied pre-condition is chosen and then executed (interleaved), and only the variables modified by that action are modified. Communication is via shared variables: in one step a process writes to a variable, and in a later step another process reads this variable. In a synchronous system on the other hand, all variables are modified in each execution step: the program is conceptually one big parallel assignment to all variables, and execution is the repeated execution of this parallel assignment. The synchronous model is suited for modeling hardware circuits where execution often is controlled by a single clock. One gets the feeling that the SMV system was originally developed with synchronous circuits as target, hence we had to give some thought as to how to model our asynchronous system in SMV, especially the modeling of pre-conditions. However, once we found the "formula", the representation was straight forward, and not very different from the corresponding Murϕ-program.

Only finite data types are allowed: booleans, scalars (enumerated), records and fixed arrays. The CTL logic allows one to specify properties such as safety, liveness, fairness and freedom from deadlock. SMV allows a completely general way of defining a transition relation (the "TRANS" construct for the reader who knows SMV), as the mu-calculus, but it also has restricted constructs, the use of which prevents certain inconsistencies to be introduced. As we shall see we use the general approach to define our pre-conditions and the restricted approach to define the effects on the state.

The full formal SMV program is contained in appendix D.2. Let us illustrate part of this program. A SMV program consists of a set of modules, one of which is the main module (to be shown later). A module is an encapsulated collection of declarations. First, we introduce (part of) the `state` module, which introduces all the global variables of the program. Modules are also used to represent records as is the case with `Msg`:

```
MODULE Msg
VAR
  first : boolean;
  last : boolean;
  toggle : boolean;
```

```
MODULE state
VAR
  K : Msg;
  req : {NONE,ONE,MANY};
  file : {NONE,ONE,MANY};
  rn : boolean;
  afile  : boolean;
  data_is_safe : boolean;
  SAFE : boolean;
  ...
INIT
    K.first = 0
  & K.last = 0
  & K.toggle = 0
  & req in {ONE,MANY}
  & file = NONE
  & rn = 0
  & data_is_safe = 1
  & afile = 0
  & SAFE = 1
  ...
```

The `VAR` section introduces all the variables, while the `INIT` section defines their initial values; booleans in SMV are represented by the type `{0,1}`. Note that we have introduced the variable `SAFE`, just as we have done in the PVS specifications. This variable is manipulated by the automaton, and the correctness criteria will be that this variable is always true. This variable was not necessary in the Murϕ-program due to the `assert` construct available there.

The `state` type can be conceived as a *passive* module in the sense that it only introduces variables and no description of dynamic behavior. The rest of the modules describe this dynamic behavior, one module for each rule in Murϕ. As an example we show the module corresponding to the `write_K` rule in the previous section 6.1:

```
MODULE write_K(s)
TRANS
  running ->
```

```
    s.spc = SF &
    !s.K_full
  ASSIGN
    next(s.K_full) := 1;
    next(s.K.first) := s.sfirst;
    next(s.K.last) := (s.file=ONE);
    next(s.K.toggle) := s.stoggle;
    next(s.spc) := WA;
    next(s.rn) := 1;
    next(s.stimer1_on) := 1;
```

The state is passed as parameter to the module, we see in a moment how it is instantiated. All references to this state are by dot-notation (ex: `s.spc`). The module is divided into two sections. The `ASSIGN` section describes the effect of the rule on the state: it is a parallel assignment statement, where `next(id)` refers to the value of the variable `id` in the next state. The parallel assignment can then be regarded as a set of equations that must hold between the current state and the next state. To illustrate this further, the next-operator can even occur on the right hand sides of these assignments (equations). This latter trick we have in fact used to model the Murϕ-program as closely as possible: in Murϕ the effect of a rule is a *sequentially* evaluated sequence of assignments, one assignment is evaluated at a time, hence the value (in the next state) of an already assigned variable may be referred to on the right hand side of a later assignment.

The `ASSIGN` section is an example of the restricted language of SMV in which one can write a predicate on the current-state/next-state in a safe way. In general, after the `TRANS` keyword, any boolean expression may be written that constrains such pairs of current-state/next-state: any current/next state pair is in the transition relation only if the value of this expression is true. In our case the `TRANS` expression describes the pre-condition of the rule: with each module follows an implicitly declared variable `running`, which will be true whenever the module (rule) is executed. The expression then says that whenever the rule is executed, the pre-condition `s.spc = SF & !s.K_full` holds; `->` is the 'implies' operator. Hence this expression only refers to the current state of each current-state/next-state pair, which effectively means that it must be true in all states.

We mentioned that the `SAFE` variable is modified by the automaton. Since procedures are not available in SMV, we must code the effects of the automaton directly where they are needed, as illustrated by the following rule that reads a new request from the environment:

```
MODULE read_req(s)
TRANS
  running ->
  s.spc = WR &
  s.req_full
ASSIGN
  next(s.req_full) := 0;
  next(s.file) := s.req;
  next(s.spc) := SF;
  -- automaton
  next(s.SAFE) := !s.abusy;
  next(s.abusy) := 1;
  next(s.afile) := 1;
```

The main module declares the state, instantiates all the rules, and states the correctness criteria:

```
MODULE main
VAR
  s   : state;
  write_req       : process write_req(s);
  read_conf       : process read_conf(s);
  read_ind        : process read_ind(s);
  read_ind_error  : process read_ind_error(s);
  lose_msg        : process lose_msg(s);
  lose_ack        : process lose_ack(s);
  read_req        : process read_req(s);
  write_K         : process write_K(s);
  read_L_MANY     : process read_L_MANY(s);
  read_L_ONE      : process read_L_ONE(s);
  write_conf      : process write_conf(s);
  stimer1_QUIT    : process stimer1_QUIT(s);
  stimer1_RETRY   : process stimer1_RETRY(s);
  stimer2         : process stimer2(s);
  read_K          : process read_K(s);
  write_ind       : process write_ind(s);
```

```
    write_L         : process write_L(s);
    write_ind_error : process write_ind_error(s);
    rtimer          : process rtimer(s);
  SPEC
    AG s.SAFE
```

The state `s` is declared as an instance of the `state` module, and is passed as parameter to all the rule modules. For each rule module, a variable (with the same name) is declared to be a process instance of it using the keyword `process`. The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. By default, all of the assignment statements in an SMV program are executed in parallel and simultaneously, but processes on the other hand are executed interleaved.

It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Each instance of a process has associated the variable `running`, the value of which is `1` if the process instance is currently selected for execution. Note how the `TRANS` statements restricts the choice of processes by relating pre-conditions to the values of the `running` variables.

The correctness criteria is defined as a CTL formula following the keyword `SPEC`. It states that for all paths (`A`), and for all states along such a path (`G`), the variable `s.SAFE` must be true.

The modeling of the protocol in SMV was straight forward once the schema for how to do was layed out. One slightly annoying thing in SMV is though the lack of a conditional statement construct, there is only a conditional expression construct, expressions being the objects occurring on the right hand sides of assignments. Also procedures and type declarations are missing in SMV.

When we apply the SMV model checker to the example, it uses 9308 seconds (2.5 hours) to verify it, allocating 1.210.892 BDD nodes.

It is, however, possible to activate SMV using an option '-f', with the result that only reachable states are examined. This gives an impressive improvement in efficiency: 24 seconds only, and 29.132 BDD nodes allocated. With respect to time this is an improvement with a factor nearly 400.

6.3 Conclusion

The main source of efficiency in Murϕ with respect to this example is that it only explores the reachable states. Also the experiment with SMV demonstrated the importance of focusing on reachable states, using option '-f'. 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. The mu-calculus [14] has recently been extended with a reachability option (like option '-f' in SMV), which is expected to be incorporated into PVS. This should further improve the PVS model checker.

Chapter 7

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 [11] 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 [11] 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 less than four weeks.

We find our protocol modeling and verification techniques to be simple and effective. We just use higher order logic without a special programming language syntax. Of course one can define special programming language syntax even within PVS (as PVS functions), but we have felt no need at all to do this. Also, an alternative to the shared variable paradigm is the message passing paradigm found in CCS [21], CSP [12] or IO-automata [18]. We have felt no loss in generality working with shared variables. On the contrary, it feels straight forward.

Our use of abstraction yields a better understanding of the protocol since it extracts out the relevant control skeleton. A deeper study will reveal systematic abstraction techniques.

Acknowledgments. 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 μ-calculus model checker (built by Geert Janssen of Eindhoven University of Technology) into PVS. SeungJoon Park (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.

Appendix A

State Exploration in Murϕ

This appendix associates to chapter 3. It contains the Murϕ program.

A.1 The Murϕ Program

```
     -- Shared Declarations --

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

Var
  K : Msg;
  K_full : boolean;
  L : boolean;


     -- Sender Declarations --
Const
  max  : 2;
  last : 3; -- verified for the values: 1, 2, 3
```

```
  nil  : last+1;

Type
  Spc  : Enum{WR, SF, WA, SC, WT2};
  Conf : Enum{OK, NOT_OK, DONT_KNOW};
  File : Array [1..last] Of Data;


Var
  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;


     -- Receiver Declarations --

Type
  Rpc  : Enum{WF, SI, SA, RTS, NOK};
  IndT : Enum{FIRST, INCOMPLETE, LAST};

Var
  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;


     -- Abstract automaton --

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

Procedure verify_REQ(f:File);
Begin
  assert !abusy "REQ";
  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)) "IND";
  afirst := (ahead=last);
  ahead := ahead + 1;
End;

Procedure verify_IND_ERR();
Begin
  assert aerror "IND_ERR";
  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) "CONF";
  abusy := false;
  aerror := !afirst;
  ahead := nil;
End;


     -- Initialisation --

Startstate
  Begin
    K.first := false;
    K.last := false;
    K.toggle := false;
    K.data := false;
    K_full := false;
    L := false;
    For i := 1 To last Do req[i] := false End;
    req_full := false;
    conf := OK;
    conf_full := false;
    spc := WR;
    sfirst := true;
    stoggle := false;
    For i := 1 To last Do file[i] := false End;
    head := nil;
    rn := 0;
    stimer1_on := false;
    stimer1_enabled := false;
    stimer2_on := false;
    stimer2_enabled := false;
    ind_data := false;
    ind_indication := FIRST;
    ind_full := false;
```

```
    ind_error := false;
    rpc := WF;
    rfirst := true;
    rtoggle := false;
    ctoggle := false;
    msg.first := false;
    msg.last := false;
    msg.toggle := false;
    msg.data := false;
    rtimer_on := true;
    rtimer_enabled := false;
    abusy := false;
    For i := 1 To last Do afile[i] := false End;
    ahead := nil;
    afirst := true;
    aerror := false;
  End;


     -- Environment Actions --

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;


     -- Sender Program --

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;


     -- Receiver Program --
```

```
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;
```

Appendix B

Theorem Proving in PVS

This appendix associates to chapter 4. It contains listings of the specifications, the tactics, the proof scripts, the lemma status, the lemma dependencies, and finally the invariants as formulated in Murϕ.

B.1 The Infinite State PVS Specification

```
protocol : THEORY

BEGIN

  Data : NONEMPTY_TYPE

  last : posnat
  max  : posnat

  Counter  : TYPE = {i: int | 0 <= i AND i <= max}
  Position : TYPE = {i: int | 1 <= i AND i <= last+1}

  File : TYPE = ARRAY[Position -> Data]

  Msg : TYPE = [# first: bool, last: bool, toggle: bool, data: Data #]

  Spc : TYPE = {WR, SF, WA, SC, WT2}
```

```
Rpc : TYPE = {WF, SI, SA, RTS, NOK}

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

nil : posnat = last + 1

incr_rn(r:Counter):Counter =
  IF r < max THEN r+1 ELSE r ENDIF

incr_head(h:Position):Position =
  IF h < nil THEN h+1 ELSE h ENDIF


State : TYPE =
  [# aerror : bool,
     afirst : bool,
     ahead : Position,
     afile : File,
     abusy : bool,
     rtimer_enabled : bool,
     rtimer_on : bool,
     msg : Msg,
     ctoggle : bool,
     rtoggle : bool,
     rfirst : bool,
     rpc : Rpc,
     ind_error : bool,
     ind_full : bool,
     ind_indication : IndT,
     ind_data : Data,
     stimer2_enabled : bool,
     stimer2_on : bool,
     stimer1_enabled : bool,
     stimer1_on : bool,
     rn : Counter,
     head : Position,
     file : File,
     stoggle : bool,
```

```
    sfirst : bool,
    spc : Spc,
    conf_full : bool,
    conf : Conf,
    req_full : bool,
    req : File,
    L : bool,
    K_full : bool,
    K : Msg,
    SAFE : bool #]

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

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):
      LET
        st = st WITH [SAFE :=
          abusy(st) AND NOT aerror(st) AND
          ahead(st) /= nil AND
          d = afile(st)(ahead(st)) AND
          i = IF ahead(st) = last THEN
                LAST
              ELSE IF afirst(st) THEN
                  FIRST
                ELSE
```

```
                    INCOMPLETE
                  ENDIF
              ENDIF],
         st = st WITH [afirst := ahead(st) = last],
         st = st WITH [ahead  := incr_head(ahead(st))]
       IN st,
     IND_ERR:
       LET
         st = st WITH [SAFE := aerror(st)],
         st = st WITH [afirst := TRUE],
         st = st WITH [aerror := FALSE]
       IN st,
     CONF(c):
       LET
         st = st WITH [SAFE :=
           abusy(st) AND NOT aerror(st) AND
           (c = OK IMPLIES ahead(st) = nil) AND
           (c = DONT_KNOW IMPLIES
                  (ahead(st) = nil OR ahead(st) = last)) AND
           (c = NOT_OK IMPLIES ahead(st) /= nil)],
         st = st WITH [abusy := FALSE],
         st = st WITH [aerror := NOT afirst(st)],
         st = st WITH [ahead  := nil]
       IN st
   ENDCASES

null_data : Data

startstate : State =
  (# SAFE           := TRUE,
     K              := (# first  := FALSE,
                          last   := FALSE,
                          toggle := FALSE,
                          data   := null_data #),
     K_full         := FALSE,
     L              := FALSE,
     req            := LAMBDA (i:Position):null_data,
     req_full       := FALSE,
     conf           := OK,
```

```
    conf_full       := FALSE,
    spc             := WR,
    sfirst          := TRUE,
    stoggle         := FALSE,
    file            := LAMBDA (i:Position):null_data,
    head            := nil,
    rn              := 0,
    stimer1_on      := FALSE,
    stimer1_enabled := FALSE,
    stimer2_on      := FALSE,
    stimer2_enabled := FALSE,
    ind_data        := null_data,
    ind_indication  := FIRST,
    ind_full        := FALSE,
    ind_error       := FALSE,
    rpc             := WF,
    rfirst          := TRUE,
    rtoggle         := FALSE,
    ctoggle         := FALSE,
    msg             := (# first  := FALSE,
                          last   := FALSE,
                          toggle := FALSE,
                          data   := null_data #),
    rtimer_on       := TRUE,
    rtimer_enabled  := FALSE,
    abusy           := FALSE,
    afile           := LAMBDA (i:Position):null_data,
    ahead           := nil,
    afirst          := TRUE,
    aerror          := FALSE
  #)

Rule_write_req(st:State,f:File):State =
  IF NOT req_full(st) THEN
    LET
      st = st WITH [req_full := TRUE],
      st = st WITH [req := f]
    IN st
  ELSE
```

```
    st
  ENDIF

Rule_read_conf(st:State):State =
  IF conf_full(st) THEN
    LET st = st WITH [conf_full := FALSE] IN st
  ELSE
    st
  ENDIF

Rule_read_ind(st:State):State =
  IF ind_full(st) THEN
    LET st = st WITH [ind_full := FALSE] IN st
  ELSE
    st
  ENDIF

Rule_read_ind_error(st:State):State =
  IF ind_error(st) THEN
    LET st = st WITH [ind_error := FALSE] IN st
  ELSE
    st
  ENDIF

Rule_lose_msg(st: State):State =
  IF K_full(st) THEN
    LET
      st = st WITH [K_full := FALSE],
      st = st WITH [stimer1_enabled := TRUE]
    IN st
  ELSE
    st
  ENDIF

Rule_lose_ack(st: State):State =
  IF L(st) THEN
    LET
      st = st WITH [L := FALSE],
      st = st WITH [stimer1_enabled := TRUE]
```

```
    IN st
  ELSE
    st
  ENDIF

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

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

Rule_read_L(st:State):State =
  IF spc(st) = WA AND L(st) THEN
    LET
      st = st WITH [L := FALSE],
      st = IF head(st) = last THEN
```

```
              LET st = st WITH [spc := SC] IN st
            ELSE
              LET st = st WITH [spc := SF] IN st
            ENDIF,
        st = st WITH [sfirst := head(st) = last],
        st = IF NOT (head(st) = last) THEN
              LET st = st WITH [rn := 0] IN st
            ELSE
              st
            ENDIF,
        st = st WITH [head := incr_head(head(st))],
        st = st WITH [stoggle := NOT (stoggle(st))],
        st = st WITH [stimer1_on := FALSE],
        st = st WITH [stimer1_enabled := FALSE]
      IN st
    ELSE
      st
    ENDIF

  Rule_write_conf(st:State):State =
    IF spc(st) = SC AND NOT conf_full(st) THEN
      LET
        st = st WITH [conf_full := TRUE],
        st = IF head(st) = nil THEN
              LET st = st WITH [conf := OK] IN st
            ELSE
              LET
                st = IF head(st) = last AND rn(st) /= 0 THEN
                      LET st = st WITH [conf := DONT_KNOW] IN st
                    ELSE
                      LET st = st WITH [conf := NOT_OK] IN st
                    ENDIF
              IN st
            ENDIF,
        st = IF head(st) = nil THEN
              LET st = st WITH [spc := WR] IN st
            ELSE
              LET
                st = st WITH [spc := WT2],
```

```
              st = st WITH [sfirst := TRUE],
              st = st WITH [stoggle := NOT stoggle(st)],
              st = st WITH [stimer2_on := TRUE],
              st = st WITH [rtimer_enabled := TRUE]
            IN st
          ENDIF,
      st = st WITH [head := nil],
      st = st WITH [rn := 0],
      st = automaton(st, CONF(conf(st)))
    IN st
  ELSE
    st
  ENDIF

Rule_stimer1(st:State):State =
  IF stimer1_on(st) AND stimer1_enabled(st) THEN
    LET
      st = IF rn(st) = max THEN
            LET st = st WITH [spc := SC] IN st
          ELSE
            LET st = st WITH [spc := SF] IN st
          ENDIF,
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

Rule_stimer2(st:State):State =
  IF stimer2_on(st) AND stimer2_enabled(st) THEN
    LET
      st = st WITH [spc := WR],
      st = st WITH [stimer2_on := FALSE],
      st = st WITH [stimer2_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF
```

```
Rule_read_K(st:State):State =
  IF rpc(st) = WF AND K_full(st) THEN
    LET
      st = st WITH [K_full := FALSE],
      st = st WITH [(msg)(first) := first(K(st))],
      st = st WITH [(msg)(last) := last(K(st))],
      st = st WITH [(msg)(toggle) := toggle(K(st))],
      st = st WITH [(msg)(data) := data(K(st))],
      st = IF NOT ctoggle(st) OR toggle(msg(st)) = rtoggle(st) THEN
             LET
               st = st WITH [rpc := SI],
               st = st WITH [rtimer_on := FALSE],
               st = st WITH [rtimer_enabled := FALSE]
             IN st
           ELSE
             LET st = st WITH [rpc := RTS] IN st
           ENDIF
    IN st
  ELSE
    st
  ENDIF

Rule_write_ind(st: State):State =
  IF rpc(st) = SI AND NOT ind_full(st) THEN
    LET
      st = st WITH [ind_full := TRUE],
      st = st WITH [ind_data := data(msg(st))],
      st = IF last(msg(st)) THEN
             LET st = st WITH [ind_indication := LAST] IN st
           ELSE
             LET
               st = IF first(msg(st)) THEN
                      LET st = st WITH [ind_indication := FIRST] IN st
                    ELSE
                      LET st = st WITH [ind_indication := INCOMPLETE] IN st
                    ENDIF
             IN st
           ENDIF,
```

```
      st = st WITH [rpc := SA],
      st = st WITH [rfirst := last(msg(st))],
      st = st WITH [ctoggle := TRUE],
      st = st WITH [rtoggle := NOT toggle(msg(st))],
      st = automaton(st, IND(ind_data(st),ind_indication(st)))
    IN st
  ELSE
    st
  ENDIF

Rule_write_L(st:State):State =
  IF (rpc(st) = SA OR rpc(st) = RTS) AND NOT L(st) THEN
    LET
      st = st WITH [L := TRUE],
      st = IF rpc(st) = SA THEN
             LET st = st WITH [rtimer_on := TRUE] IN st
           ELSE
             st
           ENDIF,
      st = st WITH [rpc := WF]
    IN st
  ELSE
    st
  ENDIF

Rule_write_ind_error(st:State):State =
  IF rpc(st) = NOK AND NOT ind_error(st) THEN
    LET
      st = st WITH [ind_error := TRUE],
      st = st WITH [rpc := WF],
      st = st WITH [rfirst := TRUE],
      st = st WITH [rtimer_on := TRUE],
      st = st WITH [stimer2_enabled := TRUE],
      st = automaton(st,IND_ERR)
    IN st
  ELSE
    st
  ENDIF
```

```
Rule_rtimer(st:State):State =
  IF rtimer_on(st) AND rtimer_enabled(st) THEN
    LET
      st = st WITH [ctoggle := FALSE],
      st = IF NOT rfirst(st) THEN
             LET
               st = st WITH [rpc := NOK],
               st = st WITH [rtimer_on := FALSE]
             IN st
           ELSE
             LET st = st WITH [stimer2_enabled := TRUE] IN st
           ENDIF,
      st = st WITH [rtimer_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

transition(st1,st2:State):bool =
  (EXISTS (f: File):
     st2 = Rule_write_req(st1,f))
  OR st2 = Rule_read_conf(st1)
  OR st2 = Rule_read_ind(st1)
  OR st2 = Rule_read_ind_error(st1)
  OR st2 = Rule_lose_msg(st1)
  OR st2 = Rule_lose_ack(st1)
  OR st2 = Rule_read_req(st1)
  OR st2 = Rule_write_K(st1)
  OR st2 = Rule_read_L(st1)
  OR st2 = Rule_write_conf(st1)
  OR st2 = Rule_stimer1(st1)
  OR st2 = Rule_stimer2(st1)
  OR st2 = Rule_read_K(st1)
  OR st2 = Rule_write_ind(st1)
  OR st2 = Rule_write_L(st1)
  OR st2 = Rule_write_ind_error(st1)
  OR st2 = Rule_rtimer(st1)
```

```
  IMPORTING sequences[State]

  Behavior: TYPE = sequence

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

END protocol



protocol_safe : THEORY

BEGIN

  IMPORTING protocol

  p,p1,p2 : VAR pred[State]

  invariant(p):bool =
    FORALL (aa:Behavior):
      program(aa) IMPLIES
        FORALL (n:nat): p(aa(n));

  preserved(p1)(p2): bool =
    p2(startstate)
      AND FORALL (st1, st2: State):
        (p1(st1) AND p2(st1) AND transition(st1, st2)) IMPLIES p2(st2);

  IMPLIES(p1,p2):bool =
    FORALL (st:State):p1(st) IMPLIES p2(st);

  &(p1,p2):pred[State] =
    LAMBDA (st:State): p1(st) AND p2(st)


  st : VAR State
```

```
%-- Invariant Predicates

ref_abusy(st):bool =
  abusy(st) = (spc(st) = SF OR spc(st) = WA OR spc(st) = SC)

ref_afirst(st):bool =
  afirst(st) = rfirst(st)

ref_aerror(st):bool =
  aerror(st) =
    ((rpc(st) = NOK) OR (spc(st) = WT2 AND rtimer_on(st) AND NOT rfirst(st)))

ref_afile(st):bool =
  afile(st) = file(st)

ref_ahead(st):bool =
  ahead(st) =
    IF (spc(st) = WT2 OR (ctoggle(st) IMPLIES stoggle(st)=rtoggle(st))) THEN
      head(st)
    ELSE
      head(st)+1
    ENDIF

spc_1(st):bool =
  (spc(st) = WR OR spc(st) = SF OR spc(st) = SC)
    IMPLIES
  rpc(st) = WF

spc_2(st):bool =
  (spc(st)=SF AND rn(st)=0) IMPLIES
    ctoggle(st) IMPLIES stoggle(st)=rtoggle(st)

spc_3(st):bool =
  spc(st)=WA IMPLIES rn(st)/=0

spc_4(st):bool =
  spc(st)=WT2 AND NOT rtimer_enabled(st) IMPLIES NOT ctoggle(st)
```

```
spc_5(st):bool =
  spc(st)=SC AND head(st)=nil IMPLIES
    (ctoggle(st) AND stoggle(st)=rtoggle(st))

spc_6(st):bool =
  spc(st)=SC AND head(st)=nil IMPLIES rfirst(st)

spc_7(st):bool =
  spc(st)=WR IMPLIES (ctoggle(st) IMPLIES stoggle(st)=rtoggle(st))

spc_8(st):bool =
  spc(st)=WT2 IMPLIES rn(st)=0

spc_9(st):bool =
  (spc(st)=SC AND rn(st)=0) IMPLIES
    ctoggle(st) IMPLIES stoggle(st)=rtoggle(st)

spc_10(st):bool =
  spc(st)=SF IMPLIES
    (ctoggle(st) IMPLIES stoggle(st)=rtoggle(st)) IMPLIES
      sfirst(st)=rfirst(st)

spc_11(st):bool =
  spc(st)=WR IMPLIES
    (ctoggle(st) IMPLIES stoggle(st)=rtoggle(st)) IMPLIES
      sfirst(st)=rfirst(st)

spc_12(st):bool =
spc(st)=WA IMPLIES
  (ctoggle(st) IMPLIES stoggle(st)=rtoggle(st)) IMPLIES
    sfirst(st)=rfirst(st)

spc_13(st):bool =
  spc(st)=SC IMPLIES
    (ctoggle(st) IMPLIES stoggle(st)=rtoggle(st)) IMPLIES
      sfirst(st)=rfirst(st)

spc_14(st):bool =
```

```
  spc(st)=WT2 IMPLIES sfirst(st)

spc_15(st):bool =
  spc(st)=WA IMPLIES head(st)/=nil

spc_16(st):bool =
  spc(st)=SF IMPLIES head(st)/=nil

rpc_1(st):bool =
  (rpc(st) = SI OR rpc(st) = SA OR rpc(st) = RTS)
    IMPLIES
  spc(st) = WA

rpc_2(st):bool =
  rpc(st) = NOK
    IMPLIES
  spc(st) = WT2

rpc_3(st):bool =
  rpc(st)=SI IMPLIES last(msg(st))=(head(st)=last)

rpc_4(st):bool =
  rpc(st)=NOK IMPLIES NOT ctoggle(st)

rpc_5(st):bool =
  rpc(st)=SI IMPLIES stoggle(st)=toggle(msg(st))

rpc_6(st):bool =
  (rpc(st)=SA OR rpc(st)=RTS) IMPLIES stoggle(st)=toggle(msg(st))

rpc_7(st):bool =
  (rpc(st)=SA OR rpc(st)=RTS) IMPLIES
    (ctoggle(st) AND toggle(msg(st))/=rtoggle(st))

rpc_8(st):bool =
  (rpc(st)=WF OR rpc(st)=RTS) IMPLIES rtimer_on(st)

rpc_9(st):bool =
  rpc(st)=SI IMPLIES (ctoggle(st) IMPLIES toggle(msg(st))=rtoggle(st))
```

```
rpc_10(st):bool =
  rpc(st)=SI IMPLIES data(msg(st))=(file(st)(head(st)))

rpc_11(st):bool =
  rpc(st)=SI IMPLIES rfirst(st)=first(msg(st))

K_1(st):bool =
  K_full(st)
    IMPLIES
  (spc(st) = WA AND rpc(st) = WF AND (NOT L(st)))

K_2(st):bool =
  K_full(st) IMPLIES last(K(st))=(head(st)=last)

K_3(st):bool =
  K_full(st) IMPLIES toggle(K(st))=stoggle(st)

K_4(st):bool =
  K_full(st) IMPLIES data(K(st))=(file(st)(head(st)))

K_5(st):bool =
  K_full(st) IMPLIES
    (ctoggle(st) IMPLIES toggle(K(st))=rtoggle(st)) IMPLIES
      first(K(st))=rfirst(st)

L_1(st):bool =
  L(st)
    IMPLIES
  (spc(st) = WA AND rpc(st) = WF AND (NOT K_full(st)))

L_2(st):bool =
  L(st) IMPLIES ctoggle(st) AND ((NOT rtoggle(st))=stoggle(st))

stimer1_1(st):bool =
  stimer1_enabled(st)
    IMPLIES
  (spc(st) = WA AND rpc(st) = WF AND (NOT K_full(st)) AND (NOT L(st)))
```

```
stimer2_1(st):bool =
  stimer2_enabled(st)
    IMPLIES
  (spc(st) = WT2 AND rpc(st) = WF AND (NOT K_full(st)) AND (NOT L(st)))

stimer2_2(st):bool =
  stimer2_enabled(st) IMPLIES rfirst(st)

stimer2_3(st):bool =
  stimer2_enabled(st) IMPLIES rfirst(st)

rtimer_1(st):bool =
  rtimer_enabled(st)
    IMPLIES
  (spc(st) = WT2 AND rpc(st) = WF AND (NOT K_full(st)) AND (NOT L(st))
  AND (NOT stimer2_enabled(st)))

rn_1(st):bool =
  rn(st)/=0 IMPLIES spc(st)/=WR

rn_2(st):bool =
  (rn(st)/=0 AND ctoggle(st) AND stoggle(st)/=rtoggle(st)) IMPLIES
    rfirst(st)=(head(st)=last)

head_1(st):bool =
  head(st)=nil IMPLIES (spc(st)=WR OR spc(st)=WT2 OR spc(st)=SC)

safe_1(st):bool =
  spc(st)=WR IMPLIES NOT abusy(st)

safe_2(st):bool =
  spc(st)=SC IMPLIES abusy(st)

safe_3(st):bool =
  spc(st)=SC IMPLIES NOT aerror(st)

safe_4(st):bool =
  spc(st)=SC IMPLIES
    ((head(st)=nil) IMPLIES (ahead(st)=nil))
```

```
      AND
    ((head(st)=last & rn(st)/=0) IMPLIES (ahead(st)=nil OR ahead(st)=last))
      AND
    ((head(st)=last & rn(st)=0) IMPLIES ahead(st) /= nil)
      AND
    ((head(st) < last) IMPLIES ahead(st) /= nil)

safe_5(st):bool =
  rpc(st)=SI IMPLIES abusy(st)

safe_6(st):bool =
  rpc(st)=SI IMPLIES NOT aerror(st)

safe_7(st):bool =
  rpc(st)=SI IMPLIES ahead(st) /= nil

safe_8(st):bool =
  rpc(st)=SI IMPLIES data(msg(st))=(afile(st))(ahead(st))

safe_9(st):bool =
  rpc(st)=SI IMPLIES
    ((last(msg(st))=(ahead(st)=last)) AND (first(msg(st))=afirst(st)))

safe_10(st):bool =
  rpc(st)=NOK IMPLIES aerror(st)

safe(st):bool =
  SAFE(st)


%-- Invariant

Delta : pred[State] =
   ref_abusy
 & ref_afirst
 & ref_aerror
 & ref_afile
 & ref_ahead
 & spc_1
```

```
& spc_2
& spc_3
& spc_4
& spc_5
& spc_6
& spc_7
& spc_8
& spc_9
& spc_10
& spc_11
& spc_12
& spc_13
& spc_14
& spc_15
& spc_16
& rpc_1
& rpc_2
& rpc_3
& rpc_4
& rpc_5
& rpc_6
& rpc_7
& rpc_8
& rpc_9
& rpc_10
& rpc_11
& K_1
& K_2
& K_3
& K_4
& K_5
& L_1
& L_2
& stimer1_1
& stimer2_1
& stimer2_2
& stimer2_3
& rtimer_1
& rn_1
```

```
 & rn_2
 & head_1

I : pred[State] = Delta & safe

pi : [pred[State] -> bool] = preserved(I)


%-- Logical Consequences

p_safe_1      : LEMMA ref_abusy IMPLIES safe_1
p_safe_2      : LEMMA ref_abusy IMPLIES safe_2
p_safe_3      : LEMMA (ref_aerror & spc_1) IMPLIES safe_3
p_safe_4      : LEMMA (ref_ahead & spc_5 & spc_9) IMPLIES safe_4
p_safe_5      : LEMMA (ref_abusy & rpc_1) IMPLIES safe_5
p_safe_6      : LEMMA (ref_aerror & rpc_1) IMPLIES safe_6
p_safe_7      : LEMMA (ref_ahead & rpc_1 & rpc_5 & rpc_9 & head_1)
                        IMPLIES safe_7
p_safe_8      : LEMMA (ref_afile & ref_ahead & rpc_5 & rpc_9 & rpc_10)
                        IMPLIES safe_8
p_safe_9      : LEMMA
                  (ref_ahead & ref_afirst & rpc_3 & rpc_5 & rpc_9 & rpc_11)
                    IMPLIES safe_9
p_safe_10     : LEMMA ref_aerror IMPLIES safe_10


%-- Delta Implications

d_ref_abusy   : LEMMA Delta IMPLIES ref_abusy
d_ref_afirst  : LEMMA Delta IMPLIES ref_afirst
d_ref_aerror  : LEMMA Delta IMPLIES ref_aerror
d_ref_afile   : LEMMA Delta IMPLIES ref_afile
d_ref_ahead   : LEMMA Delta IMPLIES ref_ahead

d_spc_1       : LEMMA Delta IMPLIES spc_1
d_spc_2       : LEMMA Delta IMPLIES spc_2
d_spc_3       : LEMMA Delta IMPLIES spc_3
d_spc_4       : LEMMA Delta IMPLIES spc_4
d_spc_5       : LEMMA Delta IMPLIES spc_5
```

```
d_spc_6        : LEMMA Delta IMPLIES spc_6
d_spc_7        : LEMMA Delta IMPLIES spc_7
d_spc_8        : LEMMA Delta IMPLIES spc_8
d_spc_9        : LEMMA Delta IMPLIES spc_9
d_spc_10       : LEMMA Delta IMPLIES spc_10
d_spc_11       : LEMMA Delta IMPLIES spc_11
d_spc_12       : LEMMA Delta IMPLIES spc_12
d_spc_13       : LEMMA Delta IMPLIES spc_13
d_spc_14       : LEMMA Delta IMPLIES spc_14
d_spc_15       : LEMMA Delta IMPLIES spc_15
d_spc_16       : LEMMA Delta IMPLIES spc_16

d_rpc_1        : LEMMA Delta IMPLIES rpc_1
d_rpc_2        : LEMMA Delta IMPLIES rpc_2
d_rpc_3        : LEMMA Delta IMPLIES rpc_3
d_rpc_4        : LEMMA Delta IMPLIES rpc_4
d_rpc_5        : LEMMA Delta IMPLIES rpc_5
d_rpc_6        : LEMMA Delta IMPLIES rpc_6
d_rpc_7        : LEMMA Delta IMPLIES rpc_7
d_rpc_8        : LEMMA Delta IMPLIES rpc_8
d_rpc_9        : LEMMA Delta IMPLIES rpc_9
d_rpc_10       : LEMMA Delta IMPLIES rpc_10
d_rpc_11       : LEMMA Delta IMPLIES rpc_11

d_K_1          : LEMMA Delta IMPLIES K_1
d_K_2          : LEMMA Delta IMPLIES K_2
d_K_3          : LEMMA Delta IMPLIES K_3
d_K_4          : LEMMA Delta IMPLIES K_4
d_K_5          : LEMMA Delta IMPLIES K_5

d_L_1          : LEMMA Delta IMPLIES L_1
d_L_2          : LEMMA Delta IMPLIES L_2

d_stimer1_1    : LEMMA Delta IMPLIES stimer1_1

d_stimer2_1    : LEMMA Delta IMPLIES stimer2_1
d_stimer2_2    : LEMMA Delta IMPLIES stimer2_2
d_stimer2_3    : LEMMA Delta IMPLIES stimer2_3
```

```
d_rtimer_1    : LEMMA Delta IMPLIES rtimer_1

d_rn_1        : LEMMA Delta IMPLIES rn_1
d_rn_2        : LEMMA Delta IMPLIES rn_2

d_head_1      : LEMMA Delta IMPLIES head_1

d_safe_1      : LEMMA Delta IMPLIES safe_1
d_safe_2      : LEMMA Delta IMPLIES safe_2
d_safe_3      : LEMMA Delta IMPLIES safe_3
d_safe_4      : LEMMA Delta IMPLIES safe_4
d_safe_5      : LEMMA Delta IMPLIES safe_5
d_safe_6      : LEMMA Delta IMPLIES safe_6
d_safe_7      : LEMMA Delta IMPLIES safe_7
d_safe_8      : LEMMA Delta IMPLIES safe_8
d_safe_9      : LEMMA Delta IMPLIES safe_9
d_safe_10     : LEMMA Delta IMPLIES safe_10


%-- Preserved

p_ref_abusy   : LEMMA pi(ref_abusy)
p_ref_afirst  : LEMMA pi(ref_afirst)
p_ref_aerror  : LEMMA pi(ref_aerror)
p_ref_afile   : LEMMA pi(ref_afile)
p_ref_ahead   : LEMMA pi(ref_ahead)

p_spc_1       : LEMMA pi(spc_1)
p_spc_2       : LEMMA pi(spc_2)
p_spc_3       : LEMMA pi(spc_3)
p_spc_4       : LEMMA pi(spc_4)
p_spc_5       : LEMMA pi(spc_5)
p_spc_6       : LEMMA pi(spc_6)
p_spc_7       : LEMMA pi(spc_7)
p_spc_8       : LEMMA pi(spc_8)
p_spc_9       : LEMMA pi(spc_9)
p_spc_10      : LEMMA pi(spc_10)
p_spc_11      : LEMMA pi(spc_11)
p_spc_12      : LEMMA pi(spc_12)
```

```
p_spc_13      : LEMMA pi(spc_13)
p_spc_14      : LEMMA pi(spc_14)
p_spc_15      : LEMMA pi(spc_15)
p_spc_16      : LEMMA pi(spc_16)

p_rpc_1       : LEMMA pi(rpc_1)
p_rpc_2       : LEMMA pi(rpc_2)
p_rpc_3       : LEMMA pi(rpc_3)
p_rpc_4       : LEMMA pi(rpc_4)
p_rpc_5       : LEMMA pi(rpc_5)
p_rpc_6       : LEMMA pi(rpc_6)
p_rpc_7       : LEMMA pi(rpc_7)
p_rpc_8       : LEMMA pi(rpc_8)
p_rpc_9       : LEMMA pi(rpc_9)
p_rpc_10      : LEMMA pi(rpc_10)
p_rpc_11      : LEMMA pi(rpc_11)

p_K_1         : LEMMA pi(K_1)
p_K_2         : LEMMA pi(K_2)
p_K_3         : LEMMA pi(K_3)
p_K_4         : LEMMA pi(K_4)
p_K_5         : LEMMA pi(K_5)

p_L_1         : LEMMA pi(L_1)
p_L_2         : LEMMA pi(L_2)

p_stimer1_1   : LEMMA pi(stimer1_1)

p_stimer2_1   : LEMMA pi(stimer2_1)
p_stimer2_2   : LEMMA pi(stimer2_2)
p_stimer2_3   : LEMMA pi(stimer2_3)

p_rtimer_1    : LEMMA pi(rtimer_1)

p_rn_1        : LEMMA pi(rn_1)
p_rn_2        : LEMMA pi(rn_2)

p_head_1      : LEMMA pi(head_1)
```

```
  p_safe        : LEMMA pi(safe)

  pi_and: LEMMA FORALL (X, Y: pred[State]):
                 (pi(X) AND pi(Y)) IMPLIES pi(X & Y)

  p_Delta : LEMMA pi(Delta)


  %-- Main Invariant

  invariant     : THEOREM invariant(I)

END protocol_safe
```

B.2 Tactics

```
;;; General preservation properties


(defstep try-out (A)
  (try (A) (fail) (skip))
  "Tryes to prove strategy (A)."
  "Trying")


(defstrat pr ()
  (then
    (preserved)
    (action))
  "Proves preservation property."
  "Proving preservation property")


(defstep preserved ()
  (then
    (expand "pi")
```

```
    (expand "preserved")
    (spread
      (split)
      ((grind)
       (then
         (skosimp)
         (expand "I")
         (expand "&")
         (flatten)
         (auto-rewrite-theory "protocol" :always? T)
         (auto-rewrite-theory "protocol_safe" :always? T)
         (stop-rewrite "Delta")
         (expand "transition")
         (split)))))
  "Prepares Proof of preservation property."
  "Preparing proof of preservation property")


(defstep action ()
  (then
    (generate-wp)
    (try-out prove-wp))
  "Proves an action."
  "Proving action")


(defstep generate-wp ()
  (then
    (skolem!)
    (replace*)
    (delete -1)
    (do-rewrite)
    (repeat* (then (lift-if +)(split +)(flatten)))
    (assert))
  "Generates Weakest Pre-condition."
  "Genrating Weakest Pre-condition")


(defstep prove-wp ()
```

```
  (then
    (musimp)
    (grind NIL))
  "Proves a weakest pre-condition."
  "Proving weakest pre-condition")


(defstep begin-action ()
  (then
    (skolem!)
    (replace -1)
    (delete -1))
  "Replaces post-state."
  "Replacing post-state")


(defstep end-action ()
  (then
    (expand "IMPLIES")
    (repeat (inst?))
    (do-rewrite)
    (lift-if)
    (musimp)
    (grind NIL))
  "Proves an action on the basis of lemmas."
  "Proving action on basis of lemmas")


;;; Consequences of Delta


(defstep delta-implies ()
  (then
    (expand "implies")
    (skolem!)
    (expand "Delta")
    (expand "&")
    (flatten))
  "Proves direct Delta consequence."
```

```
  "Proving direct Delta consequence")


(defstep consequence ()
  (then
    (expand "IMPLIES")
    (skolem!)
    (expand "&")
    (grind))
  "Proves a simple logical consequence on state predicates."
  "Proving logical consequence")


(defstep delta-consq (l)
  (then
    (lemma l)
    (expand "IMPLIES")
    (expand "&")
    (skolem!)
    (inst?)
    (expand "Delta")
    (expand "&")
    (flatten)
    (bddsimp))
  "Proves an indirect Delta consequence."
  "Proving indirect Delta consequence")


;;; Preservation of Delta


(defstep include-lemmas ()
  (then*
    (lemma "p_ref_abusy")
    (lemma "p_ref_afirst")
    (lemma "p_ref_aerror")
    (lemma "p_ref_afile")
    (lemma "p_ref_ahead")
    (lemma "p_spc_1")
```

```
(lemma "p_spc_2")
(lemma "p_spc_3")
(lemma "p_spc_4")
(lemma "p_spc_5")
(lemma "p_spc_6")
(lemma "p_spc_7")
(lemma "p_spc_8")
(lemma "p_spc_9")
(lemma "p_spc_10")
(lemma "p_spc_11")
(lemma "p_spc_12")
(lemma "p_spc_13")
(lemma "p_spc_14")
(lemma "p_spc_15")
(lemma "p_spc_16")
(lemma "p_rpc_1")
(lemma "p_rpc_2")
(lemma "p_rpc_3")
(lemma "p_rpc_4")
(lemma "p_rpc_5")
(lemma "p_rpc_6")
(lemma "p_rpc_7")
(lemma "p_rpc_8")
(lemma "p_rpc_9")
(lemma "p_rpc_10")
(lemma "p_rpc_11")
(lemma "p_K_1")
(lemma "p_K_2")
(lemma "p_K_3")
(lemma "p_K_4")
(lemma "p_K_5")
(lemma "p_L_1")
(lemma "p_L_2")
(lemma "p_stimer1_1")
(lemma "p_stimer2_1")
(lemma "p_stimer2_2")
(lemma "p_stimer2_3")
(lemma "p_rtimer_1")
(lemma "p_rn_1")
```

```
    (lemma "p_rn_2")
    (lemma "p_head_1"))
  "Includes all the invariants."
  "Including invariants")


(defstep delta-preserved ()
  (then
    (include-lemmas)
    (expand "Delta")
    (repeat* (then (rewrite "pi_and" 1)(delete 2))))
  "Proves preservation of Delta."
  "Proving preservation of Delta")
```

B.3 Proof Scripts

Note that the program contains 17 rules as enumerated below:

1. Rule_write_req
2. Rule_read_conf
3. Rule_read_ind
4. Rule_read_ind_error
5. Rule_lose_msg
6. Rule_lose_ack
7. Rule_read_req
8. Rule_write_K
9. Rule_read_L
10. Rule_write_conf
11. Rule_stimer1
12. Rule_stimer2
13. Rule_read_K
14. Rule_write_ind
15. Rule_write_L
16. Rule_write_ind_error
17. Rule_rtimer

Since many proofs decompose into a subgoal for each of these rules, this numbering can also be found in the proofs, see for example the proof of lemma `p_ref_abusy`.

```
(|protocol|
 (|incr_rn_TCC1| "" (TCC :DEFS EXPLICIT) NIL)
 (|incr_head_TCC1| "" (GRIND) NIL)
 (|automaton_TCC1| "" (SUBTYPE-TCC) NIL)
 (|automaton_TCC2| "" (SUBTYPE-TCC) NIL)
 (|null_data_TCC1| "" (POSTPONE) NIL)
 (|startstate_TCC1| "" (TCC :DEFS EXPLICIT) NIL)
 (|startstate_TCC2| "" (TCC :DEFS EXPLICIT) NIL)
 (|Rule_read_req_TCC1| "" (TCC :DEFS EXPLICIT) NIL)
 (|Rule_read_L_TCC1| "" (TCC :DEFS EXPLICIT) NIL)
 (|Rule_write_conf_TCC1| "" (TCC :DEFS EXPLICIT) NIL)
 (|Rule_write_conf_TCC2| "" (TCC :DEFS EXPLICIT) NIL))
(|protocol_safe|
 (|p_safe_1| "" (CONSEQUENCE) NIL)
 (|p_safe_2| "" (CONSEQUENCE) NIL)
 (|p_safe_3| "" (CONSEQUENCE) NIL)
 (|p_safe_4| "" (CONSEQUENCE) NIL)
 (|p_safe_5| "" (CONSEQUENCE) NIL)
 (|p_safe_6| "" (CONSEQUENCE) NIL)
 (|p_safe_7| "" (CONSEQUENCE) NIL)
 (|p_safe_8| "" (CONSEQUENCE) NIL)
 (|p_safe_9| "" (CONSEQUENCE) NIL)
 (|p_safe_10| "" (CONSEQUENCE) NIL)
 (|d_ref_abusy| "" (DELTA-IMPLIES) NIL)
 (|d_ref_afirst| "" (DELTA-IMPLIES) NIL)
 (|d_ref_aerror| "" (DELTA-IMPLIES) NIL)
 (|d_ref_afile| "" (DELTA-IMPLIES) NIL)
 (|d_ref_ahead| "" (DELTA-IMPLIES) NIL)
 (|d_spc_1| "" (DELTA-IMPLIES) NIL)
 (|d_spc_2| "" (DELTA-IMPLIES) NIL)
 (|d_spc_3| "" (DELTA-IMPLIES) NIL)
 (|d_spc_4| "" (DELTA-IMPLIES) NIL)
 (|d_spc_5| "" (DELTA-IMPLIES) NIL)
 (|d_spc_6| "" (DELTA-IMPLIES) NIL)
```

```
(|d_spc_7| "" (DELTA-IMPLIES) NIL)
(|d_spc_8| "" (DELTA-IMPLIES) NIL)
(|d_spc_9| "" (DELTA-IMPLIES) NIL)
(|d_spc_10| "" (DELTA-IMPLIES) NIL)
(|d_spc_11| "" (DELTA-IMPLIES) NIL)
(|d_spc_12| "" (DELTA-IMPLIES) NIL)
(|d_spc_13| "" (DELTA-IMPLIES) NIL)
(|d_spc_14| "" (DELTA-IMPLIES) NIL)
(|d_spc_15| "" (DELTA-IMPLIES) NIL)
(|d_spc_16| "" (DELTA-IMPLIES) NIL)
(|d_rpc_1| "" (DELTA-IMPLIES) NIL)
(|d_rpc_2| "" (DELTA-IMPLIES) NIL)
(|d_rpc_3| "" (DELTA-IMPLIES) NIL)
(|d_rpc_4| "" (DELTA-IMPLIES) NIL)
(|d_rpc_5| "" (DELTA-IMPLIES) NIL)
(|d_rpc_6| "" (DELTA-IMPLIES) NIL)
(|d_rpc_7| "" (DELTA-IMPLIES) NIL)
(|d_rpc_8| "" (DELTA-IMPLIES) NIL)
(|d_rpc_9| "" (DELTA-IMPLIES) NIL)
(|d_rpc_10| "" (DELTA-IMPLIES) NIL)
(|d_rpc_11| "" (DELTA-IMPLIES) NIL)
(|d_K_1| "" (DELTA-IMPLIES) NIL)
(|d_K_2| "" (DELTA-IMPLIES) NIL)
(|d_K_3| "" (DELTA-IMPLIES) NIL)
(|d_K_4| "" (DELTA-IMPLIES) NIL)
(|d_K_5| "" (DELTA-IMPLIES) NIL)
(|d_L_1| "" (DELTA-IMPLIES) NIL)
(|d_L_2| "" (DELTA-IMPLIES) NIL)
(|d_stimer1_1| "" (DELTA-IMPLIES) NIL)
(|d_stimer2_1| "" (DELTA-IMPLIES) NIL)
(|d_stimer2_2| "" (DELTA-IMPLIES) NIL)
(|d_stimer2_3| "" (DELTA-IMPLIES) NIL)
(|d_rtimer_1| "" (DELTA-IMPLIES) NIL)
(|d_rn_1| "" (DELTA-IMPLIES) NIL)
(|d_rn_2| "" (DELTA-IMPLIES) NIL)
(|d_head_1| "" (DELTA-IMPLIES) NIL)
(|d_safe_1| "" (DELTA-CONSQ "p_safe_1") NIL)
(|d_safe_2| "" (DELTA-CONSQ "p_safe_2") NIL)
(|d_safe_3| "" (DELTA-CONSQ "p_safe_3") NIL)
```

```
(|d_safe_4| "" (DELTA-CONSQ "p_safe_4") NIL)
(|d_safe_5| "" (DELTA-CONSQ "p_safe_5") NIL)
(|d_safe_6| "" (DELTA-CONSQ "p_safe_6") NIL)
(|d_safe_7| "" (DELTA-CONSQ "p_safe_7") NIL)
(|d_safe_8| "" (DELTA-CONSQ "p_safe_8") NIL)
(|d_safe_9| "" (DELTA-CONSQ "p_safe_9") NIL)
(|d_safe_10| "" (DELTA-CONSQ "p_safe_10") NIL)
(|p_ref_abusy| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_ref_afirst| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_ref_ahead")
     (("14" (LEMMA "d_rpc_3")
       (("14" (LEMMA "d_rpc_5")
         (("14" (LEMMA "d_rpc_9") (("14" (END-ACTION) NIL))))))))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_ref_aerror| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_ref_afirst")
     (("10" (LEMMA "d_spc_1")
       (("10" (LEMMA "d_spc_6")
         (("10" (LEMMA "d_rpc_8") (("10" (END-ACTION) NIL))))))))))
```

```
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_stimer2_2") (("12" (END-ACTION) NIL)))))
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_1") (("13" (END-ACTION) NIL)))))
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (BEGIN-ACTION)
   (("15" (LEMMA "d_rpc_1") (("15" (END-ACTION) NIL)))))
  ("16" (ACTION) NIL)
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_ref_afile| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_ref_ahead| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_spc_7") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_2") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_spc_5") (("10" (END-ACTION) NIL)))))
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_spc_4")
     (("12" (LEMMA "d_stimer2_1")
       (("12" (LEMMA "d_rtimer_1") (("12" (END-ACTION) NIL)))))))))
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1")
```

```
    (("14" (LEMMA "d_rpc_5")
      (("14" (LEMMA "d_rpc_9")
        (("14" (LEMMA "d_spc_15")
          (("14" (END-ACTION) NIL)))))))))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_spc_1| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_1") (("13" (END-ACTION) NIL)))))
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_spc_2| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_spc_7") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_2") (("9" (END-ACTION) NIL)))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_spc_3")
     (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
```

```
(|p_spc_3| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_4| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_1") (("13" (END-ACTION) NIL)))))
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_5| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_2") (("9" (END-ACTION) NIL)))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1")
     (("11" (LEMMA "d_head_1") (("11" (END-ACTION) NIL))))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_spc_6| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_spc_3")
```

```
    (("9" (LEMMA "d_L_2")
      (("9" (LEMMA "d_rn_2") (("9" (END-ACTION) NIL))))))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1")
     (("11" (LEMMA "d_head_1") (("11" (END-ACTION) NIL)))))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_7| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_spc_5") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL)
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_spc_4")
     (("12" (LEMMA "d_stimer2_1")
       (("12" (LEMMA "d_rtimer_1") (("12" (END-ACTION) NIL)))))))))
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_8| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_9| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_spc_3") (("9" (END-ACTION) NIL)))))
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
```

```
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_10| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_spc_11") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_spc_3")
     (("9" (LEMMA "d_L_2")
       (("9" (LEMMA "d_rn_2") (("9" (END-ACTION) NIL)))))))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_spc_12")
     (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_spc_1") (("16" (END-ACTION) NIL)))))
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_spc_11| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_spc_13") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL)
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_spc_14")
     (("12" (LEMMA "d_stimer2_1")
       (("12" (LEMMA "d_stimer2_3") (("12" (END-ACTION) NIL)))))))))
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
```

```
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_rpc_2") (("16" (END-ACTION) NIL)))))
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_spc_12| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL)
  ("8" (BEGIN-ACTION)
   (("8" (LEMMA "d_spc_10") (("8" (END-ACTION) NIL)))))
  ("9" (ACTION) NIL) ("10" (ACTION) NIL) ("11" (ACTION) NIL)
  ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_5") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_rpc_2") (("16" (END-ACTION) NIL)))))
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_spc_13| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_spc_3")
     (("9" (LEMMA "d_L_2")
       (("9" (LEMMA "d_rn_2") (("9" (END-ACTION) NIL)))))))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_spc_12")
     (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_rpc_2") (("16" (END-ACTION) NIL)))))
```

```
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL))))))
(|p_spc_14| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_15| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL)
  ("8" (BEGIN-ACTION)
   (("8" (LEMMA "d_spc_16") (("8" (END-ACTION) NIL)))))
  ("9" (ACTION) NIL) ("10" (ACTION) NIL) ("11" (ACTION) NIL)
  ("12" (ACTION) NIL) ("13" (ACTION) NIL) ("14" (ACTION) NIL)
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_spc_16| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_spc_15") (("9" (END-ACTION) NIL)))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_spc_15")
     (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL) ("14" (ACTION) NIL)
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rpc_1| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
```

```
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_1") (("13" (END-ACTION) NIL)))))
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_rpc_2| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL)
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_rpc_3| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_rpc_1") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_rpc_1") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_2") (("13" (END-ACTION) NIL)))))
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_rpc_4| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
```

```
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rpc_5| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_rpc_1") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_3") (("13" (END-ACTION) NIL)))))
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_rpc_6| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_rpc_1") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_3") (("13" (END-ACTION) NIL)))))
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_5") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rpc_7| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL)
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL))))))))
(|p_rpc_8| "" (PRESERVED)
```

```
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rpc_9| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rpc_10| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_spc_1") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_spc_1") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_4") (("13" (END-ACTION) NIL)))))
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_rpc_11| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (BEGIN-ACTION)
   (("13" (LEMMA "d_K_5") (("13" (END-ACTION) NIL)))))
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_K_1| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
```

```
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL)
  ("8" (BEGIN-ACTION)
   (("8" (LEMMA "d_spc_1")
     (("8" (LEMMA "d_L_1") (("8" (END-ACTION) NIL)))))))
  ("9" (ACTION) NIL) ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL)
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_K_2| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_K_1") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_K_1") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_K_3| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_1") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_K_1") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_K_4| "" (PRESERVED)
```

```
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_K_1") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL)
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_K_1") (("9" (END-ACTION) NIL)))))
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_K_1") (("10" (END-ACTION) NIL)))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (ACTION) NIL) ("15" (ACTION) NIL) ("16" (ACTION) NIL)
  ("17" (ACTION) NIL)))
(|p_K_5| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL)
  ("8" (BEGIN-ACTION)
   (("8" (LEMMA "d_spc_10") (("8" (END-ACTION) NIL)))))
  ("9" (ACTION) NIL) ("10" (ACTION) NIL) ("11" (ACTION) NIL)
  ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_K_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_K_1") (("16" (END-ACTION) NIL)))))
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_L_1| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))
  ("13" (ACTION) NIL) ("14" (ACTION) NIL)
  ("15" (BEGIN-ACTION)
   (("15" (LEMMA "d_rpc_1")
```

```
      (("15" (LEMMA "d_K_1") (("15" (END-ACTION) NIL)))))))
   ("16" (ACTION) NIL)
   ("17" (BEGIN-ACTION)
    (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
 (|p_L_2| "" (PRESERVED)
  (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
   ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
   ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
   ("10" (BEGIN-ACTION)
    (("10" (LEMMA "d_L_1") (("10" (END-ACTION) NIL)))))
   ("11" (ACTION) NIL) ("12" (ACTION) NIL) ("13" (ACTION) NIL)
   ("14" (BEGIN-ACTION)
    (("14" (LEMMA "d_L_1") (("14" (END-ACTION) NIL)))))
   ("15" (BEGIN-ACTION)
    (("15" (LEMMA "d_rpc_6")
      (("15" (LEMMA "d_rpc_7") (("15" (END-ACTION) NIL)))))))
   ("16" (ACTION) NIL)
   ("17" (BEGIN-ACTION)
    (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
 (|p_stimer1_1| "" (PRESERVED)
  (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
   ("4" (ACTION) NIL)
   ("5" (BEGIN-ACTION)
    (("5" (LEMMA "d_K_1") (("5" (END-ACTION) NIL)))))
   ("6" (BEGIN-ACTION)
    (("6" (LEMMA "d_L_1") (("6" (END-ACTION) NIL)))))
   ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
   ("10" (ACTION) NIL) ("11" (ACTION) NIL)
   ("12" (BEGIN-ACTION)
    (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))
   ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
   ("16" (ACTION) NIL)
   ("17" (BEGIN-ACTION)
    (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
 (|p_stimer2_1| "" (PRESERVED)
  (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
   ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
   ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
   ("10" (ACTION) NIL)
```

```
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL) ("14" (ACTION) NIL)
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_rpc_2")
     (("16" (LEMMA "d_K_1")
       (("16" (LEMMA "d_L_1") (("16" (END-ACTION) NIL)))))))))
  ("17" (BEGIN-ACTION)
   (("17" (LEMMA "d_rtimer_1") (("17" (END-ACTION) NIL)))))))
(|p_stimer2_2| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_stimer2_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_stimer2_3| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_stimer2_1") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rtimer_1| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_spc_1")
     (("10" (LEMMA "d_K_1")
       (("10" (LEMMA "d_L_1")
         (("10" (LEMMA "d_stimer2_1")
           (("10" (END-ACTION) NIL)))))))))))
  ("11" (BEGIN-ACTION)
```

```
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL) ("14" (ACTION) NIL)
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rn_1| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL) ("11" (ACTION) NIL)
  ("12" (BEGIN-ACTION)
   (("12" (LEMMA "d_spc_8")
     (("12" (LEMMA "d_stimer2_1") (("12" (END-ACTION) NIL)))))))
  ("13" (ACTION) NIL) ("14" (ACTION) NIL) ("15" (ACTION) NIL)
  ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
(|p_rn_2| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_rn_1") (("7" (END-ACTION) NIL)))))
  ("8" (BEGIN-ACTION)
   (("8" (LEMMA "d_spc_2") (("8" (END-ACTION) NIL)))))
  ("9" (BEGIN-ACTION)
   (("9" (LEMMA "d_L_2") (("9" (END-ACTION) NIL)))))
  ("10" (ACTION) NIL) ("11" (ACTION) NIL) ("12" (ACTION) NIL)
  ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_rpc_3") (("14" (END-ACTION) NIL)))))
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_rpc_4") (("16" (END-ACTION) NIL)))))
  ("17" (ACTION) NIL)))
(|p_head_1| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (ACTION) NIL) ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (ACTION) NIL)
  ("11" (BEGIN-ACTION)
   (("11" (LEMMA "d_stimer1_1") (("11" (END-ACTION) NIL)))))
  ("12" (ACTION) NIL) ("13" (ACTION) NIL) ("14" (ACTION) NIL)
  ("15" (ACTION) NIL) ("16" (ACTION) NIL) ("17" (ACTION) NIL)))
```

```
(|p_safe| "" (PRESERVED)
 (("1" (ACTION) NIL) ("2" (ACTION) NIL) ("3" (ACTION) NIL)
  ("4" (ACTION) NIL) ("5" (ACTION) NIL) ("6" (ACTION) NIL)
  ("7" (BEGIN-ACTION)
   (("7" (LEMMA "d_safe_1") (("7" (END-ACTION) NIL)))))
  ("8" (ACTION) NIL) ("9" (ACTION) NIL)
  ("10" (BEGIN-ACTION)
   (("10" (LEMMA "d_safe_2")
     (("10" (LEMMA "d_safe_3")
       (("10" (LEMMA "d_safe_4") (("10" (END-ACTION) NIL)))))))))
  ("11" (ACTION) NIL) ("12" (ACTION) NIL) ("13" (ACTION) NIL)
  ("14" (BEGIN-ACTION)
   (("14" (LEMMA "d_safe_5")
     (("14" (LEMMA "d_safe_6")
       (("14" (LEMMA "d_safe_7")
         (("14" (LEMMA "d_safe_8")
           (("14" (LEMMA "d_safe_9")
             (("14" (END-ACTION) NIL))))))))))))
  ("15" (ACTION) NIL)
  ("16" (BEGIN-ACTION)
   (("16" (LEMMA "d_safe_10") (("16" (END-ACTION) NIL)))))
  ("17" (ACTION) NIL)))
(|pi_and| "" (SKOLEM!)
 (("" (FLATTEN)
   (("" (EXPAND "pi")
     (("" (EXPAND "preserved")
       (("" (FLATTEN)
         (("" (SPLIT +)
           (("1" (EXPAND "&")
             (("1" (BDDSIMP) (("1" (PROPAX) NIL))))
            ("2" (DELETE -1 -3)
             (("2" (SKOLEM!)
               (("2" (FLATTEN)
                 (("2" (EXPAND "&")
                   (("2" (FLATTEN)
                     (("2" (INST?)
                       (("2" (INST?)
                         (("2" (BDDSIMP)
                           (("2"
```

```
                              (PROPAX)
                              NIL)))))))))))))))))))))))))))))
(|p_Delta| "" (DELTA-PRESERVED) NIL)
(|invariant| "" (EXPAND "invariant")
 (("" (SKOSIMP)
   (("" (EXPAND "program")
     (("" (FLATTEN)
       (("" (INDUCT "n")
         (("1" (DELETE -2)
           (("1" (EXPAND "I")
             (("1" (EXPAND "&")
               (("1" (REPLACE*)
                 (("1" (DELETE -1)
                   (("1" (SPLIT)
                     (("1" (LEMMA "p_Delta")
                       (("1" (EXPAND "pi")
                         (("1" (EXPAND "preserved")
                           (("1"
                             (FLATTEN)
                             (("1" (PROPAX) NIL)))))))))
                      ("2" (GRIND) NIL)))))))))))))
          ("2" (DELETE -1)
           (("2" (SKOSIMP)
             (("2" (INST?)
               (("2" (EXPAND "I" +)
                 (("2" (EXPAND "&")
                   (("2" (SPLIT)
                     (("1" (LEMMA "p_Delta")
                       (("1" (EXPAND "pi")
                         (("1" (EXPAND "preserved")
                           (("1"
                             (FLATTEN)
                             (("1"
                               (INST?)
                               (("1"
                                 (COPY -3)
                                 (("1"
                                   (EXPAND "I" -1)
                                   (("1"
```

```
                          (EXPAND "&")
                          (("1"
                            (FLATTEN)
                            (("1"
                              (BDDSIMP)
                              (("1"
                                (PROPAX)
                                NIL)))))))))))))))))))))
           ("2" (LEMMA "p_safe")
            (("2" (EXPAND "pi")
              (("2" (EXPAND "preserved")
                (("2"
                  (FLATTEN)
                  (("2"
                    (DELETE -1)
                    (("2"
                      (INST?)
                      (("2"
                        (COPY -2)
                        (("2"
                          (EXPAND "I")
                          (("2"
                            (EXPAND "&")
                            (("2"
                              (FLATTEN)
                              (("2"
                                (BDDSIMP)
                                (("2"
                                  (PROPAX)
                                  NIL)
)))))))))))))))))))))))))))))))))))))))))))))
```

B.4 Overview of Lemmas

```
Proof summary for theory protocol
   incr_rn_TCC1...........................................proved - complete
```

```
   incr_head_TCC1.........................................proved - complete
   automaton_TCC1.........................................proved - complete
   automaton_TCC2.........................................proved - complete
   startstate_TCC1........................................proved - complete
   startstate_TCC2........................................proved - complete
   Rule_read_req_TCC1.....................................proved - complete
   Rule_read_L_TCC1.......................................proved - complete
   Rule_write_conf_TCC1...................................proved - complete
   Rule_write_conf_TCC2...................................proved - complete
   Theory totals: 10 formulas, 10 attempted, 10 succeeded.

Proof summary for theory protocol_safe
   p_safe_1...............................................proved - complete
   p_safe_2...............................................proved - complete
   p_safe_3...............................................proved - complete
   p_safe_4...............................................proved - complete
   p_safe_5...............................................proved - complete
   p_safe_6...............................................proved - complete
   p_safe_7...............................................proved - complete
   p_safe_8...............................................proved - complete
   p_safe_9...............................................proved - complete
   p_safe_10..............................................proved - complete
   d_ref_abusy............................................proved - complete
   d_ref_afirst...........................................proved - complete
   d_ref_aerror...........................................proved - complete
   d_ref_afile............................................proved - complete
   d_ref_ahead............................................proved - complete
   d_spc_1................................................proved - complete
   d_spc_2................................................proved - complete
   d_spc_3................................................proved - complete
   d_spc_4................................................proved - complete
   d_spc_5................................................proved - complete
   d_spc_6................................................proved - complete
   d_spc_7................................................proved - complete
   d_spc_8................................................proved - complete
   d_spc_9................................................proved - complete
   d_spc_10...............................................proved - complete
   d_spc_11...............................................proved - complete
   d_spc_12...............................................proved - complete
```

```
d_spc_13.............................................proved - complete
d_spc_14.............................................proved - complete
d_spc_15.............................................proved - complete
d_spc_16.............................................proved - complete
d_rpc_1..............................................proved - complete
d_rpc_2..............................................proved - complete
d_rpc_3..............................................proved - complete
d_rpc_4..............................................proved - complete
d_rpc_5..............................................proved - complete
d_rpc_6..............................................proved - complete
d_rpc_7..............................................proved - complete
d_rpc_8..............................................proved - complete
d_rpc_9..............................................proved - complete
d_rpc_10.............................................proved - complete
d_rpc_11.............................................proved - complete
d_K_1................................................proved - complete
d_K_2................................................proved - complete
d_K_3................................................proved - complete
d_K_4................................................proved - complete
d_K_5................................................proved - complete
d_L_1................................................proved - complete
d_L_2................................................proved - complete
d_stimer1_1..........................................proved - complete
d_stimer2_1..........................................proved - complete
d_stimer2_2..........................................proved - complete
d_stimer2_3..........................................proved - complete
d_rtimer_1...........................................proved - complete
d_rn_1...............................................proved - complete
d_rn_2...............................................proved - complete
d_head_1.............................................proved - complete
d_safe_1.............................................proved - complete
d_safe_2.............................................proved - complete
d_safe_3.............................................proved - complete
d_safe_4.............................................proved - complete
d_safe_5.............................................proved - complete
d_safe_6.............................................proved - complete
d_safe_7.............................................proved - complete
d_safe_8.............................................proved - complete
d_safe_9.............................................proved - complete
```

```
d_safe_10.............................................proved - complete
p_ref_abusy...........................................proved - complete
p_ref_afirst..........................................proved - complete
p_ref_aerror..........................................proved - complete
p_ref_afile...........................................proved - complete
p_ref_ahead...........................................proved - complete
p_spc_1...............................................proved - complete
p_spc_2...............................................proved - complete
p_spc_3...............................................proved - complete
p_spc_4...............................................proved - complete
p_spc_5...............................................proved - complete
p_spc_6...............................................proved - complete
p_spc_7...............................................proved - complete
p_spc_8...............................................proved - complete
p_spc_9...............................................proved - complete
p_spc_10..............................................proved - complete
p_spc_11..............................................proved - complete
p_spc_12..............................................proved - complete
p_spc_13..............................................proved - complete
p_spc_14..............................................proved - complete
p_spc_15..............................................proved - complete
p_spc_16..............................................proved - complete
p_rpc_1...............................................proved - complete
p_rpc_2...............................................proved - complete
p_rpc_3...............................................proved - complete
p_rpc_4...............................................proved - complete
p_rpc_5...............................................proved - complete
p_rpc_6...............................................proved - complete
p_rpc_7...............................................proved - complete
p_rpc_8...............................................proved - complete
p_rpc_9...............................................proved - complete
p_rpc_10..............................................proved - complete
p_rpc_11..............................................proved - complete
p_K_1.................................................proved - complete
p_K_2.................................................proved - complete
p_K_3.................................................proved - complete
p_K_4.................................................proved - complete
p_K_5.................................................proved - complete
p_L_1.................................................proved - complete
```

```
    p_L_2.................................................proved - complete
    p_stimer1_1...........................................proved - complete
    p_stimer2_1...........................................proved - complete
    p_stimer2_2...........................................proved - complete
    p_stimer2_3...........................................proved - complete
    p_rtimer_1............................................proved - complete
    p_rn_1................................................proved - complete
    p_rn_2................................................proved - complete
    p_head_1..............................................proved - complete
    p_safe................................................proved - complete
    pi_and................................................proved - complete
    p_Delta...............................................proved - complete
    invariant.............................................proved - complete
    Theory totals: 118 formulas, 118 attempted, 118 succeeded.

Grand Totals: 128 proofs, 128 attempted, 128 succeeded.
```

B.5 Dependencies Between Lemmas

```
d_safe_1 depends on the following proved theorems:
    protocol_safe.p_safe_1

d_safe_2 depends on the following proved theorems:
    protocol_safe.p_safe_2

d_safe_3 depends on the following proved theorems:
    protocol_safe.p_safe_3

d_safe_4 depends on the following proved theorems:
    protocol_safe.p_safe_4

d_safe_5 depends on the following proved theorems:
    protocol_safe.p_safe_5

d_safe_6 depends on the following proved theorems:
    protocol_safe.p_safe_6
```

```
d_safe_7 depends on the following proved theorems:
    protocol_safe.p_safe_7

d_safe_8 depends on the following proved theorems:
    protocol_safe.p_safe_8

d_safe_9 depends on the following proved theorems:
    protocol_safe.p_safe_9

d_safe_10 depends on the following proved theorems:
    protocol_safe.p_safe_10

p_ref_abusy depends on the following proved theorems:
    protocol_safe.d_stimer2_1
    protocol_safe.d_stimer1_1

p_ref_afirst depends on the following proved theorems:
    protocol_safe.d_rpc_9
    protocol_safe.d_rpc_5
    protocol_safe.d_rpc_3
    protocol_safe.d_ref_ahead

p_ref_aerror depends on the following proved theorems:
    protocol_safe.d_spc_6
    protocol_safe.d_spc_1
    protocol_safe.d_K_1
    protocol_safe.d_rpc_1
    protocol_safe.d_rpc_8
    protocol_safe.d_stimer2_2
    protocol_safe.d_rtimer_1
    protocol_safe.d_ref_afirst
    protocol_safe.d_stimer1_1

p_ref_ahead depends on the following proved theorems:
    protocol_safe.d_spc_5
    protocol_safe.d_rpc_9
    protocol_safe.d_spc_7
    protocol_safe.d_spc_15
    protocol_safe.d_rpc_5
```

```
    protocol_safe.d_L_2
    protocol_safe.d_rpc_1
    protocol_safe.d_spc_4
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer1_1

p_spc_1 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_K_1
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer1_1

p_spc_2 depends on the following proved theorems:
    protocol_safe.d_spc_7
    protocol_safe.d_L_2
    protocol_safe.d_rpc_1
    protocol_safe.d_spc_3
    protocol_safe.d_stimer1_1

p_spc_4 depends on the following proved theorems:
    protocol_safe.d_K_1
    protocol_safe.d_rpc_1

p_spc_5 depends on the following proved theorems:
    protocol_safe.d_head_1
    protocol_safe.d_L_2
    protocol_safe.d_rpc_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer1_1

p_spc_6 depends on the following proved theorems:
    protocol_safe.d_rn_2
    protocol_safe.d_head_1
    protocol_safe.d_L_2
    protocol_safe.d_rpc_1
    protocol_safe.d_spc_3
    protocol_safe.d_stimer1_1
```

```
p_spc_7 depends on the following proved theorems:
    protocol_safe.d_spc_5
    protocol_safe.d_rpc_1
    protocol_safe.d_spc_4
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1

p_spc_9 depends on the following proved theorems:
    protocol_safe.d_rpc_1
    protocol_safe.d_spc_3

p_spc_10 depends on the following proved theorems:
    protocol_safe.d_rn_2
    protocol_safe.d_spc_1
    protocol_safe.d_L_2
    protocol_safe.d_rpc_1
    protocol_safe.d_spc_3
    protocol_safe.d_spc_11
    protocol_safe.d_spc_12
    protocol_safe.d_stimer1_1
    protocol_safe.d_rtimer_1

p_spc_11 depends on the following proved theorems:
    protocol_safe.d_spc_13
    protocol_safe.d_rpc_2
    protocol_safe.d_rpc_1
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer2_3
    protocol_safe.d_spc_14

p_spc_12 depends on the following proved theorems:
    protocol_safe.d_spc_10
    protocol_safe.d_rpc_2
    protocol_safe.d_rpc_5
    protocol_safe.d_rtimer_1

p_spc_13 depends on the following proved theorems:
```

```
    protocol_safe.d_rpc_2
    protocol_safe.d_rn_2
    protocol_safe.d_L_2
    protocol_safe.d_rpc_1
    protocol_safe.d_spc_3
    protocol_safe.d_spc_12
    protocol_safe.d_stimer1_1
    protocol_safe.d_rtimer_1

p_spc_15 depends on the following proved theorems:
    protocol_safe.d_spc_16

p_spc_16 depends on the following proved theorems:
    protocol_safe.d_spc_15
    protocol_safe.d_stimer1_1

p_rpc_1 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_K_1
    protocol_safe.d_stimer2_1
    protocol_safe.d_stimer1_1

p_rpc_2 depends on the following proved theorems:
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer1_1

p_rpc_3 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_K_2
    protocol_safe.d_rpc_1

p_rpc_5 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_K_3
    protocol_safe.d_rpc_1

p_rpc_6 depends on the following proved theorems:
    protocol_safe.d_L_1
```

```
    protocol_safe.d_K_3
    protocol_safe.d_rpc_1
    protocol_safe.d_rpc_5

p_rpc_7 depends on the following proved theorems:
    protocol_safe.d_rtimer_1

p_rpc_10 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_spc_1
    protocol_safe.d_K_4

p_rpc_11 depends on the following proved theorems:
    protocol_safe.d_K_5

p_K_1 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_spc_1
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer1_1

p_K_2 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_K_1

p_K_3 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_K_1

p_K_4 depends on the following proved theorems:
    protocol_safe.d_K_1

p_K_5 depends on the following proved theorems:
    protocol_safe.d_spc_10
    protocol_safe.d_K_1
    protocol_safe.d_rtimer_1

p_L_1 depends on the following proved theorems:
```

```
    protocol_safe.d_K_1
    protocol_safe.d_rpc_1
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer1_1

p_L_2 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_rpc_6
    protocol_safe.d_rpc_7
    protocol_safe.d_rtimer_1

p_stimer1_1 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_K_1
    protocol_safe.d_stimer2_1
    protocol_safe.d_rtimer_1

p_stimer2_1 depends on the following proved theorems:
    protocol_safe.d_rpc_2
    protocol_safe.d_L_1
    protocol_safe.d_K_1
    protocol_safe.d_rtimer_1
    protocol_safe.d_stimer1_1

p_stimer2_2 depends on the following proved theorems:
    protocol_safe.d_stimer2_1

p_stimer2_3 depends on the following proved theorems:
    protocol_safe.d_stimer2_1

p_rtimer_1 depends on the following proved theorems:
    protocol_safe.d_L_1
    protocol_safe.d_spc_1
    protocol_safe.d_K_1
    protocol_safe.d_stimer2_1
    protocol_safe.d_stimer1_1

p_rn_1 depends on the following proved theorems:
```

```
    protocol_safe.d_stimer2_1
    protocol_safe.d_spc_8

p_rn_2 depends on the following proved theorems:
    protocol_safe.d_rn_1
    protocol_safe.d_rpc_4
    protocol_safe.d_L_2
    protocol_safe.d_rpc_3
    protocol_safe.d_spc_2

p_head_1 depends on the following proved theorems:
    protocol_safe.d_stimer1_1

p_safe depends on the following proved theorems:
    protocol_safe.d_safe_10
    protocol_safe.p_safe_6
    protocol_safe.p_safe_9
    protocol_safe.d_safe_9
    protocol_safe.p_safe_1
    protocol_safe.p_safe_2
    protocol_safe.p_safe_7
    protocol_safe.d_safe_4
    protocol_safe.d_safe_5
    protocol_safe.d_safe_1
    protocol_safe.d_safe_3
    protocol_safe.d_safe_8
    protocol_safe.p_safe_10
    protocol_safe.d_safe_7
    protocol_safe.d_safe_6
    protocol_safe.p_safe_8
    protocol_safe.p_safe_5
    protocol_safe.d_safe_2
    protocol_safe.p_safe_4
    protocol_safe.p_safe_3

p_Delta depends on the following proved theorems:
    protocol_safe.d_spc_5
    protocol_safe.d_spc_6
    protocol_safe.p_rpc_6
```

```
protocol_safe.d_rpc_1
protocol_safe.p_spc_11
protocol_safe.d_spc_7
protocol_safe.d_stimer2_2
protocol_safe.p_spc_13
protocol_safe.p_rpc_5
protocol_safe.p_spc_4
protocol_safe.p_spc_16
protocol_safe.d_rpc_9
protocol_safe.d_spc_10
protocol_safe.p_stimer1_1
protocol_safe.d_rpc_6
protocol_safe.d_ref_afirst
protocol_safe.d_K_5
protocol_safe.d_rn_1
protocol_safe.p_rpc_4
protocol_safe.d_spc_15
protocol_safe.p_spc_8
protocol_safe.pi_and
protocol_safe.p_rpc_3
protocol_safe.p_rpc_7
protocol_safe.p_K_4
protocol_safe.p_spc_14
protocol_safe.p_L_2
protocol_safe.d_rpc_4
protocol_safe.p_ref_afirst
protocol_safe.d_spc_13
protocol_safe.p_stimer2_2
protocol_safe.d_L_1
protocol_safe.d_rpc_5
protocol_safe.d_K_3
protocol_safe.p_rpc_10
protocol_safe.d_rn_2
protocol_safe.p_rn_1
protocol_safe.p_spc_2
protocol_safe.d_spc_1
protocol_safe.p_ref_aerror
protocol_safe.d_K_2
protocol_safe.p_rtimer_1
```

```
protocol_safe.d_rpc_2
protocol_safe.p_rpc_11
protocol_safe.p_K_1
protocol_safe.p_spc_9
protocol_safe.p_spc_15
protocol_safe.p_rpc_8
protocol_safe.d_K_1
protocol_safe.p_rn_2
protocol_safe.d_head_1
protocol_safe.d_L_2
protocol_safe.p_rpc_2
protocol_safe.d_rpc_3
protocol_safe.d_spc_4
protocol_safe.p_K_3
protocol_safe.d_spc_3
protocol_safe.p_ref_ahead
protocol_safe.d_ref_ahead
protocol_safe.p_spc_10
protocol_safe.p_rpc_9
protocol_safe.p_spc_7
protocol_safe.d_spc_11
protocol_safe.p_spc_1
protocol_safe.d_spc_12
protocol_safe.p_ref_afile
protocol_safe.p_stimer2_3
protocol_safe.p_L_1
protocol_safe.d_spc_16
protocol_safe.d_stimer2_1
protocol_safe.d_stimer1_1
protocol_safe.d_rtimer_1
protocol_safe.p_spc_3
protocol_safe.p_spc_12
protocol_safe.d_spc_8
protocol_safe.d_stimer2_3
protocol_safe.p_rpc_1
protocol_safe.p_K_5
protocol_safe.p_spc_6
protocol_safe.p_K_2
protocol_safe.p_head_1
```

```
    protocol_safe.d_spc_14
    protocol_safe.d_K_4
    protocol_safe.p_stimer2_1
    protocol_safe.d_rpc_7
    protocol_safe.p_ref_abusy
    protocol_safe.d_rpc_8
    protocol_safe.p_spc_5
    protocol_safe.d_spc_2

invariant depends on the following proved theorems:
    protocol_safe.p_K_3
    protocol_safe.p_rpc_6
    protocol_safe.d_rpc_1
    protocol_safe.p_spc_11
    protocol_safe.d_spc_7
    protocol_safe.d_stimer2_2
    protocol_safe.p_spc_13
    protocol_safe.d_safe_10
    protocol_safe.p_rpc_5
    protocol_safe.p_spc_4
    protocol_safe.p_spc_16
    protocol_safe.d_rpc_8
    protocol_safe.d_rpc_9
    protocol_safe.d_spc_10
    protocol_safe.p_spc_3
    protocol_safe.p_stimer1_1
    protocol_safe.p_safe_6
    protocol_safe.p_safe_9
    protocol_safe.d_rpc_6
    protocol_safe.d_ref_afirst
    protocol_safe.d_spc_16
    protocol_safe.d_K_5
    protocol_safe.p_rpc_3
    protocol_safe.d_rn_1
    protocol_safe.p_head_1
    protocol_safe.p_rpc_4
    protocol_safe.d_spc_15
    protocol_safe.d_safe_9
    protocol_safe.pi_and
```

```
protocol_safe.p_safe_1
protocol_safe.p_rpc_7
protocol_safe.p_K_4
protocol_safe.p_spc_14
protocol_safe.p_L_2
protocol_safe.d_rpc_4
protocol_safe.p_ref_afirst
protocol_safe.p_stimer2_2
protocol_safe.d_L_1
protocol_safe.d_rpc_5
protocol_safe.d_K_3
protocol_safe.p_rpc_10
protocol_safe.d_rn_2
protocol_safe.p_rn_1
protocol_safe.p_spc_2
protocol_safe.p_safe
protocol_safe.p_safe_2
protocol_safe.d_spc_1
protocol_safe.p_ref_aerror
protocol_safe.d_K_2
protocol_safe.p_rtimer_1
protocol_safe.d_rpc_2
protocol_safe.p_rpc_11
protocol_safe.p_K_1
protocol_safe.p_ref_afile
protocol_safe.p_spc_9
protocol_safe.p_spc_15
protocol_safe.p_spc_8
protocol_safe.p_safe_7
protocol_safe.p_rpc_8
protocol_safe.p_spc_10
protocol_safe.d_K_1
protocol_safe.d_safe_4
protocol_safe.d_safe_5
protocol_safe.p_rn_2
protocol_safe.d_head_1
protocol_safe.d_L_2
protocol_safe.d_safe_1
protocol_safe.d_safe_3
```

```
protocol_safe.p_rpc_2
protocol_safe.d_spc_3
protocol_safe.d_rpc_3
protocol_safe.d_spc_4
protocol_safe.d_spc_13
protocol_safe.d_safe_8
protocol_safe.p_safe_10
protocol_safe.p_ref_ahead
protocol_safe.d_safe_7
protocol_safe.d_ref_ahead
protocol_safe.d_safe_6
protocol_safe.p_safe_8
protocol_safe.p_Delta
protocol_safe.p_safe_5
naturalnumbers.nat_induction
protocol_safe.p_rpc_9
protocol_safe.p_spc_7
protocol_safe.d_spc_11
protocol_safe.d_safe_2
protocol_safe.p_spc_1
protocol_safe.d_spc_12
protocol_safe.p_stimer2_3
protocol_safe.p_L_1
protocol_safe.d_stimer2_1
protocol_safe.d_stimer1_1
protocol_safe.d_rtimer_1
protocol_safe.p_safe_4
protocol_safe.p_spc_12
protocol_safe.d_spc_8
protocol_safe.d_stimer2_3
protocol_safe.d_spc_6
protocol_safe.p_rpc_1
protocol_safe.p_K_5
protocol_safe.p_spc_6
protocol_safe.p_safe_3
protocol_safe.p_K_2
protocol_safe.d_spc_14
protocol_safe.d_K_4
protocol_safe.p_stimer2_1
```

```
  protocol_safe.d_rpc_7
  protocol_safe.p_ref_abusy
  protocol_safe.d_spc_5
  protocol_safe.p_spc_5
  protocol_safe.d_spc_2
```

B.6 Invariants Formulated in Murϕ

```
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;
```

Appendix C

Abstraction and Model Checking in the μ-calculus

This appendix associates to chapter 5. It contains listings of the specifications, the tactics, the proof scripts, the lemma status, and the lemma dependencies.

C.1 The Finite State PVS Specification

```
protocol_lemmas : THEORY

BEGIN

  IMPORTING protocol_safe

  st : VAR State

  a_spc_1(st):bool =
    spc(st)=SC IMPLIES rpc(st)=WF

  a_safe_4(st):bool =
    (spc(st)=SC AND head(st)=last AND rn(st)/=0)
      IMPLIES
    (ahead(st)=last OR ahead(st)=nil)
```

```
  a_safe_7(st):bool =
    rpc(st)=SI IMPLIES ahead(st)/=nil

  a_safe_8(st): bool =
    rpc(st) = SI IMPLIES data(msg(st)) = afile(st)(ahead(st))

  a_safe_9(st):bool =
    rpc(st) = SI IMPLIES
      ((last(msg(st))=(ahead(st)=last)) AND (first(msg(st))=afirst(st)))

  I_spc_1  : LEMMA I IMPLIES a_spc_1

  I_safe_4 : LEMMA I IMPLIES a_safe_4

  I_safe_7 : LEMMA I IMPLIES a_safe_7

  I_safe_8 : LEMMA I IMPLIES a_safe_8

  I_safe_9 : LEMMA I IMPLIES a_safe_9

END protocol_lemmas


ctl[State:TYPE] : THEORY

BEGIN

  IMPORTING MU@mucalculus

  st,st0 : VAR State
  Q,i,p : VAR pred[State]
  n : VAR pred[[State,State]]

  reachable(i,n):pred[State] =
    mu(LAMBDA Q:
         LAMBDA st:
           i(st) OR
           EXISTS st0:
```

```
              Q(st0) AND n(st0,st))

  AG(i,n)(p):bool =
    FORALL st: reachable(i,n)(st) IMPLIES p(st)

END ctl


abstract_protocol : THEORY

BEGIN

  IMPORTING ctl

  File : TYPE = {NONE,ONE,MANY}

  Msg  : TYPE = [# first: bool, last: bool, toggle: bool #]

  Spc  : TYPE = {WR, SF, WA, SC, WT2}
  Rpc  : TYPE = {WF, SI, SA, RTS, NOK}

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

  tail_MANY(f:File):File =
    CASES f OF
      NONE : NONE,
      ONE  : NONE,
      MANY : MANY
    ENDCASES

  tail_ONE(f:File):File =
    CASES f OF
      NONE : NONE,
      ONE  : NONE,
      MANY : ONE
    ENDCASES

  State : TYPE =
```

```
  [# aerror : bool,
     afirst : bool,
     afile : bool,
     abusy : bool,
     data_is_safe : bool,
     rtimer_enabled : bool,
     rtimer_on : bool,
     msg : Msg,
     ctoggle : bool,
     rtoggle : bool,
     rfirst : bool,
     rpc : Rpc,
     ind_error : bool,
     ind_full : bool,
     ind_indication : IndT,
     stimer2_enabled : bool,
     stimer2_on : bool,
     stimer1_enabled : bool,
     stimer1_on : bool,
     rn : bool,
     file : File,
     stoggle : bool,
     sfirst : bool,
     spc : Spc,
     conf_full : bool,
     conf : Conf,
     req_full : bool,
     req : File,
     L : bool,
     K_full : bool,
     K : Msg,
     SAFE : bool #]

Action : DATATYPE
  BEGIN
    REQ:REQ?
    IND(ind:IndT):IND?
    IND_ERR:IND_ERR?
    CONF(conf:Conf):CONF?
```

```
  END Action

automaton(st:State,action:Action):State =
  CASES action OF
    REQ:
      LET
        st = st WITH [SAFE := NOT abusy(st)],
        st = st WITH [abusy := TRUE],
        st = st WITH [afile := TRUE]
      IN st,
    IND(i):
      LET
        st = st WITH [SAFE :=
          abusy(st) AND NOT aerror(st) AND
          afile(st) AND data_is_safe(st) AND
          (i=FIRST IMPLIES afirst(st)) AND
          (i=INCOMPLETE IMPLIES NOT afirst(st))],
        st = st WITH [afirst := i=LAST],
        st = st WITH [afile  := i/=LAST]
      IN st,
    IND_ERR:
      LET
        st = st WITH [SAFE := aerror(st)],
        st = st WITH [afirst := TRUE],
        st = st WITH [aerror := FALSE]
      IN st,
    CONF(c):
      LET
        st = st WITH [SAFE :=
          abusy(st) AND  NOT aerror(st) AND
          (c = OK IMPLIES NOT afile(st)) AND
          (c = NOT_OK IMPLIES afile(st))],
        st = st WITH [abusy := FALSE],
        st = st WITH [aerror := NOT afirst(st)],
        st = st WITH [afile := FALSE]
      IN st
  ENDCASES

initial(st:State):bool =
```

```
    SAFE(st)             = TRUE
  & K(st)                = (# first  := FALSE,
                              last   := FALSE,
                              toggle := FALSE #)
  & K_full(st)           = FALSE
  & L(st)                = FALSE
  & (ONE?(req(st)) OR MANY?(req(st)))
  & req_full(st)         = FALSE
  & conf(st)             = OK
  & conf_full(st)        = FALSE
  & spc(st)              = WR
  & sfirst(st)           = TRUE
  & stoggle(st)          = FALSE
  & file(st)             = NONE
  & rn(st)               = FALSE
  & stimer1_on(st)       = FALSE
  & stimer1_enabled(st) = FALSE
  & stimer2_on(st)       = FALSE
  & stimer2_enabled(st) = FALSE
  & ind_indication(st)  = FIRST
  & ind_full(st)         = FALSE
  & ind_error(st)        = FALSE
  & rpc(st)              = WF
  & rfirst(st)           = TRUE
  & rtoggle(st)          = FALSE
  & ctoggle(st)          = FALSE
  & msg(st)              = (# first  := FALSE,
                              last   := FALSE,
                              toggle := FALSE #)
  & rtimer_on(st)        = TRUE
  & rtimer_enabled(st)  = FALSE
  & data_is_safe(st)    = TRUE
  & abusy(st)            = FALSE
  & afile(st)            = FALSE
  & afirst(st)           = TRUE
  & aerror(st)           = FALSE


Rule_write_req(st:State,f:File):State =
```

```
  IF NOT req_full(st) AND f/=NONE THEN
    LET
      st = st WITH [req_full := TRUE],
      st = st WITH [req := f]
    IN st
  ELSE
    st
  ENDIF

Rule_read_conf(st:State):State =
  IF conf_full(st) THEN
    LET st = st WITH [conf_full := FALSE] IN st
  ELSE
    st
  ENDIF

Rule_read_ind(st:State):State =
  IF ind_full(st) THEN
    LET st = st WITH [ind_full := FALSE] IN st
  ELSE
    st
  ENDIF

Rule_read_ind_error(st:State):State =
  IF ind_error(st) THEN
    LET st = st WITH [ind_error := FALSE] IN st
  ELSE
    st
  ENDIF

Rule_lose_msg(st:State):State =
  IF K_full(st) THEN
    LET
      st = st WITH [K_full := FALSE],
      st = st WITH [stimer1_enabled := TRUE]
    IN st
  ELSE
    st
  ENDIF
```

```
Rule_lose_ack(st:State):State =
  IF L(st) THEN
    LET
      st = st WITH [L := FALSE],
      st = st WITH [stimer1_enabled := TRUE]
    IN st
  ELSE
    st
  ENDIF

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 [spc := SF],
      st = automaton(st,REQ)
    IN st
  ELSE
    st
  ENDIF

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) := file(st)=ONE],
      st = st WITH [(K)(toggle) := stoggle(st)],
      st = st WITH [spc := WA],
      st = st WITH [rn := TRUE],
      st = st WITH [stimer1_on := TRUE]
    IN st
  ELSE
    st
  ENDIF

Rule_read_L_MANY(st:State):State =
```

```
  IF spc(st) = WA AND L(st) THEN
    LET
      st = st WITH [L := FALSE],
      st = IF file(st)=ONE THEN
             LET st = st WITH [spc := SC] IN st
           ELSE
             LET st = st WITH [spc := SF] IN st
           ENDIF,
      st = st WITH [sfirst := file(st)=ONE],
      st = IF NOT file(st)=ONE THEN
             LET st = st WITH [rn := FALSE] IN st
           ELSE
             st
           ENDIF,
      st = st WITH [file := tail_MANY(file(st))],
      st = st WITH [stoggle := NOT stoggle(st)],
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

Rule_read_L_ONE(st:State):State =
  IF spc(st) = WA AND L(st) THEN
    LET
      st = st WITH [L := FALSE],
      st = IF file(st)=ONE THEN
             LET st = st WITH [spc := SC] IN st
           ELSE
             LET st = st WITH [spc := SF] IN st
           ENDIF,
      st = st WITH [sfirst := file(st)=ONE],
      st = IF NOT file(st)=ONE THEN
             LET st = st WITH [rn := FALSE] IN st
           ELSE
             st
           ENDIF,
      st = st WITH [file := tail_ONE(file(st))],
```

```
      st = st WITH [stoggle := NOT stoggle(st)],
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

Rule_write_conf(st:State):State =
  IF spc(st) = SC AND NOT conf_full(st) THEN
    LET
      st = st WITH [conf_full := TRUE],
      st = IF file(st)=NONE THEN
             LET st = st WITH [conf := OK] IN st
           ELSE
             LET
               st = IF file(st)=ONE AND rn(st) THEN
                      LET st = st WITH [conf := DONT_KNOW] IN st
                    ELSE
                      LET st = st WITH [conf := NOT_OK] IN st
                    ENDIF
             IN st
           ENDIF,
      st = IF file(st)=NONE THEN
             LET st = st WITH [spc := WR] IN st
           ELSE
             LET
               st = st WITH [spc := WT2],
               st = st WITH [sfirst := TRUE],
               st = st WITH [stoggle := NOT stoggle(st)],
               st = st WITH [stimer2_on := TRUE],
               st = st WITH [rtimer_enabled := TRUE]
             IN st
           ENDIF,
      st = st WITH [file := NONE],
      st = st WITH [rn := FALSE],
      st = automaton(st,CONF(conf(st)))
    IN st
  ELSE
```

```
    st
  ENDIF

Rule_stimer1_QUIT(st:State):State =
  IF stimer1_on(st) AND stimer1_enabled(st) THEN
    LET
      st = st WITH [spc := SC],
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

Rule_stimer1_RETRY(st:State):State =
  IF stimer1_on(st) AND stimer1_enabled(st) THEN
    LET
      st = st WITH [spc := SF],
      st = st WITH [stimer1_on := FALSE],
      st = st WITH [stimer1_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

Rule_stimer2(st:State):State =
  IF stimer2_on(st) AND stimer2_enabled(st) THEN
    LET
      st = st WITH [spc := WR],
      st = st WITH [stimer2_on := FALSE],
      st = st WITH [stimer2_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

Rule_read_K(st:State):State =
  IF rpc(st) = WF AND K_full(st) THEN
    LET
```

```
      st = st WITH [K_full := FALSE],
      st = st WITH [(msg)(first) := first(K(st))],
      st = st WITH [(msg)(last) := last(K(st))],
      st = st WITH [(msg)(toggle) := toggle(K(st))],
      st = IF NOT ctoggle(st) OR toggle(msg(st)) = rtoggle(st) THEN
             LET
               st = st WITH [rpc := SI],
               st = st WITH [rtimer_on := FALSE],
               st = st WITH [rtimer_enabled := FALSE]
             IN st
           ELSE
             LET st = st WITH [rpc := RTS] IN st
           ENDIF
    IN st
  ELSE
    st
  ENDIF

Rule_write_ind(st:State):State =
  IF rpc(st) = SI AND NOT ind_full(st) THEN
    LET
      st = st WITH [ind_full := TRUE],
      st = IF last(msg(st)) THEN
             LET st = st WITH [ind_indication := LAST] IN st
           ELSE
             LET
               st = IF first(msg(st)) THEN
                      LET st = st WITH [ind_indication := FIRST] IN st
                    ELSE
                      LET st = st WITH [ind_indication := INCOMPLETE] IN st
                    ENDIF
             IN st
           ENDIF,
      st = st WITH [rpc := SA],
      st = st WITH [rfirst := last(msg(st))],
      st = st WITH [ctoggle := TRUE],
      st = st WITH [rtoggle := NOT toggle(msg(st))],
      st = automaton(st,IND(ind_indication(st)))
    IN st
```

```
  ELSE
    st
  ENDIF

Rule_write_L(st:State):State =
  IF (rpc(st) = SA OR rpc(st) = RTS) AND NOT L(st) THEN
    LET
      st = st WITH [L := TRUE],
      st = IF rpc(st) = SA THEN
             LET st = st WITH [rtimer_on := TRUE] IN st
           ELSE
             st
           ENDIF,
      st = st WITH [rpc := WF]
    IN st
  ELSE
    st
  ENDIF

Rule_write_ind_error(st:State):State =
  IF rpc(st) = NOK AND NOT ind_error(st) THEN
    LET
      st = st WITH [ind_error := TRUE],
      st = st WITH [rpc := WF],
      st = st WITH [rfirst := TRUE],
      st = st WITH [rtimer_on := TRUE],
      st = st WITH [stimer2_enabled := TRUE],
      st = automaton(st,IND_ERR)
    IN st
  ELSE
    st
  ENDIF

Rule_rtimer(st:State):State =
  IF rtimer_on(st) AND rtimer_enabled(st) THEN
    LET
      st = st WITH [ctoggle := FALSE],
      st = IF NOT rfirst(st) THEN
             LET
```

```
              st = st WITH [rpc := NOK],
              st = st WITH [rtimer_on := FALSE]
            IN st
          ELSE
            LET st = st WITH [stimer2_enabled := TRUE] IN st
          ENDIF,
      st = st WITH [rtimer_enabled := FALSE]
    IN st
  ELSE
    st
  ENDIF

transition(st1,st2:State):bool =
  (EXISTS (f: File):
     st2 = Rule_write_req(st1,f))
  OR st2 = Rule_read_conf(st1)
  OR st2 = Rule_read_ind(st1)
  OR st2 = Rule_read_ind_error(st1)
  OR st2 = Rule_lose_msg(st1)
  OR st2 = Rule_lose_ack(st1)
  OR st2 = Rule_read_req(st1)
  OR st2 = Rule_write_K(st1)
  OR st2 = Rule_read_L_MANY(st1)
  OR st2 = Rule_read_L_ONE(st1)
  OR st2 = Rule_write_conf(st1)
  OR st2 = Rule_stimer1_QUIT(st1)
  OR st2 = Rule_stimer1_RETRY(st1)
  OR st2 = Rule_stimer2(st1)
  OR st2 = Rule_read_K(st1)
  OR st2 = Rule_write_ind(st1)
  OR st2 = Rule_write_L(st1)
  OR st2 = Rule_write_ind_error(st1)
  OR st2 = Rule_rtimer(st1)


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

safety : THEOREM
```

```
    AG(initial,transition)(safe)

END abstract_protocol



abstraction : THEORY

BEGIN

  A  : THEORY = abstract_protocol
  C  : THEORY = protocol

  IMPORTING protocol_lemmas

  abs_spc(spc:C.Spc):A.Spc =
    CASES spc OF
      WR  : WR,
      SF  : SF,
      WA  : WA,
      SC  : SC,
      WT2 : WT2
    ENDCASES

  abs_conf(conf:C.Conf):A.Conf =
    CASES conf OF
      OK : OK,
      NOT_OK : NOT_OK,
      DONT_KNOW : DONT_KNOW
    ENDCASES

  abs_rpc(rpc:C.Rpc):A.Rpc =
    CASES rpc OF
      WF  : WF,
      SI  : SI,
      SA  : SA,
      RTS : RTS,
      NOK : NOK
    ENDCASES
```

```
abs_ind(ind:C.IndT):A.IndT =
  CASES ind OF
    FIRST : FIRST,
    INCOMPLETE : INCOMPLETE,
    LAST : LAST
  ENDCASES

abs_file(h:C.Position):A.File =
  IF h=nil THEN
    NONE
  ELSIF h=last THEN
    ONE
  ELSE
    MANY
  ENDIF

abs(s:C.State):A.State =
  (#  SAFE            := SAFE(s),
      K               := (# first  := first(K(s)),
                            last   := last(K(s)),
                            toggle := toggle(K(s)) #),
      K_full          := K_full(s),
      L               := L(s),
      req             := IF C.last = 1 THEN ONE ELSE MANY ENDIF,
      req_full        := req_full(s),
      conf            := abs_conf(conf(s)),
      conf_full       := conf_full(s),
      spc             := abs_spc(spc(s)),
      sfirst          := sfirst(s),
      stoggle         := stoggle(s),
      file            := abs_file(head(s)),
      rn              := rn(s)/=0,
      stimer1_on      := stimer1_on(s),
      stimer1_enabled := stimer1_enabled(s),
      stimer2_on      := stimer2_on(s),
      stimer2_enabled := stimer2_enabled(s),
      ind_indication  := abs_ind(ind_indication(s)),
      ind_full        := ind_full(s),
```

```
      ind_error       := ind_error(s),
      rpc             := abs_rpc(rpc(s)),
      rfirst          := rfirst(s),
      rtoggle         := rtoggle(s),
      ctoggle         := ctoggle(s),
      msg             := (# first  := first(msg(s)),
                            last   := last(msg(s)),
                            toggle := toggle(msg(s)) #),
      rtimer_on       := rtimer_on(s),
      rtimer_enabled  := rtimer_enabled(s),
      data_is_safe    := rpc(s)=SI IMPLIES data(msg(s))=afile(s)(ahead(s)),
      abusy           := abusy(s),
      afile           := ahead(s)/=nil,
      afirst          := afirst(s),
      aerror          := aerror(s)
  #)

preserves(C:[C.State->C.State],A:[A.State->A.State]):bool =
  FORALL (s1,s2:C.State):
    (I(s1) AND I(s2) AND s2 = C(s1))
      IMPLIES
    abs(s2) = A(abs(s1))

c_preserves(p:pred[C.State])
           (C:[C.State->C.State],A:[A.State->A.State]):bool =
  FORALL (s1,s2:C.State):
    (I(s1) AND I(s2) AND p(s1) AND s2 = C(s1))
      IMPLIES
    abs(s2) = A(abs(s1))

write_req_preserves : LEMMA
  FORALL (s1,s2:C.State):
    (EXISTS (f:C.File):s2=C.Rule_write_req(s1,f))
      IMPLIES
    (EXISTS (f:A.File):abs(s2)=A.Rule_write_req(abs(s1),f))

read_conf_preserves : LEMMA
  preserves(C.Rule_read_conf,A.Rule_read_conf)
```

```
read_ind_preserves : LEMMA
  preserves(C.Rule_read_ind,A.Rule_read_ind)

read_ind_error_preserves : LEMMA
  preserves(C.Rule_read_ind_error,A.Rule_read_ind_error)

lose_msg_preserves : LEMMA
  preserves(C.Rule_lose_msg,A.Rule_lose_msg)

lose_ack_preserves : LEMMA
  preserves(C.Rule_lose_ack,A.Rule_lose_ack)

read_req_preserves : LEMMA
  preserves(C.Rule_read_req,A.Rule_read_req)

write_K_preserves : LEMMA
  preserves(C.Rule_write_K,A.Rule_write_K)

read_L_preserves1 : LEMMA
  c_preserves(LAMBDA (s:C.State):head(s)=last-1)
             (C.Rule_read_L,A.Rule_read_L_ONE)

read_L_preserves2 : LEMMA
  c_preserves(LAMBDA (s:C.State):head(s)/=last-1)
             (C.Rule_read_L,A.Rule_read_L_MANY)

write_conf_preserves : LEMMA
  preserves(C.Rule_write_conf,A.Rule_write_conf)

stimer1_preserves1 : LEMMA
  c_preserves(LAMBDA (s:C.State):rn(s)=max)
             (C.Rule_stimer1,A.Rule_stimer1_QUIT)

stimer1_preserves2 : LEMMA
  c_preserves(LAMBDA (s:C.State):rn(s)/=max)
             (C.Rule_stimer1,A.Rule_stimer1_RETRY)

stimer2_preserves : LEMMA
  preserves(C.Rule_stimer2,A.Rule_stimer2)
```

```
  read_K_preserves : LEMMA
    preserves(C.Rule_read_K,A.Rule_read_K)

  write_ind_preserves : LEMMA
    preserves(C.Rule_write_ind,A.Rule_write_ind)

  write_L_preserves : LEMMA
    preserves(C.Rule_write_L,A.Rule_write_L)

  write_ind_error_preserves : LEMMA
    preserves(C.Rule_write_ind_error,A.Rule_write_ind_error)

  rtimer_preserves : LEMMA
    preserves(C.Rule_rtimer,A.Rule_rtimer)


  init_preserved : PROPOSITION
    A.initial(abs(C.startstate))

  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))

  property_preserved : PROPOSITION
    FORALL (s:C.State): SAFE(abs(s)) IMPLIES SAFE(s)

END abstraction
```

C.2 Tactics

```
(defstep select (ns)
  (then*
    (hide ns)
```

```
    (delete +)
    (reveal 1 2 3 4 5))
  "Selects a consequent."
  "Selecting consequent")

(defstep auto-rewrites ()
  (then
    (auto-rewrite-theory "A" :always? T)
    (auto-rewrite-theory "C" :always? T)
    (auto-rewrite-theory "protocol_lemmas" :always? T)
    (auto-rewrite-theory "abstraction" :always? T))
  "Auto-rewrites relevant theories."
  "Auto-rewriting relevant theories")

(defstep rassert ()
  (let ((record-equality-fnums
 (find-all-sformnums
(s-forms (current-goal *ps*))
'*
#'(lambda (fmla)
    (and (equality? fmla)
 (recordtype?
  (find-supertype (type (args1 fmla))))))))
(rest-fnums
 (find-all-sformnums
(s-forms (current-goal *ps*))
'*
#'(lambda (fmla)
    (not
     (and (equality? fmla)
  (recordtype?
   (find-supertype (type (args1 fmla))))))))))
    (then (assert rest-fnums)
  (assert record-equality-fnums)))
"Asserts all the other formulas before asserting record-equalities
since the latter can be expensive."
"Simplifying with rewriting and decision procedures but delaying
record equalities")
```

```
(defstep grind-i ()
  (grind :exclude "I")
  "Grinds without rewriting I."
  "Grinding")
```

C.3 Proof Scripts

```
(|protocol_lemmas|
 (|I_spc_1| "" (EXPAND "IMPLIES")
  ((""" (SKOSIMP)
    ((""" (EXPAND "I")
      ((""" (EXPAND "&")
        ((""" (FLATTEN)
          ((""" (LEMMA "d_spc_1")
            ((""" (EXPAND "IMPLIES")
              ((""" (INST?)
                ((""" (BDDSIMP)
                  ((""" (DELETE -1)
                    ((""" (GRIND-I) NIL))))))))))))))))))))
 (|I_safe_4| "" (EXPAND "IMPLIES")
  ((""" (SKOSIMP)
    ((""" (EXPAND "I")
      ((""" (EXPAND "&")
        ((""" (FLATTEN)
          ((""" (LEMMA "d_safe_4")
            ((""" (EXPAND "IMPLIES")
              ((""" (INST?)
                ((""" (BDDSIMP)
                  ((""" (DELETE -1)
                    ((""" (GRIND-I) NIL))))))))))))))))))))
 (|I_safe_7| "" (EXPAND "IMPLIES")
  ((""" (SKOSIMP)
    ((""" (EXPAND "I")
      ((""" (EXPAND "&")
        ((""" (FLATTEN)
          ((""" (LEMMA "d_safe_7")
```

```
            (("" (EXPAND "IMPLIES")
              (("" (INST?)
                (("" (BDDSIMP)
                  (("" (DELETE -1)
                    (("" (GRIND-I) NIL)))))))))))))))))))))
 (|I_safe_8| "" (EXPAND "IMPLIES")
  (("" (SKOSIMP)
    (("" (EXPAND "I")
      (("" (EXPAND "&")
        (("" (FLATTEN)
          (("" (LEMMA "d_safe_8")
            (("" (EXPAND "IMPLIES")
              (("" (INST?)
                (("" (BDDSIMP)
                  (("" (DELETE -1)
                    (("" (GRIND-I) NIL)))))))))))))))))))))
 (|I_safe_9| "" (EXPAND "IMPLIES")
  (("" (SKOSIMP)
    (("" (EXPAND "I")
      (("" (EXPAND "&")
        (("" (FLATTEN)
          (("" (LEMMA "d_safe_9")
            (("" (EXPAND "IMPLIES")
              (("" (INST?)
                (("" (BDDSIMP)
                  (("" (DELETE -1)
                    (("" (GRIND-I) NIL))))))))))))))))))))))
(|ctl|)
(|abstract_protocol|
 (|safety| "" (EXPAND "AG")
  (("" (EXPAND "reachable") (("" (MODEL-CHECK) NIL))))))
(|abstraction|
 (|write_req_preserves| "" (SKOLEM!)
  (("" (FLATTEN)
    (("" (AUTO-REWRITE-THEORY "A" :ALWAYS? T)
      (("" (AUTO-REWRITE-THEORY "C" :ALWAYS? T)
        (("" (AUTO-REWRITE-THEORY "protocol_safe" :ALWAYS? T)
          (("" (AUTO-REWRITE-THEORY "abstraction" :ALWAYS? T)
            (("" (SKOLEM!)
```

```
(("" (CASE "last=1")
  (("1" (REPLACE*)
    (("1" (DELETE -2)
      (("1" (INST 1 "ONE")
        (("1" (EXPAND "Rule_write_req")
          (("1" (LIFT-IF)
            (("1" (LIFT-IF)
              (("1"
                (BDDSIMP)
                (("1" (PROPAX) NIL)
                 ("2" (RASSERT) NIL)
                 ("3" (RASSERT) NIL)
                 ("4"
                  (DELETE 1 2)
                  (("4"
                    (EXPAND "abs")
                    (("4"
                      (LIFT-IF)
                      (("4"
                        (BDDSIMP)
                        (("4"
                          (ASSERT)
                          (("4"
                            (APPLY-EXTENSIONALITY)
                            NIL)))))))))))
                 ("5" (PROPAX) NIL)
                 ("6"
                  (DELETE 2)
                  (("6" (ASSERT) NIL))))))))))))))))
   ("2" (REPLACE*)
    (("2" (TYPEPRED "last")
      (("2" (CASE "last>1")
        (("1" (DELETE -2 1)
          (("1" (INST 1 "MANY")
            (("1" (EXPAND "Rule_write_req")
              (("1"
                (LIFT-IF)
                (("1"
                  (LIFT-IF)
```

```
                              (("1"
                                (BDDSIMP)
                                (("1" (PROPAX) NIL)
                                 ("2" (RASSERT) NIL)
                                 ("3" (PROPAX) NIL)
                                 ("4" (RASSERT) NIL)
                                 ("5"
                                  (DELETE 1 2)
                                  (("5"
                                    (EXPAND "abs")
                                    (("5"
                                      (LIFT-IF)
                                      (("5"
                                        (BDDSIMP)
                                        (("1" (ASSERT) NIL)
                                         ("2"
                                          (ASSERT)
                                          (("2"
                                            (APPLY-EXTENSIONALITY
                                             3)
                                            NIL)))))))))))
                                 ("6"
                                  (DELETE 2)
                                  (("6" (ASSERT) NIL))))))))))))))
                     ("2" (ASSERT) NIL)))))))))))))))))))))))
(|read_conf_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOLEM!)
     (("" (FLATTEN)
       (("" (HIDE -1 -2)
         (("" (REPLACE*)
           (("" (DELETE -)
             (("" (EXPAND "Rule_read_conf")
               (("" (LIFT-IF)
                 (("" (LIFT-IF)
                   (("" (BDDSIMP)
                     (("1" (RASSERT) NIL) ("2" (RASSERT) NIL)
                      ("3" (RASSERT) NIL)
                      ("4" (PROPAX) NIL)))))))))))))))))))))
```

```
(|read_ind_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOLEM!)
     (("" (FLATTEN)
       (("" (HIDE -1 -2)
         (("" (REPLACE*)
           (("" (DELETE -)
             (("" (EXPAND "Rule_read_ind")
               (("" (LIFT-IF)
                 (("" (LIFT-IF)
                   (("" (BDDSIMP)
                     (("1" (ASSERT) NIL) ("2" (ASSERT) NIL)
                      ("3" (ASSERT) NIL)
                      ("4" (PROPAX) NIL))))))))))))))))))))))
(|read_ind_error_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOLEM!)
     (("" (FLATTEN)
       (("" (HIDE -1 -2)
         (("" (REPLACE*)
           (("" (DELETE -)
             (("" (EXPAND "Rule_read_ind_error")
               (("" (LIFT-IF)
                 (("" (LIFT-IF)
                   (("" (BDDSIMP)
                     (("1" (RASSERT) NIL) ("2" (RASSERT) NIL)
                      ("3" (RASSERT) NIL)
                      ("4" (PROPAX) NIL))))))))))))))))))))))
(|lose_msg_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (HIDE -1 -2)
       (("" (REPLACE*)
         (("" (DELETE -)
           (("" (EXPAND "Rule_lose_msg")
             (("" (LIFT-IF)
               (("" (LIFT-IF)
                 (("" (BDDSIMP)
                   (("1" (RASSERT) NIL) ("2" (RASSERT) NIL)
```

```
                  ("3" (RASSERT) NIL)
                  ("4" (PROPAX) NIL)))))))))))))))))))))
(|lose_ack_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (HIDE -1 -2)
       (("" (REPLACE*)
         (("" (DELETE -)
           (("" (EXPAND "Rule_lose_ack")
             (("" (LIFT-IF)
               (("" (LIFT-IF)
                 (("" (BDDSIMP)
                   (("1" (RASSERT) NIL) ("2" (RASSERT) NIL)
                    ("3" (RASSERT) NIL)
                    ("4" (PROPAX) NIL)))))))))))))))))))))
(|read_req_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (HIDE -1 -2)
       (("" (REPLACE*)
         (("" (DELETE -)
           (("" (EXPAND "Rule_read_req")
             (("" (LIFT-IF)
               (("" (LIFT-IF)
                 (("" (BDDSIMP)
                   (("1" (RASSERT)
                     (("1" (LIFT-IF)
                       (("1" (BDDSIMP)
                         (("1" (PROPAX) NIL)
                          ("2" (PROPAX) NIL)))))))
                    ("2" (RASSERT) NIL)
                    ("3" (RASSERT)
                     (("3" (DELETE 2) (("3" (GRIND-I) NIL)))))
                    ("4" (RASSERT) NIL) ("5" (PROPAX) NIL)
                    ("6" (PROPAX) NIL)
                    ("7" (RASSERT)
                     (("7" (DELETE 2) (("7" (GRIND-I) NIL)))))
                    ("8" (PROPAX) NIL)
                    ("9" (PROPAX) NIL)))))))))))))))))))))
```

```
(|write_K_preserves| "" (AUTO-REWRITES)
 ((""(EXPAND "preserves")
   ((""(SKOSIMP)
     ((""(HIDE -1 -2)
       ((""(REPLACE*)
         ((""(DELETE -)
           ((""(EXPAND "Rule_write_K")
             ((""(LIFT-IF)
               ((""(LIFT-IF)
                 ((""(BDDSIMP)
                   (("1" (PROPAX) NIL) ("2" (RASSERT) NIL)
                    ("3" (PROPAX) NIL) ("4" (RASSERT) NIL)
                    ("5" (DO-REWRITE)
                     (("5" (SIMPLIFY)
                       (("5" (APPLY-EXTENSIONALITY)
                         (("1" (LIFT-IF 1)
                           (("1"
                             (HIDE 4)
                             (("1"
                               (SIMPLIFY 1)
                               (("1" (PROPAX) NIL)))))))
                          ("2" (DELETE 4)
                           (("2"
                             (LIFT-IF)
                             (("2"
                               (TYPEPRED "protocol.max")
                               (("2"
                                 (BDDSIMP)
                                 (("1" (ASSERT) NIL)
                                  ("2" (ASSERT) NIL))))))))))))))
                    ("6" (DELETE 3)
                     (("6" (RASSERT) (("6" (GRIND-I) NIL)))))
                    ("7" (PROPAX) NIL)
                    ("8" (DELETE 3) (("8" (GRIND-I) NIL)))
                    ("9" (PROPAX) NIL))))))))))))))))))))
(|read_L_preserves1| "" (AUTO-REWRITES)
 ((""(EXPAND "c_preserves")
   ((""(SKOSIMP)
     ((""(HIDE -1 -2)
```

```
      (("" (REPLACE*)
        (("" (DELETE -2)
          (("" (EXPAND "Rule_read_L")
            (("" (LIFT-IF)
              (("" (BDDSIMP)
                (("1" (EXPAND "Rule_read_L_ONE")
                  (("1" (LIFT-IF)
                    (("1" (BDDSIMP)
                      (("1" (ASSERT -) NIL) ("2" (ASSERT -) NIL)
                       ("3" (ASSERT -) NIL)
                       ("4" (ASSERT -) NIL)))))))
                 ("2" (EXPAND "Rule_read_L_ONE")
                  (("2" (LIFT-IF)
                    (("2" (BDDSIMP)
                      (("1" (HIDE 2) (("1" (ASSERT) NIL)))
                       ("2" (ASSERT)
                        (("2" (APPLY-EXTENSIONALITY 3) NIL)))
                       ("3" (RASSERT) NIL)
                       ("4" (RASSERT) NIL)))))))
                 ("3" (EXPAND "Rule_read_L_ONE")
                  (("3" (APPLY (LIFT-IF))
                    (("3" (BDDSIMP)
                      (("1" (RASSERT) NIL) ("2" (RASSERT) NIL)
                       ("3" (PROPAX) NIL) ("4" (PROPAX) NIL)))))))
                 ("4" (EXPAND "Rule_read_L_ONE")
                  (("4" (APPLY (LIFT-IF))
                    (("4" (BDDSIMP)
                      (("1" (RASSERT) NIL)
                       ("2" (DELETE 3)
                        (("2" (ASSERT) (("2" (GRIND-I) NIL)))))
                       ("3" (PROPAX) NIL)
                       ("4" (PROPAX) NIL)))))))))))))))))))))))))
(|read_L_preserves2| "" (AUTO-REWRITES)
 (("" (EXPAND "c_preserves")
   (("" (SKOSIMP)
     (("" (HIDE -1 -2)
       (("" (REPLACE*)
         (("" (HIDE -1)
           (("" (EXPAND "Rule_read_L")
```

```
(("" (LIFT-IF)
  (("" (BDDSIMP)
    (("1" (EXPAND "Rule_read_L_MANY")
      (("1" (LIFT-IF)
        (("1" (BDDSIMP)
          (("1" (RASSERT)
            (("1" (APPLY-EXTENSIONALITY 2) NIL)))
           ("2" (RASSERT) NIL) ("3" (RASSERT) NIL)
           ("4" (RASSERT) NIL)))))))
     ("2" (EXPAND "Rule_read_L_MANY")
      (("2" (LIFT-IF)
        (("2" (BDDSIMP)
          (("1" (DELETE 3) (("1" (GRIND-I) NIL)))
           ("2" (RASSERT)
            (("2" (HIDE 3)
              (("2"
                (LIFT-IF)
                (("2"
                  (BDDSIMP)
                  (("1"
                    (APPLY-EXTENSIONALITY 3)
                    (("1"
                      (DELETE 4)
                      (("1"
                        (TYPEPRED "head(s1!1)")
                        (("1"
                          (LIFT-IF)
                          (("1"
                            (BDDSIMP)
                            (("1" (PROPAX) NIL)
                             ("2"
                              (ASSERT)
                              NIL)))))))))))
                   ("2"
                    (APPLY-EXTENSIONALITY 4)
                    (("2"
                      (DELETE 5)
                      (("2"
                        (LIFT-IF)
```

```
                                   (("2"
                                     (BDDSIMP)
                                     (("1" (ASSERT) NIL)
                                      ("2" (ASSERT) NIL)
                                      ("3"
                                       (ASSERT)
                                       NIL)))))))))))))))))
                        ("3" (RASSERT) NIL)
                        ("4" (RASSERT) NIL)))))))
                  ("3" (EXPAND "Rule_read_L_MANY")
                   (("3" (LIFT-IF) (("3" (RASSERT) NIL)))))
                  ("4" (EXPAND "Rule_read_L_MANY")
                   (("4" (LIFT-IF)
                     (("4" (BDDSIMP)
                       (("1" (DELETE 3)
                         (("1" (DELETE -2 -3 1)
                           (("1" (GRIND-I) NIL)))))
                        ("2" (RASSERT)
                         (("2" (LIFT-IF -)
                           (("2"
                             (DELETE 3 4)
                             (("2"
                               (BDDSIMP)
                               (("1" (ASSERT) NIL)
                                ("2" (ASSERT) NIL)
                                ("3" (ASSERT) NIL)
                                ("4" (PROPAX) NIL)))))))))
                        ("3" (PROPAX) NIL)
                        ("4" (PROPAX) NIL)))))))))))))))))))))))))
(|write_conf_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (REPLACE -3 1)
       (("" (HIDE -1 -2 -3)
         (("" (EXPAND "Rule_write_conf" 1 1)
           (("" (EXPAND "automaton")
             (("" (LIFT-IF)
               (("" (BDDSIMP)
                 (("1" (EXPAND "Rule_write_conf")
```

```
 (("1" (EXPAND "automaton")
   (("1" (LIFT-IF)
     (("1" (BDDSIMP)
       (("1" (PROPAX) NIL) ("2" (RASSERT) NIL)
        ("3" (RASSERT) NIL) ("4" (RASSERT) NIL)
        ("5" (RASSERT) NIL)
        ("6" (PROPAX) NIL)))))))))
("2" (EXPAND "Rule_write_conf")
 (("2" (EXPAND "automaton")
   (("2" (LIFT-IF)
     (("2" (BDDSIMP)
       (("1" (RASSERT) NIL)
        ("2" (RASSERT)
         (("2"
           (DO-REWRITE)
           (("2"
             (EXPAND "/=")
             (("2"
               (APPLY-EXTENSIONALITY)
               (("2"
                 (DELETE 3)
                 (("2"
                   (BDDSIMP)
                   (("1"
                     (LEMMA "I_spc_1")
                     (("1"
                       (EXPAND "IMPLIES")
                       (("1"
                         (INST?)
                         (("1"
                           (REVEAL -3)
                           (("1"
                             (BDDSIMP)
                             (("1"
                               (ASSERT)
                               NIL)))))))))))
                    ("2"
                     (LEMMA "I_spc_1")
                     (("2"
```

```
                       (EXPAND "IMPLIES")
                       (("2"
                         (INST?)
                         (("2"
                           (REVEAL -3)
                           (("2"
                             (BDDSIMP)
                             (("2"
                               (ASSERT)
                               NIL)))))))))))))))))))))))
        ("3" (DELETE 4)
         (("3"
           (EXPAND "abs" +)
           (("3"
             (EXPAND "abs_file")
             (("3"
               (BDDSIMP)
               (("3"
                 (LIFT-IF)
                 (("3"
                   (BDDSIMP)
                   (("3" (PROPAX) NIL)))))))))))))
        ("4" (DELETE 5)
         (("4"
           (EXPAND "abs" +)
           (("4"
             (EXPAND "abs_file")
             (("4"
               (LIFT-IF)
               (("4"
                 (BDDSIMP)
                 (("4" (PROPAX) NIL)))))))))))
        ("5" (DELETE 5) (("5" (GRIND-I) NIL)))
        ("6" (DELETE 3)
         (("6" (GRIND-I) NIL)))))))))))
("3" (EXPAND "Rule_write_conf")
 (("3" (EXPAND "automaton")
   (("3" (LIFT-IF)
     (("3" (BDDSIMP)
```

```
(("1" (RASSERT) NIL) ("2" (RASSERT) NIL)
 ("3" (RASSERT)
  (("3"
    (LEMMA "I_safe_4")
    (("3"
      (REVEAL -1)
      (("3"
        (EXPAND "IMPLIES")
        (("3"
          (INST?)
          (("3"
            (BDDSIMP)
            (("3"
              (DO-REWRITE)
              (("3"
                (APPLY-EXTENSIONALITY 5)
                (("1"
                  (DELETE 6)
                  (("1"
                    (MUSIMP)
                    (("1" (PROPAX) NIL)))))
                 ("2"
                  (DELETE 6)
                  (("2"
                    (LEMMA "I_spc_1")
                    (("2"
                      (EXPAND "IMPLIES")
                      (("2"
                        (INST?)
                        (("2"
                          (BDDSIMP (-1 -2))
                          (("2"
                            (EXPAND
                             "a_spc_1")
                            (("2"
                              (MUSIMP)
                              (("2"
                                (PROPAX)
                                NIL)
```

```
              ))))))))))))))))))))))))))))))
        ("4" (RASSERT) NIL) ("5" (RASSERT) NIL)
        ("6" (RASSERT) NIL)))))))))
("4" (EXPAND "Rule_write_conf")
 (("4" (EXPAND "automaton")
   (("4" (LIFT-IF)
     (("4" (BDDSIMP)
       (("1" (RASSERT) NIL)
        ("2" (RASSERT)
         (("2"
           (DELETE 3)
           (("2"
             (LIFT-IF)
             (("2"
               (BDDSIMP)
               (("1" (ASSERT) NIL)
                ("2" (ASSERT) NIL)))))))))
        ("3" (RASSERT) NIL)
        ("4" (RASSERT)
         (("4"
           (DO-REWRITE)
           (("4"
             (APPLY-EXTENSIONALITY 4)
             (("4"
               (MUSIMP)
               (("4" (PROPAX) NIL)))))))))
        ("5" (RASSERT)
         (("5"
           (DO-REWRITE)
           (("5"
             (APPLY-EXTENSIONALITY 5)
             (("5"
               (BDDSIMP)
               (("1"
                 (DELETE 4)
                 (("1"
                   (LEMMA "I_spc_1")
                   (("1"
                     (EXPAND "IMPLIES")
```

```
                        (("1"
                          (INST?)
                          (("1"
                            (REVEAL -3)
                            (("1"
                              (BDDSIMP)
                              (("1"
                                (ASSERT)
                                NIL))))))))))))
                  ("2"
                   (DELETE 4)
                   (("2"
                     (LEMMA "I_spc_1")
                     (("2"
                       (EXPAND "IMPLIES")
                       (("2"
                         (INST?)
                         (("2"
                           (REVEAL -3)
                           (("2"
                             (BDDSIMP)
                             (("2"
                               (ASSERT)
                               NIL))))))))))))))))))))
          ("6" (RASSERT) NIL))))))))
 ("5" (EXPAND "Rule_write_conf")
  (("5" (EXPAND "automaton")
    (("5" (LIFT-IF)
      (("5" (BDDSIMP)
        (("1" (RASSERT) NIL) ("2" (RASSERT) NIL)
         ("3" (RASSERT) NIL)
         ("4" (RASSERT)
          (("4"
            (DO-REWRITE)
            (("4"
              (APPLY-EXTENSIONALITY 6)
              (("4"
                (DELETE 7)
                (("4"
```

```
        (BDDSIMP)
        (("1"
          (LEMMA "I_spc_1")
          (("1"
            (EXPAND "IMPLIES")
            (("1"
              (INST?)
              (("1"
                (REVEAL -3)
                (("1"
                  (BDDSIMP)
                  (("1"
                    (ASSERT)
                    NIL)))))))))))
         ("2"
          (LEMMA "I_spc_1")
          (("2"
            (EXPAND "IMPLIES")
            (("2"
              (INST?)
              (("2"
                (REVEAL -3)
                (("2"
                  (BDDSIMP)
                  (("2"
                    (ASSERT)
                    NIL))))))))))))))))))))))
("5" (RASSERT)
 (("5"
   (DO-REWRITE)
   (("5"
     (APPLY-EXTENSIONALITY 6)
     (("5"
       (DELETE 7)
       (("5"
         (BDDSIMP)
         (("1"
           (LEMMA "I_spc_1")
           (("1"
```

```
                     (EXPAND "IMPLIES")
                     (("1"
                       (INST?)
                       (("1"
                         (REVEAL -3)
                         (("1"
                           (BDDSIMP)
                           (("1"
                             (ASSERT)
                             NIL)))))))))))
                  ("2"
                   (LEMMA "I_spc_1")
                   (("2"
                     (EXPAND "IMPLIES")
                     (("2"
                       (INST?)
                       (("2"
                         (REVEAL -3)
                         (("2"
                           (BDDSIMP)
                           (("2"
                             (ASSERT)
                             NIL))))))))))))))))))))))
        ("6" (RASSERT) NIL)))))))))
("6" (EXPAND "Rule_write_conf")
 (("6" (LIFT-IF)
   (("6" (BDDSIMP)
     (("1" (PROPAX) NIL)
      ("2" (RASSERT)
       (("2" (LIFT-IF -1)
         (("2"
           (BDDSIMP -1)
           (("1" (DELETE 3) (("1" (ASSERT) NIL)))
            ("2" (ASSERT) NIL)
            ("3" (ASSERT) NIL)
            ("4" (ASSERT) NIL)
            ("5" (PROPAX) NIL)))))))
      ("3" (DELETE 4)
       (("3" (RASSERT)
```

```
                        (("3"
                          (LIFT-IF -1)
                          (("3"
                            (BDDSIMP -1)
                            (("1" (ASSERT) NIL)
                             ("2" (ASSERT) NIL)
                             ("3" (ASSERT) NIL)
                             ("4" (ASSERT) NIL)
                             ("5" (PROPAX) NIL))))))))
                     ("4" (DELETE 5)
                      (("4" (EXPAND "abs" -1)
                        (("4"
                          (EXPAND "abs_spc")
                          (("4"
                            (MUSIMP)
                            (("4" (PROPAX) NIL))))))))
                     ("5" (DELETE 5)
                      (("5" (EXPAND "abs" -1)
                        (("5"
                          (EXPAND "abs_spc")
                          (("5"
                            (MUSIMP)
                            (("5" (PROPAX) NIL))))))))
                     ("6" (PROPAX) NIL)))))))))))))))))))))))
(|stimer1_preserves1| "" (AUTO-REWRITES)
 (("" (EXPAND "c_preserves")
   (("" (SKOSIMP)
     (("" (REPLACE -4 1)
       (("" (HIDE -1 -2 -4)
         (("" (EXPAND "Rule_stimer1")
           (("" (LIFT-IF)
             (("" (BDDSIMP)
               (("1" (EXPAND "Rule_stimer1_QUIT")
                 (("1" (LIFT-IF)
                   (("1" (BDDSIMP)
                     (("1" (RASSERT)
                       (("1" (APPLY-EXTENSIONALITY) NIL)))
                      ("2" (RASSERT) NIL) ("3" (RASSERT) NIL))))))
                ("2" (EXPAND "Rule_stimer1_QUIT")
```

```
                ("2" (LIFT-IF)
                  (("2" (BDDSIMP)
                    (("1" (RASSERT) NIL) ("2" (PROPAX) NIL)
                     ("3" (PROPAX) NIL)))))))
               ("3" (EXPAND "Rule_stimer1_QUIT")
                (("3" (LIFT-IF)
                  (("3" (BDDSIMP)
                    (("1" (RASSERT) NIL) ("2" (PROPAX) NIL)
                     ("3" (PROPAX) NIL)))))))))))))))))))))))
(|stimer1_preserves2| "" (AUTO-REWRITES)
 (("" (EXPAND "c_preserves")
   (("" (SKOSIMP)
     (("" (REPLACE -3 2)
       (("" (HIDE -1 -2 -3)
         (("" (EXPAND "Rule_stimer1")
           (("" (EXPAND "Rule_stimer1_RETRY")
             (("" (LIFT-IF)
               (("" (LIFT-IF)
                 (("" (BDDSIMP)
                   (("1" (RASSERT)
                     (("1" (APPLY-EXTENSIONALITY 2) NIL)))
                    ("2" (RASSERT) NIL) ("3" (RASSERT) NIL)
                    ("4" (RASSERT) NIL) ("5" (PROPAX) NIL)
                    ("6" (PROPAX) NIL) ("7" (RASSERT) NIL)
                    ("8" (PROPAX) NIL)
                    ("9" (PROPAX) NIL))))))))))))))))))))
(|stimer2_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (REPLACE -3 1)
       (("" (HIDE -1 -2 -3)
         (("" (EXPAND "Rule_stimer2")
           (("" (LIFT-IF)
             (("" (LIFT-IF)
               (("" (BDDSIMP)
                 (("1" (RASSERT) (("1" (APPLY-EXTENSIONALITY) NIL)))
                  ("2" (RASSERT) NIL) ("3" (RASSERT) NIL)
                  ("4" (RASSERT) NIL) ("5" (PROPAX) NIL)
                  ("6" (PROPAX) NIL) ("7" (RASSERT) NIL)
```

```
                  ("8" (PROPAX) NIL)
                  ("9" (PROPAX) NIL))))))))))))))))))
(|read_K_preserves| "" (AUTO-REWRITES)
 ((""  (EXPAND "preserves")
   ((""  (SKOSIMP)
     ((""  (LEMMA "I_safe_8")
       ((""  (EXPAND "IMPLIES")
         ((""  (INST -1 "s2!1")
           ((""  (BDDSIMP (-1 -3))
             ((""  (EXPAND "a_safe_8")
               ((""  (EXPAND "Rule_read_K" -4)
                 ((""  (LIFT-IF)
                   ((""  (BDDSIMP -4)
                     (("1" (EXPAND "Rule_read_K")
                       (("1" (ASSERT)
                         (("1" (APPLY-EXTENSIONALITY)
                           (("1" (DELETE 2) (("1" (GRIND-I) NIL)))
                            ("2" (DELETE 2) (("2" (GRIND-I) NIL)))
                            ("3" (DELETE 2) (("3" (GRIND-I) NIL)))
                            ("4" (DELETE 2) (("4" (GRIND-I) NIL)))
                            ("5" (DELETE 2) (("5" (GRIND-I) NIL)))
                            ("6" (DELETE 2) (("6" (GRIND-I) NIL)))
                            ("7"
                             (DELETE 2)
                             (("7" (GRIND-I) NIL))))))))))
                      ("2" (EXPAND "Rule_read_K")
                       (("2" (ASSERT)
                         (("2" (APPLY-EXTENSIONALITY)
                           (("1" (DELETE 3) (("1" (GRIND-I) NIL)))
                            ("2" (DELETE 3) (("2" (GRIND-I) NIL)))
                            ("3" (DELETE 3) (("3" (GRIND-I) NIL)))
                            ("4" (DELETE 3) (("4" (GRIND-I) NIL)))
                            ("5" (DELETE 3) (("5" (GRIND-I) NIL)))
                            ("6" (DELETE 3) (("6" (GRIND-I) NIL)))
                            ("7"
                             (DELETE 3)
                             (("7" (GRIND-I) NIL))))))))))
                      ("3" (EXPAND "Rule_read_K")
                       (("3" (ASSERT)
```

```
  (("3" (APPLY-EXTENSIONALITY)
    (("1" (DELETE 3) (("1" (GRIND-I) NIL)))
     ("2" (DELETE 3) (("2" (GRIND-I) NIL)))
     ("3" (DELETE 3) (("3" (GRIND-I) NIL)))
     ("4" (DELETE 3) (("4" (GRIND-I) NIL)))
     ("5" (DELETE 3) (("5" (GRIND-I) NIL)))
     ("6" (DELETE 3) (("6" (GRIND-I) NIL)))
     ("7"
      (DELETE 3)
      (("7" (GRIND-I) NIL))))))))
("4" (EXPAND "Rule_read_K")
 (("4" (ASSERT)
   (("4" (APPLY-EXTENSIONALITY)
     (("1" (DELETE 4) (("1" (GRIND-I) NIL)))
      ("2" (DELETE 4) (("2" (GRIND-I) NIL)))
      ("3" (DELETE 4) (("3" (GRIND-I) NIL)))
      ("4" (DELETE 4) (("4" (GRIND-I) NIL)))
      ("5" (DELETE 4) (("5" (GRIND-I) NIL)))
      ("6" (DELETE 4) (("6" (GRIND-I) NIL)))
      ("7"
       (DELETE 4)
       (("7" (GRIND-I) NIL))))))))
("5" (EXPAND "Rule_read_K")
 (("5" (ASSERT)
   (("5" (APPLY-EXTENSIONALITY)
     (("1" (DELETE 3) (("1" (GRIND-I) NIL)))
      ("2" (DELETE 3) (("2" (GRIND-I) NIL)))
      ("3" (DELETE 3) (("3" (GRIND-I) NIL)))
      ("4" (DELETE 3) (("4" (GRIND-I) NIL)))
      ("5" (DELETE 3) (("5" (GRIND-I) NIL)))
      ("6" (DELETE 3) (("6" (GRIND-I) NIL)))
      ("7"
       (DELETE 3)
       (("7" (GRIND-I) NIL))))))))
("6" (EXPAND "Rule_read_K")
 (("6" (ASSERT) (("6" (GRIND-I) NIL)))))
("7" (EXPAND "Rule_read_K")
 (("7" (EXPAND "abs" 2 2)
   (("7" (EXPAND "abs_rpc")
```

```
                           (("7"
                             (SIMPLIFY)
                             (("7"
                               (LIFT-IF)
                               (("7"
                                 (SIMPLIFY)
                                 (("7"
                                   (RECORD)
                                   (("7"
                                     (SIMPLIFY)
                                     (("7"
                                       (PROPAX)
                                       NIL)
          ))))))))))))))))))))))))))))))))))))))
(|write_ind_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (LEMMA "I_safe_7")
       (("" (LEMMA "I_safe_8")
         (("" (LEMMA "I_safe_9")
           (("" (EXPAND "IMPLIES")
             (("" (INST?)
               (("" (INST?)
                 (("" (INST?)
                   (("" (BDDSIMP -)
                     (("" (EXPAND "a_safe_7")
                       (("" (EXPAND "a_safe_8")
                         (("" (EXPAND "a_safe_9")
                           ((""
                             (EXPAND "Rule_write_ind" -6)
                             ((""
                               (EXPAND "automaton")
                               ((""
                                 (LIFT-IF)
                                 ((""
                                   (BDDSIMP)
                                   (("1"
                                     (EXPAND "Rule_write_ind")
                                     (("1"
```

```
      (EXPAND "automaton")
      (("1"
        (ASSERT)
        (("1" (GRIND-I) NIL)))))))
   ("2"
    (EXPAND "Rule_write_ind")
    (("2"
      (EXPAND "automaton")
      (("2"
        (ASSERT)
        (("2"
          (APPLY-EXTENSIONALITY 3)
          (("1"
            (DELETE 4)
            (("1"
              (REPLACE*)
              (("1"
                (ASSERT)
                NIL)))))
           ("2"
            (DELETE 4)
            (("2" (GRIND-I) NIL)))
           ("3"
            (DELETE 4)
            (("3" (GRIND-I) NIL)))
           ("4"
            (DELETE 4)
            (("4" (GRIND-I) NIL)))
           ("5"
            (DELETE 4)
            (("5" (GRIND-I) NIL)))
           ("6"
            (DELETE 4)
            (("6" (GRIND-I) NIL)))
           ("7"
            (DELETE 4)
            (("7" (GRIND-I) NIL)))
           ("8"
            (DELETE 4)
```

```
 (("8" (GRIND-I) NIL)))
("9"
 (DELETE 4)
 (("9" (GRIND-I) NIL)))
("10"
 (DELETE 4)
 (("10" (GRIND-I) NIL)))
("11"
 (DELETE 4)
 (("11" (GRIND-I) NIL)))
("12"
 (DELETE 4)
 (("12" (GRIND-I) NIL)))
("13"
 (DELETE 4)
 (("13" (GRIND-I) NIL)))
("14"
 (DELETE 4)
 (("14" (GRIND-I) NIL)))
("15"
 (DELETE 4)
 (("15" (GRIND-I) NIL)))
("16"
 (DELETE 4)
 (("16" (GRIND-I) NIL)))
("17"
 (DELETE 4)
 (("17" (GRIND-I) NIL)))
("18"
 (DELETE 4)
 (("18" (GRIND-I) NIL)))
("19"
 (DELETE 4)
 (("19" (GRIND-I) NIL)))
("20"
 (DELETE 4)
 (("20" (GRIND-I) NIL)))
("21"
 (DELETE 4)
```

```
            (("21" (GRIND-I) NIL)))
           ("22"
            (DELETE 4)
            (("22" (GRIND-I) NIL)))
           ("23"
            (DELETE 4)
            (("23" (GRIND-I) NIL)))
           ("24"
            (DELETE 4)
            (("24" (GRIND-I) NIL)))
           ("25"
            (DELETE 4)
            (("25" (GRIND-I) NIL)))
           ("26"
            (DELETE 4)
            (("26" (GRIND-I) NIL)))
           ("27"
            (DELETE 4)
            (("27" (GRIND-I) NIL)))
           ("28"
            (DELETE 4)
            (("28" (GRIND-I) NIL)))
           ("29"
            (DELETE 4)
            (("29" (GRIND-I) NIL)))
           ("30"
            (DELETE 4)
            (("30" (GRIND-I) NIL)))
           ("31"
            (DELETE 4)
            (("31"
              (GRIND-I)
              NIL)))))))))))
  ("3"
   (EXPAND "Rule_write_ind")
   (("3"
     (EXPAND "automaton")
     (("3"
       (ASSERT)
```

```
    (("3" (GRIND-I) NIL)))))))
("4"
 (EXPAND "Rule_write_ind")
 (("4"
   (EXPAND "automaton")
   (("4"
     (ASSERT)
     (("4"
       (APPLY-EXTENSIONALITY 5)
       (("1"
         (DELETE 6)
         (("1" (GRIND-I) NIL)))
        ("2"
         (DELETE 6)
         (("2" (GRIND-I) NIL)))
        ("3"
         (DELETE 6)
         (("3" (GRIND-I) NIL)))
        ("4"
         (DELETE 6)
         (("4" (GRIND-I) NIL)))
        ("5"
         (DELETE 6)
         (("5" (GRIND-I) NIL)))
        ("6"
         (DELETE 6)
         (("6" (GRIND-I) NIL)))
        ("7"
         (DELETE 6)
         (("7" (GRIND-I) NIL)))
        ("8"
         (DELETE 6)
         (("8" (GRIND-I) NIL)))
        ("9"
         (DELETE 6)
         (("9" (GRIND-I) NIL)))
        ("10"
         (DELETE 6)
         (("10" (GRIND-I) NIL)))
```

```
("11"
 (DELETE 6)
 (("11" (GRIND-I) NIL)))
("12"
 (DELETE 6)
 (("12" (GRIND-I) NIL)))
("13"
 (DELETE 6)
 (("13" (GRIND-I) NIL)))
("14"
 (DELETE 6)
 (("14" (GRIND-I) NIL)))
("15"
 (DELETE 6)
 (("15" (GRIND-I) NIL)))
("16"
 (DELETE 6)
 (("16" (GRIND-I) NIL)))
("17"
 (DELETE 6)
 (("17" (GRIND-I) NIL)))
("18"
 (DELETE 6)
 (("18" (GRIND-I) NIL)))
("19"
 (DELETE 6)
 (("19" (GRIND-I) NIL)))
("20"
 (DELETE 6)
 (("20" (GRIND-I) NIL)))
("21"
 (DELETE 6)
 (("21" (GRIND-I) NIL)))
("22"
 (DELETE 6)
 (("22" (GRIND-I) NIL)))
("23"
 (DELETE 6)
 (("23" (GRIND-I) NIL)))
```

```
        ("24"
         (DELETE 6)
         (("24" (GRIND-I) NIL)))
        ("25"
         (DELETE 6)
         (("25" (GRIND-I) NIL)))
        ("26"
         (DELETE 6)
         (("26" (GRIND-I) NIL)))
        ("27"
         (DELETE 6)
         (("27" (GRIND-I) NIL)))
        ("28"
         (DELETE 6)
         (("28" (GRIND-I) NIL)))
        ("29"
         (DELETE 6)
         (("29" (GRIND-I) NIL)))
        ("30"
         (DELETE 6)
         (("30" (GRIND-I) NIL)))
        ("31"
         (DELETE 6)
         (("31"
           (GRIND-I)
           NIL)))))))))))
("5"
 (EXPAND "Rule_write_ind")
 (("5"
   (EXPAND "automaton")
   (("5"
     (ASSERT)
     (("5" (GRIND-I) NIL)))))))
("6"
 (EXPAND "Rule_write_ind")
 (("6"
   (EXPAND "automaton")
   (("6"
     (ASSERT)
```

```
(("6"
  (APPLY-EXTENSIONALITY 5)
  (("1"
    (DELETE 6)
    (("1" (GRIND-I) NIL)))
   ("2"
    (DELETE 6)
    (("2" (GRIND-I) NIL)))
   ("3"
    (DELETE 6)
    (("3" (GRIND-I) NIL)))
   ("4"
    (DELETE 6)
    (("4" (GRIND-I) NIL)))
   ("5"
    (DELETE 6)
    (("5" (GRIND-I) NIL)))
   ("6"
    (DELETE 6)
    (("6" (GRIND-I) NIL)))
   ("7"
    (DELETE 6)
    (("7" (GRIND-I) NIL)))
   ("8"
    (DELETE 6)
    (("8" (GRIND-I) NIL)))
   ("9"
    (DELETE 6)
    (("9" (GRIND-I) NIL)))
   ("10"
    (DELETE 6)
    (("10" (GRIND-I) NIL)))
   ("11"
    (DELETE 6)
    (("11" (GRIND-I) NIL)))
   ("12"
    (DELETE 6)
    (("12" (GRIND-I) NIL)))
   ("13"
```

```
 (DELETE 6)
 (("13" (GRIND-I) NIL)))
("14"
 (DELETE 6)
 (("14" (GRIND-I) NIL)))
("15"
 (DELETE 6)
 (("15" (GRIND-I) NIL)))
("16"
 (DELETE 6)
 (("16" (GRIND-I) NIL)))
("17"
 (DELETE 6)
 (("17" (GRIND-I) NIL)))
("18"
 (DELETE 6)
 (("18" (GRIND-I) NIL)))
("19"
 (DELETE 6)
 (("19" (GRIND-I) NIL)))
("20"
 (DELETE 6)
 (("20" (GRIND-I) NIL)))
("21"
 (DELETE 6)
 (("21" (GRIND-I) NIL)))
("22"
 (DELETE 6)
 (("22" (GRIND-I) NIL)))
("23"
 (DELETE 6)
 (("23" (GRIND-I) NIL)))
("24"
 (DELETE 6)
 (("24" (GRIND-I) NIL)))
("25"
 (DELETE 6)
 (("25" (GRIND-I) NIL)))
("26"
```

```
        (DELETE 6)
        (("26" (GRIND-I) NIL)))
       ("27"
        (DELETE 6)
        (("27" (GRIND-I) NIL)))
       ("28"
        (DELETE 6)
        (("28" (GRIND-I) NIL)))
       ("29"
        (DELETE 6)
        (("29" (GRIND-I) NIL)))
       ("30"
        (DELETE 6)
        (("30" (GRIND-I) NIL)))
       ("31"
        (DELETE 6)
        (("31"
          (GRIND-I)
          NIL)))))))))))
("7"
 (EXPAND "Rule_write_ind")
 (("7"
   (EXPAND "automaton")
   (("7"
     (ASSERT)
     (("7" (GRIND-I) NIL)))))))
("8"
 (EXPAND "Rule_write_ind")
 (("8"
   (EXPAND "automaton")
   (("8"
     (ASSERT)
     (("8"
       (APPLY-EXTENSIONALITY 7)
       (("1"
         (DELETE 8)
         (("1" (GRIND-I) NIL)))
        ("2"
         (DELETE 8)
```

```
 (("2" (GRIND-I) NIL)))
("3"
 (DELETE 8)
 (("3" (GRIND-I) NIL)))
("4"
 (DELETE 8)
 (("4" (GRIND-I) NIL)))
("5"
 (DELETE 8)
 (("5" (GRIND-I) NIL)))
("6"
 (DELETE 8)
 (("6" (GRIND-I) NIL)))
("7"
 (DELETE 8)
 (("7" (GRIND-I) NIL)))
("8"
 (DELETE 8)
 (("8" (GRIND-I) NIL)))
("9"
 (DELETE 8)
 (("9" (GRIND-I) NIL)))
("10"
 (DELETE 8)
 (("10" (GRIND-I) NIL)))
("11"
 (DELETE 8)
 (("11" (GRIND-I) NIL)))
("12"
 (DELETE 8)
 (("12" (GRIND-I) NIL)))
("13"
 (DELETE 8)
 (("13" (GRIND-I) NIL)))
("14"
 (DELETE 8)
 (("14" (GRIND-I) NIL)))
("15"
 (DELETE 8)
```

```
 (("15" (GRIND-I) NIL)))
("16"
 (DELETE 8)
 (("16" (GRIND-I) NIL)))
("17"
 (DELETE 8)
 (("17" (GRIND-I) NIL)))
("18"
 (DELETE 8)
 (("18" (GRIND-I) NIL)))
("19"
 (DELETE 8)
 (("19" (GRIND-I) NIL)))
("20"
 (DELETE 8)
 (("20" (GRIND-I) NIL)))
("21"
 (DELETE 8)
 (("21" (GRIND-I) NIL)))
("22"
 (DELETE 8)
 (("22" (GRIND-I) NIL)))
("23"
 (DELETE 8)
 (("23" (GRIND-I) NIL)))
("24"
 (DELETE 8)
 (("24" (GRIND-I) NIL)))
("25"
 (DELETE 8)
 (("25" (GRIND-I) NIL)))
("26"
 (DELETE 8)
 (("26" (GRIND-I) NIL)))
("27"
 (DELETE 8)
 (("27" (GRIND-I) NIL)))
("28"
 (DELETE 8)
```

```
                                             (("28" (GRIND-I) NIL)))
                                            ("29"
                                             (DELETE 8)
                                             (("29" (GRIND-I) NIL)))
                                            ("30"
                                             (DELETE 8)
                                             (("30" (GRIND-I) NIL)))
                                            ("31"
                                             (DELETE 8)
                                             (("31"
                                               (GRIND-I)
                                               NIL)))))))))))
                                    ("9"
                                     (EXPAND "Rule_write_ind")
                                     (("9"
                                       (EXPAND "abs" 2 2)
                                       (("9"
                                         (EXPAND "abs_rpc")
                                         (("9"
                                           (SIMPLIFY)
                                           (("9"
                                             (LIFT-IF)
                                             (("9"
                                               (SIMPLIFY)
                                               (("9"
                                                 (RECORD)
                                                 (("9"
                                                   (SIMPLIFY)
                                                   (("9"
                                                     (PROPAX)
                                                     NIL)
              ))))))))))))))))))))))))))))))))))))))))))))))))))))
(|write_L_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (EXPAND "Rule_write_L" -3)
       (("" (LIFT-IF)
         (("" (BDDSIMP)
           (("1" (EXPAND "Rule_write_L")
```

```
             (("1" (ASSERT) (("1" (GRIND-I) NIL)))))
            ("2" (EXPAND "Rule_write_L")
             (("2" (ASSERT)
               (("2" (APPLY-EXTENSIONALITY 2)
                 (("1" (DELETE 3) (("1" (GRIND-I) NIL)))
                  ("2" (DELETE 3) (("2" (GRIND-I) NIL)))
                  ("3" (DELETE 3) (("3" (GRIND-I) NIL)))
                  ("4" (DELETE 3) (("4" (GRIND-I) NIL)))
                  ("5" (DELETE 3) (("5" (GRIND-I) NIL)))
                  ("6" (DELETE 3) (("6" (GRIND-I) NIL)))))))))
            ("3" (EXPAND "Rule_write_L")
             (("3" (ASSERT)
               (("3" (APPLY-EXTENSIONALITY 3)
                 (("1" (DELETE 4) (("1" (GRIND-I) NIL)))
                  ("2" (DELETE 4) (("2" (GRIND-I) NIL)))
                  ("3" (DELETE 4) (("3" (GRIND-I) NIL)))
                  ("4" (DELETE 4) (("4" (GRIND-I) NIL)))
                  ("5" (DELETE 3)
                   (("5" (DELETE 3) (("5" (GRIND-I) NIL)))))
                  ("6" (DELETE 4) (("6" (GRIND-I) NIL)))))))))
            ("4" (EXPAND "Rule_write_L")
             (("4" (EXPAND "abs" 3 2)
               (("4" (EXPAND "abs_rpc")
                 (("4" (SIMPLIFY)
                   (("4" (LIFT-IF)
                     (("4" (SIMPLIFY)
                       (("4" (RECORD)
                         (("4" (SIMPLIFY)
                           (("4"
                             (LIFT-IF)
                             (("4"
                               (SIMPLIFY)
                               (("4"
                                 (MUSIMP)
                                 (("4"
                                   (PROPAX)
                                   NIL)))))))))))))))))))))))))))))))))))
(|write_ind_error_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
```

```
   (("" (SKOSIMP)
     (("" (EXPAND "Rule_write_ind_error" -3)
       (("" (EXPAND "automaton")
         (("" (LIFT-IF)
           (("" (BDDSIMP)
             (("1" (EXPAND "Rule_write_ind_error")
               (("1" (EXPAND "automaton")
                 (("1" (ASSERT) (("1" (GRIND-I) NIL)))))))
              ("2" (EXPAND "Rule_write_ind_error")
               (("2" (EXPAND "automaton")
                 (("2" (ASSERT)
                   (("2" (APPLY-EXTENSIONALITY 2)
                     (("1" (DELETE 3) (("1" (GRIND-I) NIL)))
                      ("2" (DELETE 3) (("2" (GRIND-I) NIL)))
                      ("3" (DELETE 3) (("3" (GRIND-I) NIL)))
                      ("4" (DELETE 3) (("4" (GRIND-I) NIL)))
                      ("5" (DELETE 3) (("5" (GRIND-I) NIL)))
                      ("6" (DELETE 3) (("6" (GRIND-I) NIL))))))))))
              ("3" (EXPAND "Rule_write_ind_error")
               (("3" (EXPAND "automaton")
                 (("3" (EXPAND "abs" 2 2)
                   (("3" (EXPAND "abs_rpc")
                     (("3" (LIFT-IF)
                       (("3" (SIMPLIFY)
                         (("3" (RECORD)
                           (("3"
                             (SIMPLIFY)
                             (("3"
                               (PROPAX)
                               NIL)))))))))))))))))))))))))))))
(|rtimer_preserves| "" (AUTO-REWRITES)
 (("" (EXPAND "preserves")
   (("" (SKOSIMP)
     (("" (EXPAND "Rule_rtimer" -3)
       (("" (LIFT-IF)
         (("" (BDDSIMP)
           (("1" (EXPAND "Rule_rtimer")
             (("1" (ASSERT)
               (("1" (APPLY-EXTENSIONALITY)
```

```
            (("1" (DELETE 2) (("1" (GRIND-I) NIL)))
             ("2" (DELETE 2) (("2" (GRIND-I) NIL)))
             ("3" (DELETE 2) (("3" (GRIND-I) NIL)))
             ("4" (DELETE 2) (("4" (GRIND-I) NIL)))
             ("5" (DELETE 2) (("5" (GRIND-I) NIL)))
             ("6" (DELETE 2) (("6" (GRIND-I) NIL)))
             ("7" (DELETE 2) (("7" (GRIND-I) NIL)))
             ("8" (DELETE 2) (("8" (GRIND-I) NIL))))))))
      ("2" (EXPAND "Rule_rtimer")
       (("2" (ASSERT)
         (("2" (APPLY-EXTENSIONALITY 2)
           (("1" (DELETE 3) (("1" (GRIND-I) NIL)))
            ("2" (DELETE 3) (("2" (GRIND-I) NIL)))
            ("3" (DELETE 3) (("3" (GRIND-I) NIL)))
            ("4" (DELETE 3) (("4" (GRIND-I) NIL)))
            ("5" (DELETE 3) (("5" (GRIND-I) NIL)))
            ("6" (DELETE 3)
             (("6" (GRIND-I)
               (("6" (LEMMA "I_safe_8")
                 (("6" (EXPAND "IMPLIES")
                   (("6" (INST?)
                     (("6"
                       (BDDSIMP (-1 -3))
                       (("6"
                         (EXPAND "a_safe_8")
                         (("6" (PROPAX) NIL)))))))))))))))
            ("7" (DELETE 3) (("7" (GRIND-I) NIL)))))))))
      ("3" (EXPAND "Rule_rtimer")
       (("3" (ASSERT) (("3" (GRIND-I) NIL)))))
      ("4" (EXPAND "Rule_rtimer")
       (("4" (EXPAND "abs" 2 2)
         (("4" (LIFT-IF)
           (("4" (RECORD)
             (("4" (SIMPLIFY)
               (("4" (PROPAX) NIL))))))))))))))))))))))
(|transition_preserves| "" (SKOSIMP)
 (("" (EXPAND "transition")
   (("" (FLATTEN)
     (("" (SPLIT)
```

```
(("1" (SELECT 1)
  (("1" (LEMMA "write_req_preserves")
    (("1" (INST?) (("1" (BDDSIMP) (("1" (PROPAX) NIL)))))))))
 ("2" (SELECT 2)
  (("2" (LEMMA "read_conf_preserves")
    (("2" (EXPAND "preserves")
      (("2" (INST?)
        (("2" (BDDSIMP) (("2" (PROPAX) NIL))))))))))
 ("3" (SELECT 3)
  (("3" (LEMMA "read_ind_preserves")
    (("3" (EXPAND "preserves")
      (("3" (INST?)
        (("3" (BDDSIMP) (("3" (PROPAX) NIL))))))))))
 ("4" (SELECT 4)
  (("4" (LEMMA "read_ind_error_preserves")
    (("4" (EXPAND "preserves")
      (("4" (INST?) (("4" (GRIND-I) NIL)))))))))
 ("5" (SELECT 5)
  (("5" (LEMMA "lose_msg_preserves")
    (("5" (EXPAND "preserves")
      (("5" (INST?) (("5" (GRIND-I) NIL)))))))))
 ("6" (SELECT 6)
  (("6" (LEMMA "lose_ack_preserves")
    (("6" (EXPAND "preserves")
      (("6" (INST?) (("6" (GRIND-I) NIL)))))))))
 ("7" (SELECT 7)
  (("7" (LEMMA "read_req_preserves")
    (("7" (EXPAND "preserves")
      (("7" (INST?) (("7" (GRIND-I) NIL)))))))))
 ("8" (SELECT 8)
  (("8" (LEMMA "write_K_preserves")
    (("8" (EXPAND "preserves")
      (("8" (INST?) (("8" (GRIND-I) NIL)))))))))
 ("9" (SELECT (9 10))
  (("9" (CASE "head(s1!1)=last-1")
    (("1" (DELETE 1)
      (("1" (LEMMA "read_L_preserves1")
        (("1" (EXPAND "c_preserves")
          (("1" (INST?) (("1" (GRIND-I) NIL)))))))))
```

```
    ("2" (DELETE 3)
     (("2" (LEMMA "read_L_preserves2")
       (("2" (EXPAND "c_preserves")
         (("2" (INST?) (("2" (GRIND-I) NIL)))))))))))))
("10" (SELECT 11)
 (("10" (LEMMA "write_conf_preserves")
   (("10" (EXPAND "preserves")
     (("10" (INST?) (("10" (GRIND-I) NIL))))))))
("11" (SELECT (12 13))
 (("11" (CASE "rn(s1!1)=max")
   (("1" (DELETE 2)
     (("1" (LEMMA "stimer1_preserves1")
       (("1" (EXPAND "c_preserves")
         (("1" (INST?) (("1" (GRIND-I) NIL)))))))))
    ("2" (DELETE 2)
     (("2" (LEMMA "stimer1_preserves2")
       (("2" (EXPAND "c_preserves")
         (("2" (INST?) (("2" (GRIND-I) NIL)))))))))))))
("12" (SELECT 14)
 (("12" (LEMMA "stimer2_preserves")
   (("12" (EXPAND "preserves")
     (("12" (INST?) (("12" (GRIND-I) NIL))))))))
("13" (SELECT 15)
 (("13" (LEMMA "read_K_preserves")
   (("13" (EXPAND "preserves")
     (("13" (INST?) (("13" (GRIND-I) NIL))))))))
("14" (SELECT 16)
 (("14" (LEMMA "write_ind_preserves")
   (("14" (EXPAND "preserves")
     (("14" (INST?) (("14" (GRIND-I) NIL))))))))
("15" (SELECT 17)
 (("15" (LEMMA "write_L_preserves")
   (("15" (EXPAND "preserves")
     (("15" (INST?) (("15" (GRIND-I) NIL))))))))
("16" (SELECT 18)
 (("16" (LEMMA "write_ind_error_preserves")
   (("16" (EXPAND "preserves")
     (("16" (INST?) (("16" (GRIND-I) NIL))))))))
("17" (SELECT 19)
```

```
        (("17" (LEMMA "rtimer_preserves")
          (("17" (EXPAND "preserves")
            (("17" (INST?) (("17" (GRIND-I) NIL)))))))))))))))))
(|init_preserved| "" (ASSERT)
 (("" (AUTO-REWRITES) (("" (ASSERT) (("" (GRIND-I) NIL)))))))
(|property_preserved| "" (GRIND-I) NIL))
```

C.4 Overview of Lemmas

```
Proof summary for theory protocol_lemmas
   I_spc_1...............................................proved - complete
   I_safe_4..............................................proved - complete
   I_safe_7..............................................proved - complete
   I_safe_8..............................................proved - complete
   I_safe_9..............................................proved - complete
   Theory totals: 5 formulas, 5 attempted, 5 succeeded.

Proof summary for theory ctl
   Theory totals: 0 formulas, 0 attempted, 0 succeeded.

Proof summary for theory abstract_protocol
   safety................................................proved - complete
   Theory totals: 1 formulas, 1 attempted, 1 succeeded.

Proof summary for theory abstraction
   write_req_preserves...................................proved - complete
   read_conf_preserves...................................proved - complete
   read_ind_preserves....................................proved - complete
   read_ind_error_preserves..............................proved - complete
   lose_msg_preserves....................................proved - complete
   lose_ack_preserves....................................proved - complete
   read_req_preserves....................................proved - complete
   write_K_preserves.....................................proved - complete
   read_L_preserves1.....................................proved - complete
   read_L_preserves2.....................................proved - complete
   write_conf_preserves..................................proved - complete
```

```
    stimer1_preserves1..................................proved - complete
    stimer1_preserves2..................................proved - complete
    stimer2_preserves...................................proved - complete
    read_K_preserves....................................proved - complete
    write_ind_preserves.................................proved - complete
    write_L_preserves...................................proved - complete
    write_ind_error_preserves...........................proved - complete
    rtimer_preserves....................................proved - complete
    init_preserved......................................proved - complete
    transition_preserves................................proved - complete
    property_preserved..................................proved - complete
    Theory totals: 22 formulas, 22 attempted, 22 succeeded.

Grand Totals: 28 proofs, 28 attempted, 28 succeeded.
```

C.5 Dependencies Between Lemmas

```
I_spc_1 depends on the following proved theorems:
    protocol_safe.d_spc_1

I_safe_4 depends on the following proved theorems:
    protocol_safe.d_safe_4
    protocol_safe.p_safe_4

I_safe_7 depends on the following proved theorems:
    protocol_safe.d_safe_7
    protocol_safe.p_safe_7

I_safe_8 depends on the following proved theorems:
    protocol_safe.d_safe_8
    protocol_safe.p_safe_8

I_safe_9 depends on the following proved theorems:
    protocol_safe.d_safe_9
    protocol_safe.p_safe_9
```

The proofs of the above 5 lemmas depend on 45 out of the 58 invariants defined in the theory `protocol_safe` in appendix B.1. Those it does <u>not</u> depend on are the following:

```
ref_abusy
ref_aerror
spc_6
spc_9
rpc_8
stimer2_2
safe_1
safe_2
safe_3
safe_5
safe_6
safe_10
safe
```

The remaining dependencies are then as follows.

```
write_conf_preserves depends on the following proved theorems:
    protocol_safe.d_safe_4
    protocol_safe.p_safe_4
    protocol_safe.d_spc_1
    protocol_lemmas.I_spc_1
    protocol_lemmas.I_safe_4

read_K_preserves depends on the following proved theorems:
    protocol_safe.d_safe_8
    protocol_safe.p_safe_8
    protocol_lemmas.I_safe_8

write_ind_preserves depends on the following proved theorems:
    protocol_safe.p_safe_9
    protocol_safe.d_safe_8
    protocol_safe.p_safe_7
    protocol_safe.d_safe_9
    protocol_safe.d_safe_7
    protocol_lemmas.I_safe_7
```

```
    protocol_lemmas.I_safe_9
    protocol_safe.p_safe_8
    protocol_lemmas.I_safe_8

rtimer_preserves depends on the following proved theorems:
    protocol_safe.d_safe_8
    protocol_safe.p_safe_8
    protocol_lemmas.I_safe_8

transition_preserves depends on the following proved theorems:
    protocol_safe.d_safe_7
    abstraction.read_L_preserves2
    abstraction.read_L_preserves1
    protocol_lemmas.I_safe_7
    protocol_safe.d_safe_8
    abstraction.stimer1_preserves1
    protocol_safe.d_safe_4
    abstraction.write_ind_preserves
    abstraction.read_req_preserves
    abstraction.write_L_preserves
    protocol_lemmas.I_safe_8
    protocol_safe.p_safe_4
    abstraction.read_ind_error_preserves
    abstraction.write_conf_preserves
    abstraction.read_K_preserves
    abstraction.lose_msg_preserves
    abstraction.write_K_preserves
    protocol_safe.d_spc_1
    protocol_safe.p_safe_7
    abstraction.write_req_preserves
    abstraction.stimer1_preserves2
    protocol_safe.p_safe_9
    abstraction.read_ind_preserves
    protocol_safe.d_safe_9
    abstraction.rtimer_preserves
    protocol_lemmas.I_safe_9
    protocol_safe.p_safe_8
    abstraction.read_conf_preserves
    protocol_lemmas.I_spc_1
```

```
abstraction.write_ind_error_preserves
abstraction.lose_ack_preserves
protocol_lemmas.I_safe_4
abstraction.stimer2_preserves
```

Appendix D

Model Checking the Abstraction in Other Systems

This appendix associates to chapter 6. It contains listings of programs in Murϕ and SMV respectively.

D.1 The Abstraction in Murϕ

```
     -- Shared Declarations --

Type
  Msg  : Record first, last, toggle : boolean End;

Var
  K : Msg;
  K_full : boolean;
  L : boolean;


     -- Sender Declarations --

Type
```

```
  Spc  : Enum{WR, SF, WA, SC, WT2};
  Conf : Enum{OK, NOT_OK, DONT_KNOW};
  File : Enum{NONE,ONE,MANY};


Var
  req : File;
  req_full : boolean;
  conf : Conf;
  conf_full : boolean;
  spc : Spc;
  sfirst : boolean;
  stoggle : boolean;
  file : File;
  rn : boolean;
  stimer1_on : boolean;
  stimer1_enabled : boolean;
  stimer2_on : boolean;
  stimer2_enabled : boolean;


     -- Receiver Declarations --

Type
  Rpc  : Enum{WF, SI, SA, RTS, NOK};
  IndT : Enum{FIRST, INCOMPLETE, LAST};

Var
  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;
```

```
    -- Abstract automaton --

Var
  abusy  : boolean;
  afile  : boolean;
  afirst : boolean;
  aerror : boolean;
  data_is_safe : boolean;


Procedure verify_REQ();
Begin
  assert !abusy "REQ";
  abusy := true;
  afile := true;
End;

Procedure verify_IND(i:IndT);
Begin
  assert  abusy & !aerror & afile &
          data_is_safe &
          (i=FIRST -> afirst) &
          (i=INCOMPLETE -> !afirst) "IND";
  afirst := (i  = LAST);
  afile  := (i != LAST);
End;

Procedure verify_IND_ERR();
Begin
  assert aerror "IND_ERR";
  afirst := true;
  aerror := false;
End;

Procedure verify_CONF(c:Conf);
Begin
  assert abusy & !aerror &
         (c=OK -> !afile) &
```

```
        (c=NOT_OK -> afile) "CONF";
  abusy := false;
  aerror := !afirst;
  afile := false;
End;


     -- Some Sender Related Functions --

Function tail_MANY(f:File):File;
Begin
  Switch f
    Case NONE : Return NONE;
    Case ONE  : Return NONE;
    Case MANY : Return MANY;
  End;
End;

Function tail_ONE(f:File):File;
Begin
  Switch f
    Case NONE : Return NONE;
    Case ONE  : Return NONE;
    Case MANY : Return ONE;
  End;
End;


     -- Initialisation --

Startstate
  Begin
    K.first := false;
    K.last := false;
    K.toggle := false;
    K_full := false;
    L := false;
    req := ONE;
    req_full := false;
```

```
    conf := OK;
    conf_full := false;
    spc := WR;
    sfirst := true;
    stoggle := false;
    file := NONE;
    rn := false;
    stimer1_on := false;
    stimer1_enabled := false;
    stimer2_on := false;
    stimer2_enabled := false;
    ind_indication := FIRST;
    ind_full := false;
    ind_error := false;
    rpc := WF;
    rfirst := true;
    rtoggle := false;
    ctoggle := false;
    msg.first := false;
    msg.last := false;
    msg.toggle := false;
    rtimer_on := true;
    rtimer_enabled := false;
    data_is_safe := true;
    abusy := false;
    afile := false;
    afirst := true;
    aerror := false;
  End;

Startstate
  Begin
    K.first := false;
    K.last := false;
    K.toggle := false;
    K_full := false;
    L := false;
    req := MANY;
    req_full := false;
```

```
    conf := OK;
    conf_full := false;
    spc := WR;
    sfirst := true;
    stoggle := false;
    file := NONE;
    rn := false;
    stimer1_on := false;
    stimer1_enabled := false;
    stimer2_on := false;
    stimer2_enabled := false;
    ind_indication := FIRST;
    ind_full := false;
    ind_error := false;
    rpc := WF;
    rfirst := true;
    rtoggle := false;
    ctoggle := false;
    msg.first := false;
    msg.last := false;
    msg.toggle := false;
    rtimer_on := true;
    rtimer_enabled := false;
    data_is_safe := true;
    abusy := false;
    afile := false;
    afirst := true;
    aerror := false;
  End;


     -- Environment Actions --

Ruleset f : File Do
  Rule "write_req"
    f!=NONE &
    !req_full
      ==>
    req_full := true;
```

```
    req := f;
  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;


     -- Sender Program --
```

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

Rule "write_K"
  spc = SF &
  !K_full
    ==>
  K_full := true;
  K.first := sfirst;
  K.last := (file=ONE);
  K.toggle:= stoggle;
  spc:= WA;
  rn := true;
  stimer1_on := true;
End;

Rule "read_L_MANY"
  spc = WA &
  L
    ==>
  L := false;
  If file=ONE
    Then spc := SC;
    Else spc := SF;
  End;
  sfirst := (file=ONE);
  If !(file=ONE) Then rn := false End;
  file := tail_MANY(file);
  stoggle := !stoggle;
  stimer1_on := false;
  stimer1_enabled := false;
End;
```

```
Rule "read_L_ONE"
  spc = WA &
  L
    ==>
  L := false;
  If file=ONE
    Then spc := SC;
    Else spc := SF;
  End;
  sfirst := (file=ONE);
  If !(file=ONE) Then rn := false End;
  file := tail_ONE(file);
  stoggle := !stoggle;
  stimer1_on := false;
  stimer1_enabled := false;
End;

Rule "write_conf"
  spc = SC &
  !conf_full
    ==>
  conf_full := true;
  If file=NONE Then
      conf := OK
    Elsif file=ONE & rn Then
      conf := DONT_KNOW;
    Else
      conf := NOT_OK;
  End;
  If file=NONE Then
      spc := WR;
    Else
      spc := WT2;
      sfirst := true;
      stoggle := !stoggle;
      stimer2_on := true;
      rtimer_enabled := true;
  End;
```

```
  file := NONE;
  rn := false;
  verify_CONF(conf);
End;

Rule "stimer1_QUIT"
  stimer1_on & stimer1_enabled
    ==>
  spc := SC;
  stimer1_on := false;
  stimer1_enabled := false;
End;

Rule "stimer1_RETRY"
  stimer1_on & stimer1_enabled
    ==>
  spc := SF;
  stimer1_on := false;
  stimer1_enabled := false;
End;

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


     -- Receiver Program --

Rule "read_K"
  rpc = WF &
  K_full
    ==>
  K_full := false;
  msg.first := K.first;
  msg.last := K.last;
```

```
  msg.toggle := K.toggle;
  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;
  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_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;
```

D.2 The Abstraction in SMV

```
MODULE main
VAR
  s   : state;
  write_req       : process write_req(s);
  read_conf       : process read_conf(s);
  read_ind        : process read_ind(s);
  read_ind_error  : process read_ind_error(s);
  lose_msg        : process lose_msg(s);
  lose_ack        : process lose_ack(s);
  read_req        : process read_req(s);
```

```
  write_K         : process write_K(s);
  read_L_MANY     : process read_L_MANY(s);
  read_L_ONE      : process read_L_ONE(s);
  write_conf      : process write_conf(s);
  stimer1_QUIT    : process stimer1_QUIT(s);
  stimer1_RETRY   : process stimer1_RETRY(s);
  stimer2         : process stimer2(s);
  read_K          : process read_K(s);
  write_ind       : process write_ind(s);
  write_L         : process write_L(s);
  write_ind_error : process write_ind_error(s);
  rtimer          : process rtimer(s);
SPEC
  AG s.SAFE


MODULE Msg
VAR
  first : boolean;
  last : boolean;
  toggle : boolean;


MODULE state
VAR
  K : Msg;
  K_full : boolean;
  L : boolean;
  req : {NONE,ONE,MANY};
  req_full : boolean;
  conf : {OK, NOT_OK, DONT_KNOW};
  conf_full : boolean;
  spc : {WR, SF, WA, SC, WT2};
  sfirst : boolean;
  stoggle : boolean;
  file : {NONE,ONE,MANY};
  rn : boolean;
  stimer1_on : boolean;
  stimer1_enabled : boolean;
```

```
  stimer2_on : boolean;
  stimer2_enabled : boolean;
  ind_indication : {FIRST, INCOMPLETE, LAST};
  ind_full : boolean;
  ind_error : boolean;
  rpc : {WF, SI, SA, RTS, NOK};
  rfirst : boolean;
  rtoggle : boolean;
  ctoggle : boolean;
  msg : Msg;
  rtimer_on : boolean;
  rtimer_enabled : boolean;
  abusy  : boolean;
  afile  : boolean;
  afirst : boolean;
  aerror : boolean;
  data_is_safe : boolean;
  SAFE : boolean;
INIT
    K.first = 0
  & K.last = 0
  & K.toggle = 0
  & K_full = 0
  & L = 0
  & req in {ONE,MANY}
  & req_full = 0
  & conf = OK
  & conf_full = 0
  & spc = WR
  & sfirst = 1
  & stoggle = 0
  & file = NONE
  & rn = 0
  & stimer1_on = 0
  & stimer1_enabled = 0
  & stimer2_on = 0
  & stimer2_enabled = 0
  & ind_indication = FIRST
  & ind_full = 0
```

```
  & ind_error = 0
  & rpc = WF
  & rfirst = 1
  & rtoggle = 0
  & ctoggle = 0
  & msg.first = 0
  & msg.last = 0
  & msg.toggle = 0
  & rtimer_on = 1
  & rtimer_enabled = 0
  & data_is_safe = 1
  & abusy = 0
  & afile = 0
  & afirst = 1
  & aerror = 0
  & SAFE = 1


MODULE write_req(s)
TRANS
  running ->
  !s.req_full
ASSIGN
  next(s.req_full) := 1;
  next(s.req) := {ONE,MANY};


MODULE read_conf(s)
TRANS
  running ->
  s.conf_full
ASSIGN
  next(s.conf_full) := 0;


MODULE read_ind(s)
TRANS
  running ->
  s.ind_full
```

```
ASSIGN
  next(s.ind_full) := 0;


MODULE read_ind_error(s)
TRANS
  running ->
  s.ind_error
ASSIGN
  next(s.ind_error) := 0;


MODULE lose_msg(s)
TRANS
  running ->
  s.K_full
ASSIGN
  next(s.K_full) := 0;
  next(s.stimer1_enabled) := 1;


MODULE lose_ack(s)
TRANS
  running ->
  s.L
ASSIGN
  next(s.L) := 0;
  next(s.stimer1_enabled) := 1;


MODULE read_req(s)
TRANS
  running ->
  s.spc = WR &
  s.req_full
ASSIGN
  next(s.req_full) := 0;
  next(s.file) := s.req;
  next(s.spc) := SF;
```

```
  -- automaton
  next(s.SAFE) := !s.abusy;
  next(s.abusy) := 1;
  next(s.afile) := 1;


MODULE write_K(s)
TRANS
  running ->
  s.spc = SF &
  !s.K_full
ASSIGN
  next(s.K_full) := 1;
  next(s.K.first) := s.sfirst;
  next(s.K.last) := (s.file=ONE);
  next(s.K.toggle) := s.stoggle;
  next(s.spc) := WA;
  next(s.rn) := 1;
  next(s.stimer1_on) := 1;


MODULE read_L_MANY(s)
TRANS
  running ->
  s.spc = WA &
  s.L
ASSIGN
  next(s.L) := 0;
  next(s.spc) :=
    case
      s.file=ONE : SC;
      1          : SF;
    esac;
  next(s.sfirst) := (s.file=ONE);
  next(s.rn) :=
    case
      !(s.file=ONE) : 0;
      1             : s.rn;
    esac;
```

```
  next(s.file) :=
    case
      s.file=NONE : NONE;
      s.file=ONE  : NONE;
      s.file=MANY : MANY;
    esac;
  next(s.stoggle) := !s.stoggle;
  next(s.stimer1_on) := 0;
  next(s.stimer1_enabled) := 0;


MODULE read_L_ONE(s)
TRANS
  running ->
  s.spc = WA &
  s.L
ASSIGN
  next(s.L) := 0;
  next(s.spc) :=
    case
      s.file=ONE : SC;
      1          : SF;
    esac;
  next(s.sfirst) := (s.file=ONE);
  next (s.rn) :=
    case
      !(s.file=ONE) : 0;
      1             : s.rn;
    esac;
  next(s.file) :=
    case
      s.file=NONE : NONE;
      s.file=ONE  : NONE;
      s.file=MANY : ONE;
    esac;
  next(s.stoggle) := !s.stoggle;
  next(s.stimer1_on) := 0;
  next(s.stimer1_enabled) := 0;
```

```
MODULE write_conf(s)
TRANS
  running ->
  s.spc = SC &
  !s.conf_full
ASSIGN
  next(s.conf_full) := 1;
  next(s.conf) :=
    case
      s.file=NONE       : OK;
      s.file=ONE & s.rn : DONT_KNOW;
      1                 : NOT_OK;
    esac;
DEFINE
  empty := (s.file=NONE);
ASSIGN
  next(s.spc) :=            case  empty : WR; 1 : WT2; esac;
  next(s.sfirst) :=         case !empty : 1; 1 : s.sfirst; esac;
  next(s.stoggle) :=        case !empty : !s.stoggle; 1 : s.stoggle; esac;
  next(s.stimer2_on) :=     case !empty : 1; 1 : s.stimer2_on; esac;
  next(s.rtimer_enabled) := case !empty : 1; 1 : s.rtimer_enabled; esac;
  next(s.file) := NONE;
  next(s.rn) := 0;
  -- automaton
  next(s.SAFE) :=
    s.abusy & !s.aerror &
    (next(s.conf)=OK -> !s.afile) &
    (next(s.conf)=NOT_OK -> s.afile);
  next(s.abusy) := 0;
  next(s.aerror) := !s.afirst;
  next(s.afile) := 0;


MODULE stimer1_QUIT(s)
TRANS
  running ->
  s.stimer1_on & s.stimer1_enabled
ASSIGN
```

```
  next(s.spc) := SC;
  next(s.stimer1_on) := 0;
  next(s.stimer1_enabled) := 0;


MODULE stimer1_RETRY(s)
TRANS
  running ->
  s.stimer1_on & s.stimer1_enabled
ASSIGN
  next(s.spc) := SF;
  next(s.stimer1_on) := 0;
  next(s.stimer1_enabled) := 0;


MODULE stimer2(s)
TRANS
  running ->
  s.stimer2_on & s.stimer2_enabled
ASSIGN
  next(s.spc) := WR;
  next(s.stimer2_on) := 0;
  next(s.stimer2_enabled) := 0;


MODULE read_K(s)
TRANS
  running ->
  s.rpc = WF &
  s.K_full
ASSIGN
  next(s.K_full) := 0;
  next(s.msg.first) := s.K.first;
  next(s.msg.last) := s.K.last;
  next(s.msg.toggle) := s.K.toggle;
DEFINE
  new := !s.ctoggle | (next(s.msg.toggle) = s.rtoggle);
ASSIGN
  next(s.rpc)            := case new : SI; 1 : RTS; esac;
```

```
  next(s.rtimer_on)      := case new : 0; 1 : s.rtimer_on; esac;
  next(s.rtimer_enabled) := case new : 0; 1 : s.rtimer_enabled; esac;


MODULE write_ind(s)
TRANS
  running ->
  s.rpc = SI &
  !s.ind_full
ASSIGN
  next(s.ind_full) := 1;
  next(s.ind_indication) :=
    case
      s.msg.last  : LAST;
      s.msg.first : FIRST;
      1           : INCOMPLETE;
    esac;
  next(s.rpc) := SA;
  next(s.rfirst) := s.msg.last;
  next(s.ctoggle) := 1;
  next(s.rtoggle) := !s.msg.toggle;
  -- automaton
  next(s.SAFE) :=
    s.abusy & !s.aerror & s.afile &
    s.data_is_safe &
    (next(s.ind_indication)=FIRST -> s.afirst) &
    (next(s.ind_indication)=INCOMPLETE -> !s.afirst);
  next(s.afirst) := (next(s.ind_indication) = LAST);
  next(s.afile)  := !(next(s.ind_indication) = LAST);
  next(s.data_is_safe) := s.data_is_safe;


MODULE write_L(s)
TRANS
  running ->
  (s.rpc = SA | s.rpc = RTS) &
  !s.L
ASSIGN
  next(s.L) := 1;
```

```
  next(s.rtimer_on) := case s.rpc = SA : 1; 1 : s.rtimer_on; esac;
  next(s.rpc) := WF;


MODULE write_ind_error(s)
TRANS
  running ->
  s.rpc = NOK &
  !s.ind_error
ASSIGN
  next(s.ind_error) := 1;
  next(s.rpc) := WF;
  next(s.rfirst) := 1;
  next(s.rtimer_on) := 1;
  next(s.stimer2_enabled) := 1;
  -- automaton
  next(s.SAFE) := s.aerror;
  next(s.afirst) := 1;
  next(s.aerror) := 0;


MODULE rtimer(s)
TRANS
  running ->
  s.rtimer_on & s.rtimer_enabled
ASSIGN
  next(s.ctoggle) := 0;
  next(s.rpc) :=
    case !s.rfirst : NOK; 1 : s.rpc; esac;
  next(s.rtimer_on) :=
    case !s.rfirst : 0; 1 : s.rtimer_on; esac;
  next(s.stimer2_enabled) :=
    case s.rfirst : 1; 1 : s.stimer2_enabled; esac;
  next(s.rtimer_enabled) := 0;
```

Bibliography

[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 [11].

[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 Computer 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] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[9] E. A. Emerson. *Handbook of Theoretical Computer Science: Temporal and Modal Logic*, chapter 16, pages 996–1072. Elsevier Science Publishers, 1990.

[10] 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.

[11] 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.

[12] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.

[13] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[14] G. Janssen. ROBDD software. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[15] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Trans. on S.W. Engg*, SE-10(4):325–342, July 1984.

[16] L. Lamport. The Temporal Logic of Actions. Technical report, Digital Equipment Corporation (DEC) Systems Research Center, Palo Alto, California, USA, April 1994.

[17] 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.

[18] 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.

[19] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[20] 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.

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

[22] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O automata. Draft manuscript, 1995.

[23] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE), Saratoga - NY, Lecture Notes in Artificial Intelligence, Volume 607*, pages 748–752. Springer Verlag, June 1992.

[24] 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. Long version of this paper is available.

[25] N. Shankar. Mechanized verification of real-time systems using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, USA, March 1993.