VERIFYING A SLIDING-WINDOW PROTOCOL USING PVS

Vlad Rusu
IRISA/INRIA, Campus Univ. de Beaulieu, 35042 Rennes Cedex, France
rusu@irisa.fr

Abstract We present the deductive verification of safety and liveness properties of a sliding-window protocol using the PVS theorem prover. The protocol is modeled in an operational style which is close to an actual program. It has parametric window sizes for both sender and receiver, and unbounded, lossy communication channels carrying unbounded data. The proofs are done using invariant-strengthening techniques, encoded as PVS automated strategies based on heuristics and decision procedures.

1. INTRODUCTION

In the rapidly growing field of formal verification, *deductive techniques* [13] and tools [4, 8, 14] are general and powerful ways to verify properties of hardware and software systems. However, many practitioners and researchers hesitate to use deductive methods, which are perceived to be difficult, time-consuming and requiring advanced theoretical knowledge. This is in contrast with *algorithmic techniques* [2, 5, 9] which are completely automatic once a suitable abstraction of the system has been found and entered into, *e.g.*, a model-checking tool.

The goal of this paper is to advocate the use of deductive techniques. Specifically, we show that although deductive verification is not fully automatic, it can be performed in a systematic way, using specific approaches, strategies, and automatic procedures on decidable subproblems that reduce the need for human intervention. This intervention remains necessary, but only as a general guideline to organize the verification process, i.e, to decompose the verification problem into manageable subproblems. Then, the subproblems can be solved in a highly automated manner.

As a case study we use an infinite-state sliding-window protocol for which we verify safety and liveness properties using the PVS theorem

prover [14]. The protocol is a popular application for verification methods, both algorithmic and deductive. To our knowledge, our work is among the most general in terms of the properties verified and of the level of detail at which the protocol is modeled.

The rest of the paper is organized as follows. In Section 2 we describe the sliding-window protocol and related work on algorithmic and deductive approaches that have been applied to it. In Section 3 we briefly present the PVS theorem prover and describe, on a very simple example, the *invariant-strengthening* proof technique. In Section 4 we present in some detail our encoding of the sliding-window protocol in PVS, and in Section 5 we show how to prove safety and liveness properties by making extensive use of invariant strengthening. We conclude the presentation with ideas on how to further automate the deductive verification process. The specification and verification took three weeks to a moderately experienced PVS user and amounted to proving a total of eighteen theorems. PVS is available free of charge at `http://pvs.csl.sri.com`, and our sliding-window case study can be downloaded from `http://www.irisa.fr/pampa/perso/rusu/forte`.

2. RELATED WORK

We briefly present the sliding-window protocol and previous approaches that have been employed to verify it.

The Sliding-Window Protocol The sliding-window protocol (see, *e.g.*, [17]) is a member of the data link layer. Its main function is to deliver a sequence of messages between a sender and a receiver in the correct order, regardless of possible losses that can occur during the transmission on the communication channels. The general principle underlying the protocol is that of pipelining. Instead of waiting for an acknowledgment after each message, the sender will continue sending messages until it has in its buffer, or "window", a total of, say, `sw` unacknowledged messages.

Similarly, the receiver will not just discard a message that has not the next expected sequence number, but will store it in its window, provided it is at most `rw` many positions from the expected message. With this flexibility on both sides and using adequate values for the window sizes `sw` and `rw`, typically computed from other parameters, *e.g.*, the expected travel times of messages, the protocol can achieve very efficient transmission rates.

Verification of the Protocol The properties to be verified are divided into *safety* properties, which ensure that messages are delivered

in the correct order, and *liveness* properties, which guarantee that every message sent is eventually delivered.

Algorithmic approaches. These approaches typically verify finite-state instances of the protocol. In [15] safety properties for small window and channel sizes are verified by brute-force model-checking. In [12] the authors use one specific equivalence relation that allows them to handle window sizes as high as seven and to prove both safety and liveness properties. More recently, the paper [6] shows how to verify both kinds of properties on a version of the protocol with channel size eleven and window sizes two for both sender and receiver, using a specialized data structure called a Queue Difference Diagram.

Deductive approaches. These approaches typically verify general infinite-state specifications of the protocol, *e.g.*, with unbounded window and channel sizes. An early deductive verification effort is presented in [1]. Only safety properties are verified, and a large number of invariants (over a thousand) were involved in the process. Another early work [3] presents the verification of safety and liveness properties of a specification of the protocol which includes real-time aspects. The level of automation of tools available at that time was low and the verification process included proving a large number (tens to hundreds) of trivial facts, *e.g.*, that addition is commutative.

A recent paper involving a much more automated approach is [16]. Here, the authors use the weak monadic second-order logic with one successor (WS1S), a decidable logic that is expressive enough to encode finite sets and sequences and limited integer operations. A higher-level language is used to describe both the protocol and safety properties, which are then translated to WS1S. Two versions of the protocol are verified. One version considers a sender with an unbounded window size, which can be nondeterministically modified on-the-fly. In the second version, the sender's window size is 256. The receiver's window size is, in both versions, one. A total of eight invariants are required to prove the main safety property. Liveness properties are not considered.

This work. We verify an operational description of the protocol which is very close to a programming language, with parametric window sizes for both sender and receiver and unbounded communication channels. As [16], we emphasize on generality, automation, and compactness (*i.e.*, there is a relatively small number of lemmas to prove), but treat the more general case of receiver's window size greater than one and verify both safety and liveness properties.

3. PVS AND EXTENDED AUTOMATA

In this section we briefly describe the PVS specification and verification system and the model of *extended automata* that can be used for modeling reactive programs such as communication protocols. We demonstrate on a simple example how to prove properties of extended automata using the invariant-strengthening technique and how this technique is implemented in PVS. To the best of our knowledge, invariant strengthening in PVS was first mentioned in [11].

PVS. The PVS system consists of an input language, a typechecker, and an interactive prover. The *input language* is typed higher-order logic with a rich type system including simple types such as booleans, enumerations, integers, and records, and more complex function types, subtypes, dependent types, and abstract datatypes. Having such an expressive language makes it easy to specify, *e.g.*, concurrent programs in a natural way, very close to a programming language. The drawback is that *typechecking* the input language is undecidable. However, PVS transforms this apparent weakness into an actual strength, because whenever the typechecker cannot decide whether an expression is type-correct, it generates a proof obligation. Most of the proof obligations can be discharged automatically, and obligations that cannot be proved often point to subtle errors in the specification.

A PVS *proof* is a tree, the root of which is the theorem being proved. The leaves of the tree are called *pending subgoals.* A proof proceeds as a sequence of commands, each of which transforms the proof tree by either proving a pending subgoal or by replacing a pending subgoal by a new set of pending subgoals. There are many proof commands, from propositional and first-order logic commands, to decision procedures and heuristic quantifier instantiation techniques, all of which can be combined into high-level, user-defined proof strategies.

Extended Automata. Extended automata are a computational model for reactive programs. An extended automaton A consists of a finite set of typed variables V, an initial condition Θ and a finite set of guarded transitions $\mathcal{T}$. The variables can be either control or data variables; the control variables are of a finite type *Location.* Each transition $\tau \in \mathcal{T}$ is labelled and consists of a guard and an assignment. For example, in the automaton illustrated in Fig. 1, the type *Location* consists of the two values l_1, l_2. There is one control variable pc and one integer data variable x. The initial condition is Θ: $pc = l_1 \wedge x = 0$. There are two

transitions, τ_1 : $l_1 \xrightarrow{inc} l_2$ and τ_2 : $l_2 \xrightarrow{dec} l_1$. The guard of transition τ_1 is $pc = l_1$ and its assignment is $x := x + 1$.

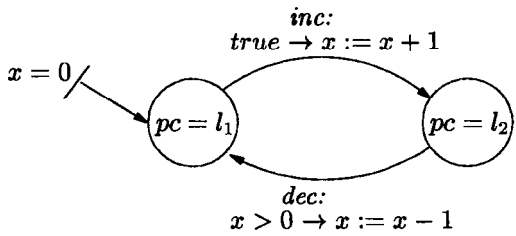

Figure 1. Example of Extended Automaton

A *state* is a type-consistent valuation of the variables. Each transition τ induces a *transition relation* ρ_τ relating the possible before and after states. The global transition relation of the system is $\rho_T = \bigcup_{\tau \in \tau} \rho_\tau$. A *run* of the automaton is an infinite sequence of states, in which the first state satisfies the initial condition and every two consecutive states are in the transition relation. For example, in the extended automaton illustrated in Fig. 1, there is only one initial state that has location l_1 and variable $x = 0$. From this state the program can take the transition *inc* (its guard is *true*), assigning 1 to x, moving to location l_2, etc.

An *invariant* is an assertion that holds at every state of every run. Invariants are the simplest form of safety properties. For example, it is not hard to show that $x \geq 0$ is an invariant of the automaton in Fig. 1.

Invariant strengthening. A state predicate is *inductive* if it holds initially and, if it is true at a given state s, then it is also true at all states s' that are successors of s through the transition relation ρ_T. For example, in the automaton represented in Figure 1, it is not hard to check that the predicate $x \geq 0$ is inductive. Indeed, it is true initially, and from any state satisfying $x \geq 0$, any transition, *i.e.*, *inc* or *dec*, will lead to a state also satisfying the predicate.

Clearly, an inductive predicate is an invariant. The converse is not true: consider, for example, the predicate $pc = l_2 \supset x = 1$. It is not hard to see that it is an invariant, that is, on every run, whenever control is at location l_2, variable x equals 1. But the predicate is not inductive: by knowing only that it is true before transition *inc*, it cannot be inferred that it is still true after the transition is taken. This does not mean the predicate is not an invariant, it is just that we cannot prove this fact by using only the predicate itself as induction hypothesis. To succeed

in the proof we need additional information, which can be obtained by *invariant strengthening.*

For example, consider again the predicate φ : $pc = l_2 \supset x = 1$, a state s' of the automaton in Fig. 1 satisfying this predicate, and a state s from which s' is obtained by taking transition *inc.* Then, clearly, s satisfies ψ : $pc = l_1 \supset x = 0$. The predicate ψ is a pre-condition for φ to hold. Now, it turns out that conjunction $\varphi \wedge \psi$ is inductive, thus, it is an invariant. In particular, we have just proved that φ is an invariant too.

Thus, invariant strengthening is the following process: to prove a predicate is an invariant, first try to prove that it is inductive. If this is the case, then the proof is done. Otherwise, compute the pre-condition of the predicate, and try to prove that the conjunction of the invariant with the pre-condition is inductive. The process can be iterated until an inductive invariant is obtained. This is not guaranteed to happen in general, because the problem of proving invariants of general extended automata is undecidable. However, by using this approach in an interactive theorem prover, the user can often detect infinite patterns of behavior and formulate a predicate which is not just the pre-condition of a predicate by one transition, but the fixpoint of an infinite sequence of such operations. We demonstrate this latter feature in the next section.

4. THE SLIDING-WINDOW PROTOCOL IN PVS

In this section we present the sliding-window protocol in more detail, then show how to encode it as an extended automaton in PVS.

Architecture of the Protocol The architecture of the protocol is represented in Figure 2. The sender obtains data from a FIFO stream of data called the `source`. Each data element is first saved into the sender's window, a FIFO buffer called `sndWindow`. The sender takes an element from its window, associates an index to it (a natural number keeping track of the order in which data has been obtained from the source), and sends the resulting record called a Message to the Message Channel `MsgChan`. The latter is a lossy FIFO, which may lose, but not reorder or create messages.

On the receiver side, messages are removed from `MsgChan` and, if a message's index is within the bounds of the receiver's window `rcvWindow`, it is stored there, otherwise, it is discarded. The receiver delivers contiguous sequences of messages to the external data `target`, and acknowledges the message with the highest index that it has delivered by sending the val-

ue of that index to the acknowledgment channel `AckChan` (another lossy queue). Finally, the sender reads acknowledgments from `AckChan`. An acknowledgment is considered valid if it is between the bounds defined by the bottom of the sender's window and the current top of that window, *i.e.*, up to the message number obtained from the data source. A valid acknowledgement `a` makes the bottom of the sender's window become `a+1`, meaning that all messages with value at most `a` are acknowledged.

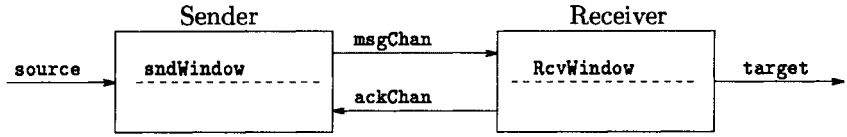

Figure 2. Architecture of the Sliding-Window Protocol

PVS encoding: declarations. First, the *labels* of the protocol are encoded into theory `Action_and_types` (Figure 3) that includes some type declarations and an `Action` abstract datatype to encode the externally visible actions of the protocol. The meaning of each action is explained in the comments. Note that actions have *parameters*, *e.g.*, `GET(d)` to express obtaining datum `d` from the data source. (Technically, in terms of abstract datatypes, `GET` is a constructor, `d` is an accessor, and `GET?` is a recognizer, but for better understanding it is preferable to see them as a declaration of actions with parameters.) Thus, the data being transferred between sender and receiver are natural numbers, and messages serve as recipients for the data together with an index (another natural number) that allows to reconstruct the order of messages, in case of losses during transmission.
The state of the automaton is encoded as the PVS record type `State` (cf. Fig. 4), with fields for variables of the sender, receiver, and channels:

Thus, the sender's window size `sw` and receiver's window size `rw` are strictly positive natural numbers. The receiver has three locations `receiving`, `flushing`, and `sendingAck`. Since there is only one location for the sender, it is not necessary to use a field for it. The meaning of the state variables is the following:

- variables for the data source: `source` is the infinite stream of data, modeled as a function from `nat` to `nat`. `sourceIndex` is the index of the next data element that will be passed to the sender,
- variables for the sender: `sndWindow` is the sender's window, modeled as a function from `nat` to `nat`. `sndLow` is the bottom of the sender's

```
Action_and_types  :  Theory
BEGIN
Msg: TYPE = [#data: nat, index: nat #]
Action  :  DATATYPE
BEGIN
  GET(d: nat): GET?                          %get datum d from data source
  SEND_MSG(m:Msg):  SEND_MSG?          %send message m to message channel
  RCV_ACK(a:nat):RCV_ACK?                % receive ack a from ack channel
  TIMEOUT:  TIMEOUT?                           %sender receives a timeout
  RCV_MSG (m:Msg): RCV_MSG?       %receive message m from message channel
  PUT(d:  int):  PUT?                      %deliver datum d to data target
  SEND_ACK(a:  nat):  SEND_ACK?                %send ack a to ack channel
  LOSE_MSG:  LOSE_MSG?                         %environment  loses  message
  LOSE_ACK:  LOSE_ACK?                  %environment  loses  acknowledgment
END  Action
END  Action_and_types
```

Figure 3. Sliding-Window Protocol: Types and Actions

```
sliding_window:  THEORY
BEGIN
IMPORTING  Action_and_types
sw, rw  :  posnat                        % sender and receiver window sizes
rcvControlType:   TYPE={receiving,flushing,sendingAck}  %receiver location
State:  TYPE =                                          % global state type
 [# source:  [nat-> nat], sourceIndex  :  nat,                 % data source
    sndWindow:  [nat-> nat],                               %sender's  window
    sndlow,                                    % bottom of sender's window
    sndWinIndex,                  % counts data received from data source
    sndIndex:  nat,                 % counts data sent to message channel
    rcvControl  :  rcvControlType,     %control variable for the receiver
    rcvWindow  :  [nat-> int],                             %receiver's  window
    rcvlow,                                  % bottom of receiver% window
    headMsgChan,tailMsgChan:  nat,  msgChan:  [nat->Msg],% message channel
    headAckChan,tailAckChan  :  nat,  ackChan:  [nat->nat],   % ack channel
    target  :[nat->int],targetIndex  :  nat,                  % data target
    actual  :  boolean #]  ...                       %rest of theory follows
```

Figure 4. Sliding-Window Protocol: State type

window, thus, at any moment in the execution, the current sender's window ranges from `sndWindow(sndLow)` to `sndWindow(sndLow+sw-1)`. The variable `sndWinIndex` is the position in the sender's window where the data source can place a data element, and variable `sndIndex` is the position of the next message that the sender will place of the message channel,

- variables of the receiver: `rcvControl` is the control variable. `rcvWindow` is the receiver's window, and `rcvLow` is the bottom of the receiver's

window. Thus, at any moment in time, the receiver's window consists of elements from `rcvWindow(rcvLow)` to `rcvWindow(rcvLow+rw-1)`,

- variables of the channels: the message channel `msgChan` is a function from `nat` to `Msg`. It is a `fifo` whose active content at any moment in time ranges from `msgChan(tailMsgChan)` to `msgChan(headMsgChan-1)`. There are similar declarations for the acknowledgment channel `ackChan`.

Initial states and transition relation. The *initial states*, not shown here from lack of space, consist in setting every integer variable to zero, the receiver's control to `receiving`, and receiver's window to `Absent` (syntactic sugar for -1) to denote absence of any data in it: `FORALL(n: nat): s`rcvWindow(n) = Absent`.

The *transitions* of the model are encoded as a mapping `next` from `State` and `Action` to `State`. `Action` is the PVS abstract datatype depicted in Figure 3. There are nine transitions in all: four for the sender, three for the receiver, and two for the environment that may lose messages. The system works by nondeterministically firing an enabled transition, then choosing another transition and firing it, etc. A separate PVS theory **`runs.pvs`** (not shown here from lack of space) defines runs as sequences over a generic state type and the notions of invariants and inductive invariants.

The first transition of the sender specifies that a `GET` action with parameter `d` can fire if there is space in the window: `s`sndWinIndex < s`sndLow + sw`, and `d` is the next element of the data source. If this is the case, the sender's window is modified to hold the value `d` at position `sndWinIndex`, then, `sndWinIndex` and `sourceIndex` are both incremented:

```
GET(d) : IF s`sndWinIndex < s`sndLow + sw AND d = s`source(s`sourceIndex)
         THEN s WITH [sndWindow:= s`sndWindow WITH[(s`sndWinIndex):= d],
                      sndWinIndex:= s`sndWinIndex+1,
                      sourceIndex:= s`sourceIndex+1]
         ELSE s WITH [actual := false] ENDIF,
```

The second transition of the sender takes a message `m` whose data field is the element at position `sndIndex` of the sender's window, associates the value of that same index to it, places it on the message channel, and increments indices:

```
SEND_MSG(m):IF s`sndIndex<s`sndWinIndex AND m`data=s`sndWindow(s`sndIndex)
            AND m`index = s`sndIndex THEN
              s WITH [msgChan:= s`msgChan WITH [(s`headMsgChan):= m],
                      headMsgChan := s`headMsgChan+1,
                      sndIndex:= s`sndIndex+1]
```

```
            ELSE s WITH [actual := false] ENDIF,
```

A timer (not explicitly modeled in this specification) is started each time a `SEND_MSG` operation occurs. When a timeout occurs and there is no other action possible (*i.e.*, sender's window is full and all data from it has been sent to message channel and no acknowledgments are available), then prepare to start resending from bottom of sender's window:

```
TIMEOUT: IF s`sndWinIndex = s`sndLow + sw AND s`sndIndex = s`sndWinIndex
         AND s`headAckChan = s`tailAckChan THEN
           s WITH [sndIndex := s`sndLow]
         ELSE s WITH [actual := false] ENDIF,
```

The last action of the sender consist in receiving an acknowledgement `a` (if available) from the acknowledgment channel. If `a` is in the sender's window, then we set the bottom of the sender's window `sndLow` to `a+1`, meaning that all messages with index at most `a` have been acknowledged and will never be sent again. The index `sndIndex` (of the next message to be sent) is set to the maximum between itself and `a+1`:

```
RCV_ACK(a): IF s`headAckChan>s`tailAckChan AND a=s`ackChan(s`tailAckChan)
            THEN IF a >= s`sndLow AND a < s`sndWinIndex THEN
                    s WITH[sndLow:= a+1,
                           sndIndex:= IF a>s`sndIndex THEN a+1
                                      ELSE s`sndIndex ENDIF,
                           tailAckChan := s`tailAckChan+1]
                 ELSE s WITH[tailAckChan := s`tailAckChan+1] ENDIF
            ELSE s WITH [actual := false] ENDIF,
```

The first action of the receiver is to receive a message (if available) from the message channel. The control has to allow it (`rcvControl = receiving`). If the message's index is in the window of the receiver, it is stored at the correct position:

```
RCV_MSG(m): IF s`rcvControl = receiving AND s`headMsgChan > s`tailMsgChan
             AND m = s`msgChan(s`tailMsgChan) THEN
               IF (m`index >= s`rcvLow AND m`index < s`rcvLow+rw) THEN
                  s  WITH[rcvWindow:=s`rcvWindow  WITH[(m`index):=m`data],
                      tailMsgChan:=s`tailMsgChan+1]
             ELSE s  WITH  [tailMsgChan:=s`tailMsgChan+1]  ENDIF
            ELSE s WITH [actual:=false] ENDIF,
```

The second action of the receiver is to acknowledge data. The message with index just below the receiver window's bottom is acknowledged. The control has to allow it (`rcvControl = sendingAck`), then control goes back to `receiving`:

```
SEND_ACK(a): IF s`rcvControl = sendingAck AND a = s`rcvLow-1 THEN
               s WITH[ackChan := s`ackChan WITH[(s`headAckChan) := a],
                      headAckChan := s`headAckChan+1,
                      rcvControl := receiving]
             ELSE s WITH [actual := false] ENDIF,
```

The last action of the sender is also the most complex in the whole protocol. It consists in "flushing" the receiver window to the data target. This means sending to the data target the longest contiguous sequence (starting at the bottom of the sender's window) of data items that are not equal to **Absent.** The **rcvControl** field is used to control this process. At the beginning, **rcvControl = receiving,** and if the bottom of the receiver window contains **Absent,** then the state is left unchanged, because there is nothing to flush. Otherwise, the control switches to **flushing,** and stays there to copy data from receiver's window to data target until an **Absent** data item is met, in which case control goes to **sending_ack:**

```
PUT(d) : IF (s`rcvControl = receiving OR s`rcvControl = flushing) AND
          d = s`rcvWindow(s`rcvLow) THEN
            IF d /= Absent THEN
               s WITH[rcvControl := flushing,
                      target := s`target WITH[(s`targetIndex):= d],
                      targetIndex:= s`targetIndex+1,rcvLow := s`rcvLow+1]
            ELSIF s`rcvControl =flushing THEN
               s WITH [rcvControl := sendingAck]
            ELSE s ENDIF
         ELSE s WITH [actual := false] ENDIF,
```

Finally, there are two actions of the environment: losing a message and losing an acknowledgement (if there are any available). They consist in incrementing **tailMsgChan** and **tailAckChan,** respectively.

5. VERIFYING PROPERTIES

In this section we verify safety and liveness properties of the protocol.

Safety Properties

The main safety property required from the protocol is that the sequence of data delivered to the data target is a prefix of that obtained from the data source:

```
main_safety_property: THEOREM invariant(LAMBDA (s: State):
            FORALL (i: nat): (i < s`rcvLow => s`target(i) = s`source(i)))
```

The idea for proving Theorem **main_safety_property** is to prove the equality of sequences **source, target,** etc (cf. Fig. 2) from the "outside" to the "inside". We prove, that up to a given position, the data source equals the sender's window, the data target equals the receiver's window, and that the two windows are equal. Then, the equality of these sequences imply the **main_safety_property** theorem:

- $\forall i < sndWinIndex : sndWindow(i) = source(i)$
- $\forall i < rcvLow : rcvWindow(i) = target(i)$

- $rcvLow \leq sndWinIndex$
- $\forall i < rcvLow : rcvWindow(i) = sndWindow(i)$.

It should be noted that organizing the verification as above, *i.e.*, decomposing the problem into four subproblems, is a user-dependent choice. However, once the subproblems are formulated as lemmas, the process of proving them is completely systematic. It is an invariant-strengthening process: a PVS strategy for proving inductiveness of invariants is applied. If the property is inductive, the proof succeeds, otherwise, PVS returns a number of pending subgoals. All these subgoals correspond to transitions that do not preserve the property under proof. By examining the subgoals, we formulate auxiliary invariants that, if proved, would eliminate the pending subgoals and settle the original property.

1: proving $\forall i < sndWinIndex : sndWindow(i) = source(i)$. This is expressed in PVS as Lemma `source_equals_send` below. By applying our PVS strategy for proving inductive invariants, we obtain two pending subgoals, which suggest to prove the auxiliary Lemma `source_equals_send_1`:

```
source_equals_send: LEMMA invariant(LAMBDA (s: State): FORALL (i : nat):
                      i < s`sndWinIndex => s`source(i) = s`sndWindow(i))
source_equals_send_1: LEMMA invariant(LAMBDA(s: State):
                                        s`sndWinIndex=s`sourceIndex)
```

The conjunction of the predicates in the above lemmas is inductive, and the strategy for proving inductive invariants succeeds: the step is completed.

2: proving $\forall i < rcvLow : rcvWindow(i) = target(i)$. This is similar to Step 1 and is expressed in PVS as Lemma `target_equals_receive` below. The approach for proving it also takes one invariant-strengthening step.

```
target_equals_receive: LEMMA invariant (LAMBDA(s:State): FORALL (i: nat):
                          i < s`rcvLow => s`target(i) = s`rcvWindow(i))
```

3: proving $rcvLow \leq sndWinIndex$. This step and the following are more complex and we explain them in more detail. The property is expressed as:

```
rcvLow_leq_sndWinIndex : LEMMA invariant(LAMBDA(s: State) :
                                          s`rcvLow <= s`sndWinIndex)
```

The inductiveness strategy fails on the above lemma. The pending subgoal in the PVS proof identifies the `PUT` transition of the protocol (cf. Section 4) to be responsible for the failure. This is because the `PUT` transition increments variable `rcvLow`, thus, if `rcvLow <= sndWinIndex` holds before the transition is taken, it might not hold afterwards. The subgoal also suggests how to solve the problem: by showing (1): `rcvWindow(sndWinIndex) =`

`Absent.` This is because the `PUT` transition can increase `rcvLow` and invalidate Lemma `rcvLow_leq_sndWinIndex` only when both `rcvLow = sndWinIndex` and `rcvWindow(rcvLow) /= Absent` hold before the transition. By proving (1), we prove that the latter situation cannot happen.

Now, if we formulate (1) as a lemma and try to prove it by basic invariant strengthening we run into an infinite loop. Indeed, PVS requires to prove `rcvWindow(sndWinIndex+1) = Absent, rcvWindow(sndWinIndex+2) = Absent,` etc. Noticing this pattern we realize that the actual lemma we need to prove is the "fixpoint" of these properties:

```
rcvLow_leq_sndWinIndex_1: LEMMA invariant(LAMBDA(s: State):FORALL(i:nat):
                            i >= s`sndWinIndex => s`rcvWindow(i)= Absent)
```

We apply the inductiveness strategy to prove this lemma, and fail again. The pending subgoal now points to transition `RCV_MSG` of the protocol (cf. Section 4). Indeed, this transition modifies `rcvWindow,` which could invalidate the `rcvLow_leq_sndWinIndex_1` lemma. There is one pending subgoal, which suggests to prove the property (2) `msgChan(tailMsgChan) 'index < sndWinIndex.` Indeed, in this case `rcvWindow` is modified at a position that is not referred to in the above lemma, thus, the lemma will not be invalidated if (2) holds. But attempting to prove (2) by simple invariant-strengthening actually points us to proving the more general lemma

```
rcvLow_leq_sndWinIndex_2: LEMMA invariant(LAMBDA(s: State):FORALL(i:nat):
                i < s`headMsgChan => s`msgChan(i)`index < s`sndWinIndex)
```

The inductiveness strategy succeeds in proving this lemma, and Step 3 is done.

4: proving $\forall i < rcvLow : rcvWindow(i) = sndWindow(i)$. This property is expressed as the following PVS lemma:

```
send_equals_receive: LEMMA invariant(LAMBDA(s: State): FORALL (i : nat):
                        i < s`rcvLow => s`sndWindow(i) = s`rcvWindow(i))
```

The inductiveness strategy fails to prove the lemma, and leaves two pending subgoals. The first subgoal corresponds to the `GET` transition of the protocol (cf. Section 4). This is because that transition modifies `sndWindow,` which could invalidate the lemma. The second subgoal corresponds to the `PUT` transition, which modifies `rcvLow` and could also invalidate the lemma.

To prove the first subgoal it is enough to prove (3) `rcvLow<=sndWinIndex.` Indeed, the `GET` transition modifies `sndWindow` at position `sndWinIndex,` and (3) would guarantee that this modification has no influence on our lemma. We have already proved (3) in Lemma `rcvLow_leq_sndWinIndex.`

The second subgoal suggests to prove: if (4) `rcvWindow(rcvLow) /= Absent,` then (5) `rcvWindow(rcvLow) = sndWindow(rcvLow).` This is because if (4) holds, then the `PUT` transition increments `rcvLow`, and what we have

to show is that the predicate in Lemma `send_equals_receive` holds when `i = rcvLow`, which is implied by (5).
Now, proving that (4) implies (5) by simple invariant strengthening would again lead us into an infinite loop, thus, the solution is to prove

```
send_equals_receive_1: LEMMA invariant(LAMBDA(s: State): FORALL(i: nat):
(i>=s`rcvLow  AND  s`rcvWindow(i)/=Absent)=>  s`rcvWindow(i)=s`sndWindow(i))
```

This property is not inductive, and the strategy fails again, leaving two pending subgoals. The first subgoal is due to transition `GET`, which modifies `sndWindow`, and can be settled by proving that `rcvWindow(sndWinIndex) = Absent` always holds before this transition. This was already proved in Lemma `rcvLow_leq_sndWinIndex_1`. Finally, the second subgoal is due to transition `RCV_MSG` modifies `rcvWindow`, and is settled by proving the following lemma, which is inductive and is proved automatically.

```
send_equals_receive_2:LEMMA invariant(LAMBDA(s: State): FORALL (i: nat):
   i<s`headMsgChan  =>s`msgChan(i)`data=s`sndWindow(s`msgChan(i)`index))
```

Liveness Properties

The main liveness property is that every data item from the data source is eventually delivered to the data target. If we call *fair* the runs that eventually obtain every data item from the source, and *live* the runs that eventually deliver every data item to the target, then we have to show that every fair run is live:

```
fair : PRED[(run)] = LAMBDA (r: (run)) : FORALL (m :nat): EXISTS (n:nat):
                                                    r(n)`sourceIndex > m
live : PRED[(run)] = LAMBDA (r: (run)) : FORALL (m :nat): EXISTS (n:nat):
                                                    r(n)`targetIndex > m
main_fairness_property: THEOREM FORALL(r:(run)) : fair(r) => live(r)
```

Here, `(run)` is the type of runs, defined in theory `runs.pvs` as the subtype of function from `nat` to `State` whose initial state satisfy the initial condition and such that any two consecutive states satisfy the transition relation. As for safety, the proof of Theorem `main_fairness_property` is divided into several steps and required to prove auxiliary invariants.

```
fair_aux1:  PRED[(run)]=LAMBDA  (r:(run)):  FORALL(m:nat):  EXISTS(n:nat):
                                                  r(n)`sndWinIndex > m
fairness_property_aux1:  LEMMA  FORALL(r:(run)):  fair(r)  =>  fair_aux1(r)
fair_aux2:PRED[(run)]=LAMBDA  (r:(run)):FORALL  (m:nat):  EXISTS  (n:nat):
                                                    r(n)`sndLow > m
fairness_property_aux2:  LEMMA  FORALL  (r:(run)):  fair_aux1(r)  =>
                                                       fair_aux2(r)
safe_aux2:  LEMMA  invariant  (LAMBDA(s:State):s`sndWinIndex<=s`sndLow+sw)
fair_aux3:  PRED[(run)]=LAMBDA  (r:(run)):FORALL  m:  nat):EXISTS  (n:nat):
                                                    r(n)`rcvLow > m
safe_aux3bis : LEMMA invariant (LAMBDA (s:State) : FORALL (i:nat):
                          i < s`headAckChan => s`ackChan(i) < s`rcvLow)
safe_aux3 : LEMMA invariant (LAMBDA (s: State): s`sndLow <= s`rcvLow)
```

```
fairness_property_aux3 : LEMMA FORALL (r:(run)) : fair_aux2(r) =>
                                                    fair_aux3(r)
fairness_property_aux4:LEMMA FORALL(r: (run)): fair_aux3(r) => live(r)
```

Lemmas `fairness_property_aux1` to `fairness_property_aux4` imply the main fairness theorem, and the verification is terminated.

6. CONCLUSION AND FUTURE WORK

In an attempt to demonstrate that deductive methods can be used in a systematic way to validate communication protocols, we present the verification of safety and liveness properties of a sliding-window protocol using the PVS specification and verification system. Some final remarks in favor of our thesis:

- The proofs of the safety properties are almost all the same. Indeed, most of the new proofs during the verification process were created by copy and paste, and only some quantifiers needed specific intervention to be properly instantiated. This was also true to some extent for the liveness properties,
- the strategy for proving inductive invariants suggests, in case of failure, new invariants to prove for settling the unproved subgoals. Thus, the method provides useful feedback for performing invariant strengthening,
- the new invariants did not need to be carefully analyzed and fully understood for proceeding in the proof. We have found the mechanized proof with PVS to be an automatic process, which does include the human in the loop, but does not excessively tax his patience or ingenuity.

We are currently working on techniques to further the automation of deductive verification. In particular, we are interested in a class of extended automata with integer variables and function symbols for which we study the automatic generation of auxiliary invariants and the automatic proof of inductiveness of invariants in the context of the automatically generated invariants. The class is expressive enough to cover the sliding-window protocol specification, and a large fraction of the properties presented in this paper.

Acknowledgment

Thanks to Duncan Clarke for carefully reading the manuscript and for useful comments and suggestions.

REFERENCES

[1] D. Brand and W. H. Joyner. Verification of HDLC. *IEEE Transactions on Communications* 30(5): 1136-1142, 1982.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation,* 98(2):142–170, 1992.

[3] R. Cardell-Oliver. The specification and verification of sliding-window protocols in higher order logic. Technical Report No. 183, Computer Laboratory, University of Cambridge, 1989.

[4] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual Version 6.1. Technical Report RT-0203, INRIA, July 1997.

[5] J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. *Computer-Aided Verification,* LNCS 1102, 1996.

[6] P. Godefroid and D.E. Long. Symbolic protocol verification using Queue BDDs. *Formal Methods in System Design,* 14(13):257-271, 1999.

[7] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conference on Computer-Aided Verification,* LNCS 1254, 1997.

[8] M. Gordon and T.F. Melham. *Introduction to the HOL system.* Cambridge University press, 1994.

[9] G.J. Holzmann. The model checker Spin. IEEE Transactions on Software Engineering, 23(5): 279–295, 1997.

[10] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design,* 11(2):157–185, 1997.

[11] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe,* LNCS 1051, 1996.

[12] R. Kaivola. Using compositional preorders in the verification of a sliding window protocol. *Computer-Aided Verification* LNCS 1633, 1997.

[13] Z. Manna and A. Pnueli. *Temporal verification of reactive systems.* Vol. 1: Specification, Vol. 2: Safety. Springer-Verlag, 1991 and 1995.

[14] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering,* 21(2):107-125, 1995.

[15] J. L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding-window protocol. In *Protocol Specification, Testing, and Verification, North-Holland, 1987.*

[16] M. Smith and N. Klarlund. Verification of a sliding-window protocol using IOA and MONA. In *Formal Description Techniques & Protocol Specification, Testing and Verification,* Kluwer Academic Publishing, 2000.

[17] A. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.