Compositional Verification of a Communication Protocol for a Remotely Operated Aircraft ☆

Alwyn E. Goodloe*,[a], César A. Muñoz[a]

[a] *National Aeronautics and Space Administration, Langley Research Center, Hampton, VA 23681, USA*

Abstract

This paper presents the formal specification and verification of a communication protocol between a ground station and a remotely operated aircraft. The protocol can be seen as the vertical composition of protocol layers, where each layer performs input and output message processing, and the horizontal composition of different processes concurrently inhabiting the same layer, where each process should satisfy a distinct delivery requirement. A compositional technique is used to formally prove that the protocol satisfies these requirements. Although the protocol itself is not novel, the methodology employed in its verification extends existing techniques by automating the tedious and usually cumbersome part of the proof, thereby making the iterative design process of protocols feasible.

Key words: Protocol verification, interactive theorem proving, compositional reasoning

1. Introduction

A Remotely Operated Aircraft (ROA) is a distributed system where its critical components are dispersed between the airborne vehicle and the ground station. Flight commands are sent from the ground station to the vehicle and telemetry data are sent from the aircraft to the ground station.

AirSTAR [1, 2, 3] is an experimental jet-powered ROA designed and built by NASA's Langley Research Center (LaRC) for use as a testbed for research on software health management and flight control. The operational platform of AirSTAR has a simple organization with one vehicle in the air and a single ground station, where the pilots are rarely out of visual sight of the aircraft.

☆This work was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NNX08AE37A awarded to the National Institute of Aerospace (NIA) while the authors were resident at NIA. Authors are listed in alphabetical order.

*Corresponding author.

Email addresses: a.goodloe@nasa.gov (Alwyn E. Goodloe), Cesar.A.Munoz@nasa.gov (César A. Muñoz)

When flying, commands from AirSTAR's ground-based pilot are broadcast to the aircraft and telemetry data from the aircraft are broadcast to the ground station. Aircraft commands are time sensitive in the sense that if a message is lost or corrupted in transit, then it should not be resent because it would be considered stale by the time a new copy arrives. This requirement is called the *weak delivery requirement*. On the other hand, engineers and researchers on the ground need to receive all data produced by the aircraft in order to analyze aircraft performance as well as to plan future aircraft flights. Hence, the aircraft communication protocol should guarantee that all telemetry data broadcast is eventually delivered. This requirement is called the *guaranteed delivery requirement*. This paper proposes a formally verified communication protocol for AirSTAR that satisfies these delivery requirements.

Since the requirements of weak and guaranteed delivery are orthogonal to each other, the communication protocol has been structured as two different protocols: the *weak delivery protocol* (WDP) and the *guaranteed delivery protocol* (GDP). The relationship between WDP and GDP is similar to the relationship between the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP) of the Internet protocol suite. As with UDP and TCP, WDP and GDP are multi-layered protocols. In particular, a *link layer* is considered that performs error detection and multiplexes WDP and GDP messages into the physical communication medium. However, WDP and GDP are considerably smaller and simpler than UDP and TCP. For instance, since there is only one vehicle and one ground station, WDP and GDP do not need a network layer.

This paper focuses on the *functional correctness* of WDP and GDP's delivery properties, when both protocols run in parallel and share the communication medium and the link layer. The correctness requirement for GDP is that messages are received in the order they are sent. In the case of WDP, the correctness requirement states that every message received was in the sequence of sent messages. Since both protocols share the lower layers of the protocol stack, the delivery properties have to be verified for the communication protocol formed by WDP and GDP, denoted WDP $\parallel$ GDP. This design structure introduces complexities that tax current model checking capabilities. Consequently, the verification work presented in this paper is supported by interactive theorem proving technology.

The use of a theorem prover, as opposed to a model checker, also allows for a specification that is more abstract than an implementation, but concrete enough to provide a detailed description of the design, amenable to rapid prototyping. The formal verification is carried out in the Prototype Verification System (PVS) [4]. The full development is electronically available from `http://shemesh.larc.nasa.gov/people/cam/AirSTAR`. The verification approach employs a compositional technique where the delivery properties of the communication in the composed protocol WDP $\parallel$ GDP are lifted from the delivery properties of its components. A significant percentage of these proofs are automated via PVS's proof scripting language. This promotes the iterative design of systems, as laborious proofs need not be repeated by hand at each design iteration.

The paper is structured as follows. The multi-layered architecture of the communication protocol is presented in Section 2. Section 3 describes each component of the stack except for GDP, which is detailed in Section 4. The verification of the protocol is discussed in Section 5. Section 6 discusses related works and Section 7 concludes the paper.

2. Communication Protocol Architecture

The communication protocol is structured in a *protocol stack*, where each layer handles a different aspect of message processing. As a message moves down the stack, each layer performs some processing and adds packet headers. As a message moves up the stack, the corresponding packet headers are removed. Because there is no network layer for routing, the layers of the protocol stack roughly correspond to the application layer, transport layer, link layer, and physical layer. Given that the physical layer is concerned with the details of the communication hardware, it is not modeled in this analysis. Instead, its functional behaviour is abstracted by a communication medium, which will be referred to as the *ether*.

At the top layer of the protocol stack is the *application layer*. All messages sent and received from the application layer are presumed to be sent via WDP or GDP depending on required message delivery guarantees. In other words, it is assumed that the application chooses between the WDP and GDP protocols when sending a message. The next layer down corresponds to the *transport layer* and it is here that the core of the delivery protocols reside. WDP simply sends a message, but provides no guarantee that the message ever arrived at its destination. Hence, messages may be lost or corrupted in transit and are never resent. GDP is designed to provide its user with a guarantee that any message sent is eventually received. The link layer is the next layer in the protocol stack. The delivery protocols directly interface with the link layer as there is no network layer. The *link layer* performs error detection and multiplexes the messages from the WDP and GDP layers. The ether models the communication channels over which messages are sent and received.

The proposed protocol stack is illustrated in Figure 1. The protocol stack can be explained both vertically and horizontally. Vertically, each layer performs a specific transformation on a message, adding headers as it traverses down the stack and removing headers as it traverses up the stack. Horizontally, the WDP and GDP lie at the same layer, but they behave differently as they satisfy different requirements. These may be viewed as disjoint independent components occupying the same layer in the stack and sharing the link layer. Each protocol in the stack typically has a sender and receiver process. A message processed by the sender at one node should be processed by the receiver at the destination node.

In the model of the protocol stack, the protocol layers are connected using First In First Out (FIFO) queues. This structure is illustrated in Figure 1, where each queue is depicted as a small rectangle with an arrow pointing in

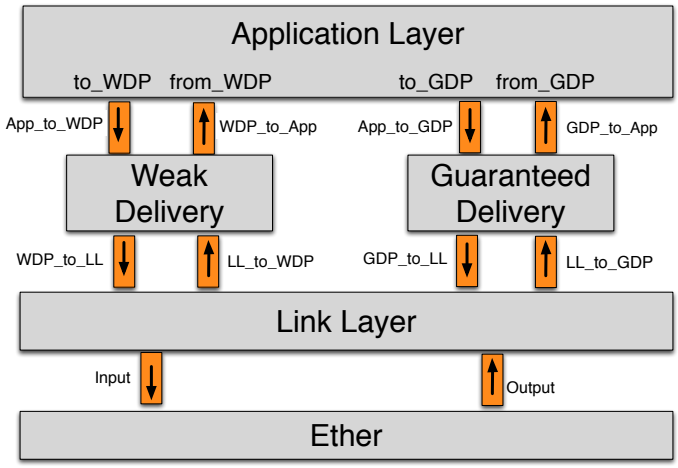

Figure 1: Protocol stack

the direction of the information flow with a label naming the queue attached. Ignoring the details of the application layer, the messages to be sent by WDP and GDP are modeled by a pair of sequences `to_GDP` and `to_WDP`. At the receiving process, the messages are placed in the pair of sequences `from_GDP` and `from_WDP`. Tracing the flow of a WDP message, a message in `to_WDP` is moved to the `App_to_WDP` queue, the weak delivery layer adds a header and places the packet on the `WDP_to_LL` queue, the link layer adds its header and moves the packet to ether's `input` queue. The ether is not a protocol layer itself, but a model of the transport medium that removes packets from the ether's `input` and placed them on the ether's `output` queue. The link layer removes the message from the output queue, strips off the outer header, performs its processing, and moves the packet onto the `LL_to_WDP` queue. The weak delivery layer removes the packet from the `LL_to_WDP` queue, removes any header, and places the packet on the `WDP_to_App` queue. The application layer removes the message from the `WDP_to_App` queue and adds it to the `from_WDP` structure.

3. Protocol Specification

The communication protocol presented in Section 2 has been specified and formally verified in the PVS specification language. First, this section gives a brief introduction to the syntax and design patterns used throughout the specification. This is followed by a description of all of the components of the specification except the GDP layer, which, given its complexity, is described in Section 4.

3.1. Specification Language

PVS is a tightly coupled specification language and interactive theorem prover. The specification language is quite expressive in that it extends classical higher-order logic with dependent types and predicate subtyping. This rich type system makes type checking undecidable, but the type checker copes with this by producing proof obligations that must be discharged using the theorem prover. PVS supports primitive types such a real numbers and Boolean values, and composed types such as abstract data types, tuples, and records. Enumeration types are considered to be a special case of abstract data types.

In this paper, abstract data type declarations take the form

$$\begin{aligned} T \quad = \quad & cons_1(a_{11} : T_{11}, \ldots, a_{1m_1} : T_{1m_1}) : rec_1? \ + \\ & \ldots \\ & cons_i(a_{i1} : T_{i1}, \ldots, a_{im_i} : T_{im_i}) : rec_i? \ + \\ & \ldots \\ & cons_n(a_{n1} : T_{n1}, \ldots, a_{nm_n} : T_{nm_n}) : rec_n?, \end{aligned}$$

where $cons_i$ is a constructor, a_{im_i} is an accessor, and $rec_i?$ is a type recognizer, which is a symbol used to branch on the type structure. Furthermore, a record type T is defined as

$$T = a_1 : T_1 \ \times \ \cdots \times a_n : T_n,$$

where a_i is a field accessor. In PVS, field access is performed using the backquote operator, so if `s` is a record and `a` is one of its fields, then `s‘a` is the value of `a` in `s`. The `with` operator overrides record fields, e.g., `s with [‘a := e, ‘b := f]` is a new record that is equal to `s` in all its fields except in `a` and `b` where it has the values `e` and `f`, respectively.

Each layer of the protocol stack is modeled by a state type, a state transition function, and a state transition relation. The state of the machine is represented by a record that consists of internal variables, e.g., counters, and input/output communication variables. The functional component is a deterministic state machine that takes the current state and an action and returns the next state. The relational component is a transition relation that nondeterministically selects a valid action for the state machine to execute.

The following naming conventions are used. The PVS type that represents the state of a machine and its relational component have the same name. This is possible since PVS supports name overloading. Furthermore, the functional component is always called `next`. As explained in Section 5.2, it is this uniformity in the structure of each component and use of name conventions that makes it possible to write strategies that automate the proofs.

For expository purposes certain details in the PVS model have been elided in the presentation of the protocols given in this paper. For instance, field names that are not need in the discussion of the protocol are left out. Moreover, in some instances, simplified variants of functions and relations are given. The reader is referred to the actual PVS development for the precise definitions of the models presented in this paper.

3.2. Ether

The ether state is specified as a pair of multisets (bags) that represent, respectively, input and output communication channels, i.e.,

$$\texttt{Ether} = \texttt{input} : \texttt{bag[LinkFrame]} \times \texttt{output} : \texttt{bag[LinkFrame]},$$

where `LinkFrame` is the type of the message frame sent over the link layer and is formally defined in Section 3.3. The specification of the ether considers the fact that messages may be duplicated, corrupted, or dropped in the physical layer or while in transit. The possible actions are defined by the datatype `EtherAction` as

```
DropIn(linkframe:LinkFrame) :  DropIn? +
DropOut(linkframe:LinkFrame) :  DropOut? +
DupIn(linkframe:LinkFrame) :  DupIn? +
DupOut(linkframe:LinkFrame):  DupOut? +
NoiseIn(linkframe:LinkFrame) :  NoiseIn? +
NoiseOut(linkframe:LinkFrame) :  NoiseOut?.
```

The ether state machine perturbs the communication medium by taking the current state and the action to perform and returns a transformed ether with a frame either corrupted, dropped, or duplicated. The PVS code for dropping, duplicating, and corrupting a frame on the inbound is given as follows.

```
next(s:Ether,a:EtherAction) : Ether =
  CASES a OF
    DropIn(linkframe) :
      s WITH [`ether`input := remove(linkframe,s`ether`input)]

    DupIn(linkframe) :
      IF member(linkframe,s`ether`input) THEN
        s WITH [`ether`input := add(linkframe,s`ether`input)]
      ELSE
        s  % No change in state
      ENDIF,

    NoiseIn(linkframe) :
      IF ¬checksum?(linkframe) THEN
        s WITH [`ether`input := add(linkframe,s`ether`input)]
      ELSE
        s  % No change in state
      ENDIF,
    ...
  ENDCASES
```

The relational component that specifies the nondeterministic behavior of the ether is defined by the predicate

$$\texttt{Ether(s,n:Ether):boolean} = \quad \texttt{n} = \texttt{s} \\ \vee\ \exists\texttt{(a:EtherAction):n = next(s,a)}.$$

3.3. Link Layer

The link layer serves as an interface between the protocol stack and the communication medium. It provides common services needed by the protocols that lie at the next higher layer and performs a check-sum error detection. Furthermore, the link layer multiplexes messages sent from the WDP and GDP layers by wrapping them in a common header, and demultiplexes them on the receiving side by removing this header and sending the unwrapped frame to the appropriate protocol for processing.

A link layer frame is composed of a check-sum and either a GDP or WDP frame:

$$\texttt{LinkFrame} = \texttt{cs: CheckSum} \times \texttt{frame: Frame},$$

where the type `Frame` can be thought of as a disjoint sum of WDP and GDP frames. The details of performing a check-sum are abstracted away. The type `Link` consists of the queues `GDP_to_LL`, `WDP_to_LL`, `LL_to_GDP`, and `LL_to_WDP`, and the ether state.

The link layer functionality is represented by a transition function that, given the current state of type `Link` and the action to perform, yields the next state, where the possible actions are: *send a message*, either WDP or GDP, and *receive a message*. If sending a message, the sate machine removes a frame from the corresponding `GDP_to_LL` or `WDP_to_LL` queue, forms a link layer frame as the product of that frame and its check-sum, and places the result in the ether's input channel. If receiving a message, a `LinkFrame` is removed from the ether's output channel, the check-sum is verified and if invalid, the packet is dropped. Otherwise, the protocol checks if the packet is a GDP or WDP frame, strips off the check-sum, and places the message on the appropriate `LL_to_GDP` or `LL_to_WDP` queue.

The predicate `Link` specifies the relation between current and next link states. As in the case of the ether machine, the nondeterministic choice of a link action is specified using an existential quantifier. Since the choice of what action to perform is made nondeterministically, there is no priority given to GDP or WDP in the case of contention when messages are sent.

3.4. Transport Layer

The transport layer consists of the Weak Delivery Protocol (WDP) and Guaranteed Delivery Protocol (GDP). For each of the WDP and GDP protocols there are sender and receiver processes.

The state of WDP sender process, which is called `WDPSender`, consists of the queue `App_to_WDP` and the link state. Sending a message is modeled as removing a message from the `App_to_WDP` queue and adding it to the `WDP_to_LL`

queue in the link state. The state of WDP receiver process, which is called `WDPReceiver`, consists of the queue `WDP_to_App` and the link state. Receiving a message is modeled as removing a message from the `LL_to_WDP` queue and adding it to the `WDP_to_App` queue in the link state.

In contrast to WDP, GDP is very complex and it is modeled after a sliding-window protocol [5]. As it is the most significant component of the communication protocol, Section 4 is dedicated to its specification.

3.5. Application Layer

The application layer consists of sender and receiver processes for both WDP and GDP messages. The process that sends GDP messages maintains an index to the next message in `to_GDP` to be sent, copies that message to `App_to_GDP`, and increments the index. The process that sends WDP messages behaves similarly by copying messages from `to_WDP` to `App_to_WDP`. The receiver processes handle the data structures `GDP_to_App`, `from_GDP`, `WDP_to_App`, and `from_WDP`.

4. Guaranteed Delivery Protocol

The Guaranteed Delivery Protocol shall satisfy the AirSTAR's guaranteed delivery requirement. The sliding-window protocol with block acknowledgment developed by Gouda [6, 7] was deemed to be the most suitable for the communication pattern in AirSTAR. The distinguishing feature of this protocol is that although each message is sent in an individual frame, the receiver replies with a single message acknowledging the receipt of a contiguous block of sequence numbers. Although an informal proof of this protocol may be found in the literature, the one presented in this paper appears to be the first formal mechanical proof of the protocol. This section contains an informal presentation of the protocol along with a small example.

4.1. Sliding Window Protocol

As indicated above, the GDP sender and receiver processes are modeled using a sliding window protocol. These processes maintain bounded buffers called *windows*. The sender window `ackd` contains the messages sent that are waiting for a block acknowledgment. The receiver window, called `rcvd`, contains the messages received but not yet delivered to the application layer.

The upper bounds on the size of the sender's and receiver's windows are called, respectively, `sw` and `rw`. Each window entry has two fields: a data field and a Boolean mask field. The `ackd` mask field is set to `false` when that message is sent and `true` when an acknowledgment is received. The `rcvd` mask field is set to `true` when a message is received. The data in the buffers may be viewed as being indexed by the positive integers, although in the actual specification some amount of machinery is needed to map an unbounded range of sequence numbers to a bounded buffer.

In order to track the buffer entry of the next message to be sent and next expected acknowledgement, the sender maintains the following indices. The

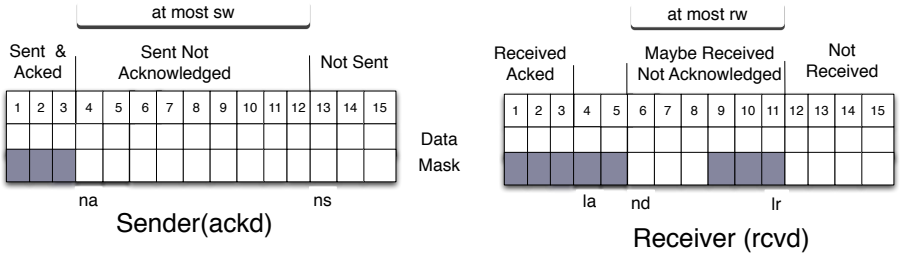

Figure 2: Sliding window

variable `ns` is an index to the sequence number of the next data item to be sent and the variable `na` is an index to the first sequence number that has yet to be acknowledged. That is, sequence numbers below `na` have all been acknowledged as received by the sender, but sequence number `na` has not yet been acknowledged. An invariant $\texttt{na} \leq \texttt{ns} \leq \texttt{na} + \texttt{sw}$ is maintained by the sender, indicating that the window of sent but not acknowledged data is of size at most `sw`. The sender will not send messages with a sequence number greater than $\texttt{na} + \texttt{sw}$ until data message `na` is acknowledged. The sender may receive acknowledgments for sequence numbers k, where $\texttt{na} \leq k < \texttt{ns}$, in any possible order; yet, only when a block acknowledgment for the contiguous sequence numbers (`na`,n), where $n < \texttt{ns}$, has been received is the value of `na` slid forward to $n + 1$. If a timeout action occurs before message `na` is acknowledged, then it is resent.

The receiver symmetrically tracks the buffer entry of the next expected message to arrive as well as sequence number of the last acknowledged message. Periodically, acknowledged messages are sent to the application layer hence the receiver also tracks the last buffer entry so delivered. The variable `nd` points to the lowest sequence number that has yet to be delivered to the application layer. The variable `lr` points the highest sequence number that has yet to be received with the constraint that $\texttt{lr} \leq \texttt{nd} + \texttt{rw}$. The receiver accepts messages for sequence numbers k, where $\texttt{nd} \leq k < \texttt{nd} + \texttt{rw}$, in any order, and ignores messages out of this range. When the receiver has received the contiguous block of sequence numbers (`nd`,n), the index `nd` is slid forward to $n + 1$ and the corresponding messages are delivered to the application layer. The variable `la` points to the last acknowledged sequence number, i.e., messages with a sequence number below `la` have all been acknowledged. Note that messages with a sequence number m, where $\texttt{la} \leq m \leq \texttt{nd} - 1$, have been received and delivered, but not yet acknowledged. Periodically, GDP sends the block acknowledgment for sequence numbers $(\texttt{la}, m - 1)$ and `la` is reset to `nd`.

The execution of the protocol is illustrated using Figure 2. In order to not clutter the figure, data fields are left blank and `true` values in the mask field are shaded while `false` entries are not. According to the figure, the sequence numbers $1, 2, 3$ have been sent and acknowledged. The sequence numbers $4, \ldots, 12$ have been sent, but not necessarily delivered to the receiver. Furthermore, the

sequence numbers 4 and 5 have been sent and received, but not yet acknowledged; $9, 10, 11$ are delivered; and $6, 7, 8$ are lost in transit. At some point the sender times out and resends these messages. When sequence numbers $4, \ldots, 11$ have been received by the receiver and delivered to the application layer, the index nd slides forward to 12. When the acknowledgment message is sent by the receiver acknowledging the receipt of the sequence numbers $4, \ldots, 11$, the receiver sets `la` to 12. If the acknowledgment is lost in transit, the sequence numbers $4, \ldots, 11$ would eventually be resent and, although the messages would be ignored by the receiver process, acknowledgments would be resent.

4.2. GDP Sender

The sender process implements the procedure for the sliding window protocol sender outlined above. The internal state of the sender's window with its corresponding indices is defined using the record type

$$\begin{aligned} \texttt{WinSenderPrivate} \;=\; & \texttt{na} : \texttt{nat} \times \\ & \texttt{ns} : \texttt{subrange(na,na+sw)} \;\times \\ & \texttt{ackd} : \texttt{maxwindow?(ns-na)}, \end{aligned}$$

where the dependent type system of PVS is used to encode invariant relations between the fields. The type `maxwindow?(ns-na)` specifies that the window has maximum size `ns-na`. Thus, bounding the maximum index into the buffer.

The state of the sender is defined as

$$\begin{aligned} \texttt{GDPSender} \;=\; & \texttt{App\_to\_GDP} : \texttt{fifo[Data]} \times \\ & \texttt{winsender} : \texttt{WinSenderPrivate} \times \\ & \texttt{link} : \texttt{Link}, \end{aligned}$$

forming a tuple of the queue holding data to be sent, the internal indices and buffers of the sender side of the sliding-window protocol, and the link state, which includes the ether state.

The sender process is modeled by the transition relation `GDPSender`, which relates the current state of the GDP sender process and a possible next state. It is formally defined as follows.

$$\begin{aligned} \texttt{GDPSender(s,n:GDPSender)} = \; & \texttt{Link(s`link,n`link)} \wedge \ldots \\ & \vee\ \texttt{n} = \texttt{s} \\ & \vee\ \exists\texttt{(a:WinSenderAction)} :\ \ \texttt{n = next(s,a)}, \end{aligned}$$

where the unspecified parts in this definition state that the fields that are not modified by transitions remain unchanged. The possible actions performed by the sender are *send* a message, *process* an acknowledgment, and *timeout* due to the fact that an acknowledgment has not been received in a predefined time. The actions are formally defined by the enumeration type:

$$\texttt{WinSenderAction} = \{\texttt{Send},\ \ \texttt{GetAck},\ \ \texttt{Timeout}\}.$$

The following describes the sender transition to the next state. In each case, the next state is the same as the current state except for the changes described below.

- `Send`. If there is data to be sent and there is space in the `ackd` buffer, i.e.,

$$\neg\ \texttt{empty\_fifo?(App\_to\_GDP)}\ \wedge\ \texttt{ns} - \texttt{na} < \texttt{sw},$$

  then the following changes take place:

  - The data is removed from the `App_to_GDP` queue.
  - A GDP frame is formed from the value removed from `App_to_GDP` and the sequence number `ns` and added to the `GDP_to_LL` queue to send it to the link layer for further processing.
  - The index `ns` is incremented by 1.
  - The window `ackd` is updated, i.e., its length is incremented by 1, `data`(n) is assigned the value of the data removed from the queue and `mask`(n) is assigned the value `false`, where n is the current length of the buffer.

- `GetAck`. An acknowledgment message contains two fields, `lb` and `ub`, that denote the lower bound and upper bound on the sequence numbers being acknowledged. If the message being acknowledged falls outside of the window, i.e., $\texttt{lb} < \texttt{na} \vee \texttt{ub} \geq \texttt{ns}$, then ignore the message removing it from the `LL_to_GDP` queue. If $\texttt{na} \leq \texttt{lb} \wedge \texttt{ub} < \texttt{ns}$, then remove the acknowledgment message from `LL_to_GDP`. The `ackd` mask entries $\texttt{lb} - \texttt{na}, \ldots, \texttt{ub} - \texttt{na}$ are set to `true`. The function `slide` is then invoked to update `ackd` by sliding the index of the next acknowledged frame, e.g., `na`, forward.

- `Timeout`. If a timeout has occurred and $\texttt{ns} > \texttt{na}$, then retransmit the message with the sequence number `na`.

4.3. GDP Receiver

The receiver process implements the procedure for the sliding window protocol receiver and is somewhat symmetric to the sender. The internal state of the receiver's window with its corresponding indices is defined using the record type

$$\begin{aligned}\texttt{WinReceiverPrivate} \;=\; & \texttt{nd} : \texttt{nat} \times \\ & \texttt{la} : \texttt{upto(nd)} \times \\ & \texttt{lr} : \texttt{subrange(nd,nd+rw)} \times \\ & \texttt{rcvd} : \texttt{max\_window?(lr-nd)},\end{aligned}$$

where, as with the sender, the dependent type system of PVS encodes the invariant relationships given in Section 4.1.

The state of the GDP receiver is defined as

$$\begin{aligned} \texttt{GDPReceiver} \;=\; & \texttt{GDP\_to\_App} : \texttt{fifo[Data]} \times \\ & \texttt{winreceiver} : \texttt{WinReceiverPrivate} \times \\ & \texttt{link} : \texttt{Link}, \end{aligned}$$

forming a tuple of the queue holding data that has been received, the internal indices and buffers of the receiver side of the sliding-window protocol, and the link state, which includes the ether state.

The receiver process is modeled by the transition relation `GDPReceiver`, which relates the current state of the GDP receiver and a possible next state. It is formally defined as follows.

$$\begin{aligned} \texttt{GDPReceiver(s,n:GDPReceiver)} = \texttt{Link(s`link,n`link)} \wedge \ldots \\ \vee\ \texttt{n} = \texttt{s} \\ \vee\ \exists\texttt{(a:WinReceiverAction)} :\ \ \texttt{n = next(s,a)}, \end{aligned}$$

where the unspecified parts in this definition state that the fields that are not modified by transitions remain unchanged. The possible actions performed by the receiver are *process* a message and *send* an acknowledgement. The actions are formally defined by the enumeration type:

$$\texttt{WinReceiverAction} = \{\texttt{Receive},\ \ \texttt{SendAck}\}.$$

The following describes the receiver transition to the next state. In each case, the next state is the same as the current state except for the changes described below.

- `Receive`. If the message on the top of the `LL_to_GDP` queue is not a data message, then nothing is changed. Otherwise, let i be the value of this message's sequence number. Depending on the value of i, the protocol takes the following action:
  - If $i \geq \texttt{nd} + \texttt{rw} \ \vee\ \texttt{la} \leq i \ \wedge\ i < \texttt{nd}$, then the message is outside of the receiver's window and is removed from the `LL_to_GDP` and discarded.
  - If $i < \texttt{la}$, which means that the message has already been acknowledged, but for whatever reason the sender has resent it, then send an acknowledgment back. That is, the message is removed from the `LL_to_GDP` queue and discarded and an acknowledgment added to the `GDP_to_LL` queue.
  - If $\texttt{nd} \leq i < \texttt{nd} + \texttt{rw}$, then the sequence number is within the window and so the data is placed in the `rcvd` buffer and the mask set to true. The following state changes then take place:
    - The message is removed from the `LL_to_GDP` queue.

* The function `deliver` is invoked that places the data in `rcvd` from `nd` to n on the `GDP_to_App` queue, where `nd`$\ldots n$ is a contiguous block of data that has been received, but not yet delivered to the application layer. The value of `nd` is set to $n+1$.
* The value of `nd` is set to the index of the first mask in `rcvd` that is false.
* The value of `lr` is set to the maximum between `lr` and $i+1$.
* The function `slide` is invoked to update `rcvd` by sliding the index of the next delivered frame, e.g. `nd`, forward.

- `SendAck`. If `nd` $>$ `la`, then form an acknowledgment message for the indices `la`$, \ldots,$ `nd` $- 1$. The next state is the same as the current state except that
  - The value of `la` is set to `nd`.
  - The new acknowledgment message is added to the `GDP_to_LL` queue.

5. Protocol Verification

This paper focuses on the analysis of functional correctness of the WDP and GDP protocols in the PVS theorem prover.

Functional correctness of protocols is usually expressed by *safety properties*, i.e., predicates that hold in every reachable state of a state transition system. In PVS, transition systems can be modeled using the general theories defined in [8]. In that paper, a transition system $\mathcal{T}$ is a triple $(S, I, \rightarrow)$, where S is an abstract type, I is an initial set of elements of type S, and $\rightarrow \subseteq \ S \times S$ is a transition relation of elements of type S. The set R^n denotes the states that are reachable in exactly n steps. It is inductively defined as $R^0 = I$ and $R^{n+1} = \{s' \in S \mid (\exists (s : S) : s \in R^n \ \wedge \ s \rightarrow s'\}$. A possibly infinite sequence of states $s_0, \ldots, s_n, \ldots$ is called a *trace* if $s_i \in R_i$ for all i. The PVS type `Run` consists of all traces of a given transition system. A predicate P on S is an *invariant* of $\mathcal{T}$ if P holds on any reachable state. Formally,

$$\texttt{invariant}(P) \ = \ \forall (s : S, n : \texttt{nat}) : s \in R^n \Longrightarrow P(s).$$

5.1. WDP ‖ GDP

For the purpose of this verification, a system of two distributed nodes is considered, one of which is the sender and the other is the receiver. The two nodes interact only trough the ether. That system, denoted WDP ‖ GDP, is illustrated in Figure 3.

The state of WDP ‖ GDP is the union of `AppWDPSender`, `AppWDPReceiver`, `AppGDPSender`, `AppGDPReceiver`, `WDPSender`, `WDPReceiver`, `GDPSender`, and `GDPReceiver`, where (1) the sender and receiver processes each share the link layer, and (2) the input and output channels of the ether state in the sender are connected, respectively, to the output and input channels of the ether state

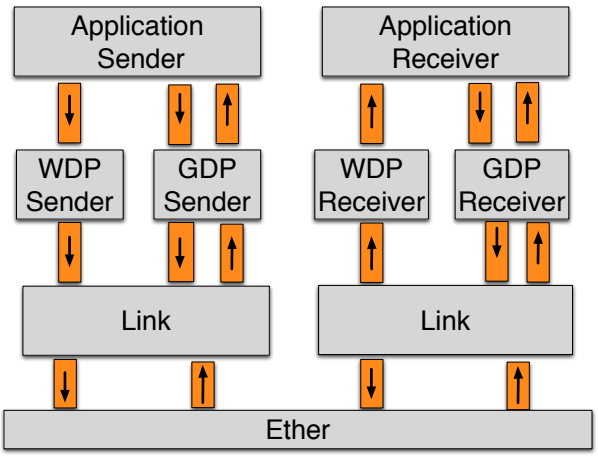

Figure 3: System WDP ‖ GDP

in the receiver. The predicate `WDP_GDP` represents the transition relation of the system. It is defined as the disjunction of the transition relations `AppWDPSender`, `AppWDPReceiver`, `AppGDPSender`, `AppGDPReceiver`, `WDPSender`, `WDPReceiver`, `GDPSender`, and `GDPReceiver`, where fields that are not modified by transitions remain unchanged.

The initial states of WDP ‖ GDP have all indices set to 0. Furthermore, queues and sequences are empty, except for the sequences `to_WDP` and `to_GDP` that have arbitrary values representing, respectively, the WDP and GDP messages to be sent by the sender process.

The predicates that express the weak delivery and guaranteed delivery requirements of WDP ‖ GDP are defined, respectively, by

$$\texttt{WDP\_sound?(s:WDP\_GDP)} = \texttt{s`from\_WDP} \subseteq \texttt{s`to\_WDP},$$
$$\texttt{GDP\_sound?(s:WDP\_GDP)} = \texttt{s`from\_GDP} \preceq \texttt{s`to\_GDP},$$

where $\preceq$ is the prefix relation between sequences. The predicate `WDP_sound?` states that all WDP messages that the receiver node delivers to the application layer were indeed sent by the sender's application layer. The predicate `GDP_sound?` states that GDP messages are delivered by the receiver to the application layer in the same order as they were sent by the sender's application layer.

The primary verification objective of this work is to formally prove that the predicates `WDP_sound?` and `GDP_sound?` are invariant properties of the system WDP ‖ GDP. Proving invariants of transition systems is considered routine in the theorem proving community. However, for the case of WDP ‖ GDP, the problem is made harder by the fact that the transition relation `WDP_GDP` is formed from the disjunction of several nested relations representing lower layers

of the stack. Since invariants must be shown to hold under each transition in each layer, each proof requires the discharge of a large number of cases. As it will be shown in the following sections, this complexity can be mitigated by the use of proof decomposition and proof automation.

The system WDP ‖ GDP can be seen as the asynchronous composition of two systems, namely WDP and GDP, that share common resources such as the link layer and the communication medium. Each one of the systems WDP and GDP consists of two nodes, representing a sender and a receiver process. The state of WDP is the projection of the sate of `WDP_GDP` to the fields that are relevant to WDP. The transition relation of WDP, called `WDP`, is defined by the disjunction of the relations `AppWDPSender`, `AppWDPReceiver`, `WDPSender`, and `WDPReceiver`. The system GDP is defined in a similar way.

When each protocol is considered independently, it can be verified that the predicates `WDP_sound?` and `GDP_sound?` are invariants of WDP and GDP, respectively. Section 5.2 shows how these verifications are highly automated via the development of proof strategies. A verification approach is proposed in Section 5.3 that, under certain conditions, guarantees that a predicate is an invariant of a composed system, if the predicate is an invariant of one of its components. This approach is used in Section 5.4 to prove that `WDP_sound?` and `GDP_sound?` are also invariants of WDP ‖ GDP. In the case of GDP, a liveness property is also considered. That property states that all messages are eventually delivered to the receiver under a fairness assumption on the sender side. Since that proof straightforwardly follows a similar proof in [8], it is not detailed in this paper. However, it is included in the formal development and briefly discussed in Section 5.5.

5.2. Proving Invariant Properties of WDP and GDP

Deductive proofs of invariant properties of a transition system typically proceed by natural induction on the length of the system traces. These inductive proofs often require the transformation of the invariant properties to a form that can be proved by induction. However, not all invariants properties of WDP and GDP require sophisticated proofs. There are many relationships that are local to either the sender process or the receiver process. For instance, the property stating that the index of the next message to be sent is greater than or equal to the index of the next message waiting to be acknowledged, i.e., $\texttt{na} \leq \texttt{ns}$, only concerns the sender, and the property that states that the index of the next message to be delivered to the application layer is greater than or equal to the index of the last message to be acknowledged, i.e., $\texttt{la} \leq \texttt{nd}$, only concerns the receiver. As these properties can be described solely in terms of the state of one of the processes, they can be easily encoded using the PVS's subtype and dependent type system. These type annotations generate proof obligations that are automatically discharged by the theorem prover.

Safety properties related to the interaction between the sender and receiver processes cannot be statically specified using the type system. These kinds of properties require more complex reasoning. In these cases, the nontrivial task of finding auxiliary invariants that enable the inductive proof of a property is

subject to the ingenuity of the human prover. In this paper, a set of proof strategies have been developed that automate the verification of invariant properties of GDP and WDP. Although these strategies are used in the context of GDP and WDP, they can be adapted to work on discrete transition systems that are based on Rusu's PVS theories [8].

In PVS, strategies, which are called tactics in other theorem provers, are written in a restricted Common Lisp language [9] that preserves the soundness of the theorem prover. These strategies enable the definition of advanced logical combinators based on other strategies and the basic deduction rules of PVS. The strategy code is 1161 lines long and can be found as part of this formal development at the aforementioned web site.

In developing the PVS models of the WDP and GDP, certain syntactical and structural conventions have been followed for naming and defining the protocol components. For instance, the definition of each transition relation T has the form

$$\begin{aligned} T(s, n : T) : \texttt{boolean} = \quad & T_1(s_1, n_1) \ \wedge \ \hat{s_1} = \hat{n_1} \\ & \vee \ \ldots \\ & \vee \ T_m(s_m, n_m) \ \wedge \ \hat{s_m} = \hat{n_m} \\ & \vee \ s = n \\ & \vee \exists (a : T\texttt{Action}) : n = \texttt{next}(s, a), \end{aligned}$$

where $m \geq 0$. Furthermore, for $1 \leq i \leq m$, s_i and n_i represent, respectively, the parts of s and n that are relevant to the transition T_i, and $\hat{s_i}$ and $\hat{n_i}$ represent the complements of s_i and n_i, respectively. The case $s = n$ guarantees that the transition relations considered here are reflexive, i.e., the transition systems are closed under stuttering. These notions will be formalized in Section 5.3.

The family of strategies `unroll-`T, where T is a transition relation, takes advantage of the modular definition of T. Each one of these strategies unfolds the definition of T by expanding the definition of `next` and recursively calling `unroll-`T_i, for $1 \leq i \leq m$. Since actions in T are specified using a case constructor, when splitting the case, the strategy `unroll-`T comments each branch in the case with the action name. These strategies are building blocks of more sophisticated strategies. They are also useful in interactive proofs as they relieve the user from expanding all the numerous cases by hand. The comments that are added by the strategies will assist the user in recognizing each particular case. Each strategy `unroll-`T is defined according to the following algorithm:

1. Expand the definition of T.
2. Expand the definition of `next` and comment each case with the name of the action.
3. Split disjunction.
4. For $1 \leq i \leq m$ perform `unroll-`T_i.

As many other protocols, WDP and GDP use data structures such as FIFO queues, bags, and finite sequences. The strategy `simp-inv` simplifies a given

proposition by applying several properties of these structures in addition to standard PVS decision procedures.

A general strategy for proving invariant properties of transition systems is called `discharge-inv`. This strategy tries to prove propositions of the form `invariant`(P), where P is a predicate on T, by using the induction principle defined in [8] along with the strategies `simp-inv` and `unroll-`T. It is defined according to the following algorithm:

1. Instantiate the induction principle in [8] as appropriate.
2. In the base case, apply `simp-inv`.
3. In the inductive case:
   - Apply `unroll-`T.
   - Apply `simp-inv` to each one of the cases generated in the previous step.

After applying `discharge-inv`, most cases are automatically discharged. The remaining unproven cases provide enough information to assist the developer in finding auxiliary invariants that get applied to complete the proof.

One of PVS notable weaknesses is the matching algorithm for higher-order functions, such as invariant predicates. The simple strategy `use-inv` takes advantage of the syntactical conventions in the definition of transition relations to find an appropriate instantiation for an inductive lemma that is passed as a parameter.

The proofs of WDP and GDP soundness proceed by induction on the traces, but complexities arise due to the protocol stack. In fact, the structure of both proofs mimics the stack structure as the property must be shown to hold at each layer. A pattern emerges where the proof of an invariant at one layer requires an auxiliary invariant, where the initial invariant is applied to one level below. This pattern can be seen in the structure of the proof of WDP soundness. In order to show

$$\texttt{s`from\_WDP} \subseteq \texttt{s`to\_WDP},$$

the property must be shown to hold the next layer down:

$$\texttt{s`App\_to\_WDP} \subseteq \texttt{s`to\_WDP}.$$

This pattern repeats going down and back up the stack proving the following invariants:

- All messages in `WDP_to_LL` are in `to_WDP`.
- All WDP messages in the ether are in `to_WDP`.
- All messages in `LL_to_WDP` are in `to_WDP`.
- All messages in `WDP_to_App` are in `to_WDP`.

The proof of GDP soundness follows a similar pattern, but additional invariants are needed as described below. Discovering all the of required auxiliary invariants is difficult when carrying out the proof by hand due to the complexity of the model. Discharging a similar invariant at each layer of the sender and receiver's stack leads to a great deal of repetitiveness. The proposed strategies automate this process as much as possible.

Armed with the strategies described above, the following theorems are proved in a highly automated way.

Theorem 1. *The proposition* `invariant(WDP_sound?)` *holds in WDP.*

Sketch of PVS Proof. The proof proceeds by induction on the length of the WDP traces. After applying the strategy `discharge-inv`, two cases remain. The first case comes from the transition relation `WDPReceiver` and states that WDP frames ready to be delivered by the transport layer to the application layer appear in `to_WDP`. This property is an invariant of the system and is proved by induction using `discharge-inv`. Once this invariant property is proved, the case is discharged by the strategy `use-inv`. The second case comes from the transition relation `AppWDPSender` and simply states that the sequence `to_WDP` does not lose messages. It is easily proved by unfolding a definition. □

Theorem 2. *The proposition* `invariant(GDP_sound?)` *holds in GDP.*

Sketch of PVS Proof. The proof proceeds by induction on the length of the GDP traces. After applying the strategy `discharge-inv`, two cases remain. The first case comes from the transition relation `AppGDPSender` and states that the sequences `to_GDP` do not decrease in size. It can be easily proved by unfolding a definition. The second case comes from the transition relation `GDPReceiver` and states that the sequence of GDP frames delivered by the transport layer to the application layer appear in the same order in `to_GDP`. This property is an invariant of the system and is proved by induction using `discharge-inv`. Once this invariant property is proved, the case is discharged by the strategy `use-inv` and the expansion of some definitions. □

As illustrated by the proofs above, the lemmas that are necessary to prove the first case in Theorem 1 and the second case in Theorem 2 are suggested by the strategy `discharge-inv`. In the process of discharging these lemmas using the strategies, some nontrivial invariant properties are discovered. In particular, the following properties are first discovered by `discharge-inv` and then proved to hold in GDP.

- The application layer pointer to the next GDP frame to send is equal to the counter of sent messages plus the number of frames in the queue.
- Both the queue connecting the application layer to the GDP layer and `ackd` are segments of `to_GDP`.
- The counter of received messages is less than or equal to the counter of sent messages, i.e., `lr` $\leq$ `ns`.

- The counter of delivered messages is less than or equal to the counter of sent messages, i.e., $\texttt{nd} \leq \texttt{ns}$.

- The largest sequence number for which an acknowledgment has been received is less than or equal to the counter of the sent acknowledgments

$$\texttt{na} + \texttt{last\_true}(\texttt{ackd}) \leq \texttt{la},$$

  where the function `last_true` returns the difference between `na` and the largest sequence number for which an acknowledgment has been received.

5.3. Proving Invariants of Composed Systems

In the previous section, it has been proved that `WDP_sound?` is an invariant of WDP and that `GDP_sound?` is an invariant of GDP. However, the verification goal is to show that both of them are also invariants of WDP ‖ GDP. This goal could be trivially achieved if WDP and GDP were completely independent. They are not. The GDP and WDP sender and receiver processes share the same link layer and ether interfaces. It could be possible to prove that `WDP_sound?` and `GDP_sound?` are invariants of WDP ‖ GDP using the strategies described in the previous section. However, this approach does not profit from the invariants that have been already proved for WDP and GDP, and therefore they have to be proved again for WDP ‖ GDP.

In this paper, a different approach is proposed. Instead of reproving all the invariants, a general theory of asynchronous composition of transition systems is developed in PVS, where invariants of a composed system are *lifted* from invariants of its components. To this end, it is considered that the state of a transition system is a tuple denoted $[P \mid S]$, where P is a private state and S is a shared state. Given two transition systems $\mathcal{T}_1 = ([P_1 \mid S], I_1, \rightarrow_{\mathcal{T}_1})$ and $\mathcal{T}_2 = ([P_2 \mid S], I_2, \rightarrow_{\mathcal{T}_2})$, define $\mathcal{T}_1 \parallel \mathcal{T}_2$ as follows. The state of the composed system has a copy of the private states of each transition system, but only one of the shared state, i.e., $S_{\mathcal{T}1\parallel\mathcal{T}2} = [P_1 \times P2 \mid S]$. The initial state of $\mathcal{T}_1 \parallel \mathcal{T}_2$ is defined as

$$I_{\mathcal{T}_1\parallel\mathcal{T}_2} \;=\; \{s : S_{\mathcal{T}_1\parallel\mathcal{T}_2} \mid s{\downarrow}_{\mathcal{T}_1} \in I_{\mathcal{T}_1} \wedge s{\downarrow}_{\mathcal{T}_2} \in I_{\mathcal{T}_2}\},$$

where the restriction operators $s{\downarrow}_{\mathcal{T}_i}$ and $s{\downarrow}_{[\mathcal{T}_i]}$, for $i = \{1, 2\}$, are defined such that the first operator projects the composed state to the state of $\mathcal{T}_i$, which only includes the private and shared state of $\mathcal{T}_i$, and the second operator only projects the private part of $\mathcal{T}_i$. The transition relation of the composed system is defined as

$$\begin{aligned} s \rightarrow_{\mathcal{T}_1\parallel\mathcal{T}_2} s' = \quad & \{(s, s') : S_{\mathcal{T}_1\parallel\mathcal{T}_2} \times S_{\mathcal{T}_1\parallel\mathcal{T}_2} \mid s{\downarrow}_{\mathcal{T}_1} \rightarrow_{\mathcal{T}_1} s'{\downarrow}_{\mathcal{T}_1} \wedge s{\downarrow}_{[\mathcal{T}_2]} = s'{\downarrow}_{[\mathcal{T}_2]} \\ & \vee \; s{\downarrow}_{\mathcal{T}_2} \rightarrow_{\mathcal{T}_2} s'{\downarrow}_{\mathcal{T}_2} \wedge s{\downarrow}_{[\mathcal{T}_1]} = s'{\downarrow}_{[\mathcal{T}_1]}\}. \end{aligned}$$

An *abstraction* α of a transition system $\mathcal{T}$ is a function that maps states into states such that

1. if s_0 is an initial state in $\mathcal{T}$, then $\alpha(s_0)$ is also an initial state of $\mathcal{T}$, and

2. if $s_n \rightarrow_{\mathcal{T}} s_{n+1}$ then $\alpha(s_n) \rightarrow_{\mathcal{T}} \alpha(s_{n+1})$.

Furthermore, α is said to be *fixed under* a predicate P if

$$P(\alpha(s{\downarrow_{\mathcal{T}}})) \Longrightarrow P(s{\downarrow_{\mathcal{T}}}).$$

Let $\mathcal{T}$ be a composed transition system of $\mathcal{T}_1$ and $\mathcal{T}_2$, and α be an abstraction of $\mathcal{T}_1$. The transition system $\mathcal{T}_2$ *does not interfere with* α if for all $s_n, s_{n+1} \in S_{\mathcal{T}}$,

$$s_n{\downarrow_{\mathcal{T}_2}} \rightarrow_{\mathcal{T}_2} s_{n+1}{\downarrow_{\mathcal{T}_2}} \implies \alpha(s_n{\downarrow_{\mathcal{T}_1}}) \rightarrow_{\mathcal{T}_1} \alpha(s_{n+1}{\downarrow_{\mathcal{T}_1}}).$$

The following theorem, which is formally proved in PVS, is sufficient to prove that an invariant of one side of the parallel operator is also an invariant of the composed system.

Theorem 3. *Let $\mathcal{T}$ be a composed transition system of $\mathcal{T}_1$ and $\mathcal{T}_2$, and P be an invariant of $\mathcal{T}_1$. The predicate $P_{\mathcal{T}}$, where $P_{\mathcal{T}}(s : S_{\mathcal{T}}) = P(s{\downarrow_{\mathcal{T}_1}})$, is an invariant of the transition system $\mathcal{T}$ if there is an* abstraction α *of $\mathcal{T}_1$ fixed under P such that $\mathcal{T}_2$ does not interfere with α.*

Sketch of PVS Proof. Consider an arbitrary trace $s_0, \ldots, s_n$ in $\mathcal{T}$. It is shown that P holds in s_n. First, it is shown that $\alpha(s_0{\downarrow_{\mathcal{T}_1}}), \ldots, \alpha(s_n{\downarrow_{\mathcal{T}_1}})$ is a trace in $\mathcal{T}_1$. There are two cases:

1. The transition (s_i, s_{i+1}) is a transition in $\mathcal{T}_1$. In this case, $\alpha(s_i{\downarrow_{\mathcal{T}_1}}) \rightarrow_{\mathcal{T}_1} \alpha(s_{i+1}{\downarrow_{\mathcal{T}_1}})$ since α is an abstraction of $\mathcal{T}_1$.
2. The transition (s_i, s_{i+1}) is a transition in $\mathcal{T}_2$. In this case, $\alpha(s_i{\downarrow_{\mathcal{T}_1}}) \rightarrow_{\mathcal{T}_1} \alpha(s_{i+1}{\downarrow_{\mathcal{T}_1}})$ since $\mathcal{T}_2$ does not interfere with α.

Therefore, $\alpha(s_0{\downarrow_{\mathcal{T}_1}}), \ldots, \alpha(s_n{\downarrow_{\mathcal{T}_1}})$ is a trace in $\mathcal{T}_1$. Since P is an invariant of $\mathcal{T}_1$, P holds in $\alpha(s_i{\downarrow_{\mathcal{T}_1}})$, for $i \leq n$. Since α is fixed under P, P holds in $s_i{\downarrow_{\mathcal{T}_1}}$ as well. The result then follows from the fact that $P_{\mathcal{T}}(s_i)$ is defined as $P(s_i{\downarrow_{\mathcal{T}_1}})$. □

Theorem 3 is stated in an abstract setting. In order to use this theorem on given transitions systems $\mathcal{T}_1$ and $\mathcal{T}_2$, an abstract function α that satisfies the hypotheses of the theorem must be provided. That function represents a projection of the combined state that keeps the information that is important for the validity of the invariant on $\mathcal{T}_1$, i.e., α is fixed under P, and that erases the information that is only relevant to $\mathcal{T}_2$, i.e., $\mathcal{T}_2$ does not interfere with α. It is usually the case, at least in the protocols presented in this paper, that when $s{\downarrow_{T_2}} \rightarrow_{\mathcal{T}_2} n{\downarrow_{T_2}}$, $\alpha(s{\downarrow_{T_1}})$ and $\alpha(n{\downarrow_{T_1}})$ are identical. Therefore, in order to get $\alpha(s{\downarrow_{T_1}}) \rightarrow_{\mathcal{T}_1} \alpha(n{\downarrow_{T_1}})$ is necessary for the transition relation in $\mathcal{T}_1$ to be reflexive. For this reason, all the transition relations in this paper are defined closed under stuttering. However, it should be noted that Theorem 3 does not require reflexivity of the transition relation in $\mathcal{T}_1$ (or $\mathcal{T}_2$).

5.4. WDP ‖ GDP is Sound

For the case of the distributed system WDP ‖ GDP, the queues `App-to-WDP` and `WDP-to-App` are private to `WDP`. The sequences `to_WDP` and `from_WDP` reside in the application layer. However, for analytical purposes they can be seen as belonging to WDP since they are not shared in any way with the GDP processes. The queues `App-to-GDP` and `GDP-to-App` as well as the fields `winsender` and `winreceiver` are private to `GDP`. All the other fields, i.e., the link and the ether interfaces, are shared. It should be noted that although these structures are shared, it is not like classical shared variable concurrency in the sense that the WDP and GDP processes do not share variables to which they both read and write. Instead, the shared structures provide a service to the WDP and GDP layers, but by design, the frames written by one higher-layer protocol will never be transformed into frames from a different layer protocol and frames written by a higher-layer protocol will never be delivered to a different higher-layer protocol.

Theorem 4 (WDP ‖ GDP Soundness)**.** *The protocol WDP ‖ GDP satisfies the weak and guarantee delivery properties, i.e.,* `invariant(WDP_sound?)` *and* `invariant(GDP_sound?)` *hold in WDP ‖ GDP.*

Sketch of PVS Proof. For the first proposition, the abstraction that is needed is a filter that removes GDP packets from the link layer and the ether interface. The abstraction $\alpha_w(s : \texttt{WDP})$ is defined such that $\alpha_w(s) = s$ in all fields but:

$$\begin{aligned}
\alpha_w(s\texttt{`link`GDP_to_Link}) &= \texttt{empty},\\
\alpha_w(s\texttt{`link`Link_to_GDP}) &= \texttt{empty},\\
\alpha_w(s\texttt{`ether`input}) &= \texttt{remove_gdp}(s\texttt{`ether`input}),\\
\alpha_w(s\texttt{`ether`output}) &= \texttt{remove_gdp}(s\texttt{`ether`output}),
\end{aligned}$$

where `empty` is the empty queue and `remove_gdp` removes all GDP frames from a multiset. Then, it is proved that α_w is an abstraction of `WDP` fixed under `WDP_sound?` and that `GDP` does not interfere with α_w. By Theorem 1, the invariant `WDP_sound?` holds on WDP. Therefore, by Theorem 3, `WDP_sound?` is also an invariant of WDP ‖ GDP. For the second proposition, the abstraction that is needed in this case is a filter that removes WDP packets from the link layer and the ether interface. The proof is similar to the first case but uses Theorem 2. As noted in Section 5.3, in order to use Theorem 3 to lift the invariants in Theorem 1 and Theorem 2, it is necessary to prove that the transition relations `WDP` and `GDP` are reflexive, which they are trivially so. □

The hypotheses of this theorem are automatically discharged by strategies that have been developed to prove that a given function is an abstraction, that an abstraction is fixed under an invariant, and that a transition relation does not interfere with an abstraction. These strategies also use the family of `unroll-`T strategies discussed in Section 5.2, but they are considerable more specific to WDP and GDP. Hence, they are less prone to generalization.

5.5. GDP Liveness

In addition to the safety properties given above, it has been shown that GDP satisfies the liveness property for sliding window-protocols given in Rusu [8]. Fair runs of the protocol are defined as those that eventually obtain all the data from the sender and live runs of the protocol are defined as those that eventually deliver all sent data to the receiver. Formalizing these properties amounts to instantiating Rusu's PVS definitions with the data structures defined in this paper. GDP's layered structure introduces a number of data structures that must be accounted for.

The definition of the fairness predicate given in [8] states that in any trace of the protocol, for every sequence number m there exists a state in the trace where $\texttt{ns} > m$. This is stated formally as follows:

$$\texttt{fair}(r : \texttt{Run}) = \ \forall(m : \texttt{nat}) : \ \exists(n : \texttt{nat}) : r_n\texttt{`ns} > m,$$

where `Run` is the type of all traces in the GDP and $r_n \in R^n$, i.e., r_n is a state reachable in exactly n steps of GDP. Similarly, the definition of the liveness predicate states that in any trace of the protocol, for every sequence number m there exists a state in the trace where $\texttt{nd} > m$. Formally, this is expressed as follows:

$$\texttt{live}(r : \texttt{Run}) = \ \forall(m : \texttt{nat}) : \ \exists(n : \texttt{nat}) : r_n\texttt{`nd} > m.$$

The liveness property of the GDP protocol states that every fair trace is live. In PVS:

```
liveness: THEOREM
  ∀ (r: Run): fair(r) ⟹ live(r)
```

The theorem above is proved in PVS and it is part of the formal development available in the aforementioned web site. The proof of this theorem follows the proof found in [8], modulo the difference in data structures. The main steps of that proof require the following invariants

```
s`App_to_GDP`length + s`sender`ns = s`to_GDP`ptr,
```

which shows the relationship between application level data and the next item in the `ackd` window to be sent, and

```
s`GDP_to_App`length + s`from_GDP`ptr = s`receiver`nd,
```

which gives the relationship between the next frame in `rcvd` to be delivered and the data received by the application layer. Both invariants are discharged using strategies mentioned in Section 5.2.

6. Related Work

Numerous variations of the basic sliding window protocol have been subjected to hand verification techniques. Stenning [10] is likely to have been the

first to discuss the correctness of such protocols. Refinement techniques have been used by Shankar and Lam [11], Snepscheut [12] and Hoogerwoord [13] to derive the basic sliding window protocol. Process algebras have also been used to manually verify one-bit sliding window protocols [14, 15, 16]. Badban et al [17] consider a protocol with arbitrary, but finite window size, while others assume an unbounded window size.

Model checking has been applied to a number of sliding window protocols. Holzmann [18, 19] verified both safety and liveness properties for a protocol with a window size of five and Kaivola [20] did the same for a protocol with a window size of seven. Applying abstraction and model checking, Sthal was able to verify a protocol with a window size of sixteen [21].

Others have applied automated theorem provers to verify sliding window protocols. Cardell-Oliver used HOL to verify safety properties [22]. A timed model is given in [23] and a safety property is verified using PVS. Rusu [8] proved safety and liveness of a protocol with unbounded window size in PVS. Safety and liveness properties of a protocol with arbitrary finite window size employing modulo-arithmetic were verified using process algebra techniques with the assistance of the PVS prover in [17].

The sliding window protocols verified in aforementioned efforts were considerably simpler than the sliding window protocol with block acknowledgment response that is presented here. Only Badban et al. [17] also consider a protocol with arbitrary, but finite window size. Previous work considered the sliding window protocol acting in isolation rather than as a component in a protocol stack. This assumption considerably simplifies the proofs and does not accurately reflect how protocols are actually designed.

The sliding window protocol with block acknowledgement was introduced in [7] along with pencil and paper proofs of safety and liveness properties. The authors treated the sliding window protocol in isolation so the safety property is stated in terms of the sender and receiver windows, where the safety property stated in this paper is given in terms of the messages sent and received by the application layer. Given that the core of the GDP protocol is essentially the same as the protocol presented in [7], it is not surprising that the soundness proofs of both require the same basic invariants. Yet the proofs given here cannot be seen as simple transcriptions of those pencil and paper proofs as additional complexities arise due to the fact that proofs of GDP invariants must consider the protocol stack layers above and below. In particular, the need to prove the invariant at each layer of the protocol stack creates additional work and complexities, which were mitigated by the use of proof scripts to automate the proof process. In contrast to the use of theorem proving technology to formally prove correctness, the proofs in [7] are informal and conducted by hand. Their proofs mainly consist of a statement of the key invariants and comments on axiomatic techniques to prove invariants. The proof presented here was formally conducted using the model for transition systems developed by Rusu [8], where the proof is performed by induction on the length of the system trace. Another contrast between the two approaches stems from automating the correctness proofs using PVS strategies as described in Section 5, which is more akin to the

software development process than conducting hand proofs.

Concurrently executing programs are complex artifacts making it difficult to reason about their correctness. The classical theory of Owicki and Gries [24] was the first complete logic for proving correctness of parallel programs with shared variables. Each concurrently executing component is annotated with Hoare-style assertions comprising a proof outline. The key restriction placed on a proof outline is that it must be free from interference from the other concurrently executing components. Components P_1 and P_2 are said to be interference free if for all assertions p_i of P_1 and for all atomic action a_j of P_2

$$\{p_i \ \wedge \ \text{pre}\ a_j\}\ a_j\ \{p_i\}$$

and vice versa. If P_1 has n statements and P_2 has m, then the number of noninterference proof obligations will be on the order of $n \times m$. An additional drawback is that the theory is not compositional because a change in one component can affect the proofs of the other components. Mechanized verification of concurrent programs using Owicki/Gries has been performed in Isabelle [25] and PVS [26, 27].

Rely-guarantee techniques evolved from the Owicki/Gries theory by encoding interference information in the specification itself and hence used in the verification of constituent processes. Thus no additional proof obligation of interference is needed and consequently this method is compositional [28, 29]. In rely-guarantee, a component P satisfies its specification if under the assumptions that P's precondition *pre* is initially satisfied and any transition satisfies a predicate *rely*, then P ensures that any component transition satisfies the guarantee predicate *guar* and if P terminates, the the final state satisfies the postcondition *post*. The composition rule is specified as follows.

$$\frac{\begin{array}{l}\text{rely} \ \vee \ \text{guar}_1 \Longrightarrow \text{rely}_2 \\ \text{rely} \ \vee \ \text{guar}_2 \Longrightarrow \text{rely}_1 \\ \text{guar}_1 \ \vee \ \text{guar}_2 \Longrightarrow \ \text{guar} \\ P_1 \ \vdash \ (\text{pre}, \text{rely}_1, \text{guar}_1, \text{post}_1) \\ P_2 \ \vdash \ (\text{pre}, \text{rely}_2, \text{guar}_2, \text{post}_2)\end{array}}{P_1 || P_2 \ \vdash \ (\text{pre}, \text{rely}, \text{guar}, \text{post}_1 \ \wedge \ \text{post}_2)}$$

Note that the term environment encompasses the states of the concurrently executing components. Nieto [30] formalized rely-guarantee in Isabelle. Recently, rely-guarantee techniques have been married with separation logic [31] resulting in simpler proofs. Rushby [32] has developed a version of a rely-guarantee rule for use in the verification of timed reactive systems. Charpentier [33] has recently explored the composition of invariants for concurrent systems. In that work, the authors explored both invariants satisfied by every component of the composed system as well as situations similar to the one explored here, where an invariant in the composed system is satisfied by one component of the composed system. The approach proposed in this paper is not as general as those techniques, but since it is targeted toward the system under analysis, it is largely mechanizable.

7. Conclusion

A communication protocol stack intended to be used by remotely operated vehicles has been presented. Soundness and liveness properties of the protocol stack components have been formulated and proven.

All the mathematical development presented here, including the framework to compose transition systems, was formally carried out in the PVS verification system and is publicly available. In order to facilitate an iterative design process, proof strategies have been developed to automate tedious and complex tasks in the verification process, such as finding inductive invariants and proving safety properties of composed systems. The techniques presented in this paper complement the techniques in [8] by allowing them to be applied to larger systems where the designs evolve over time.

The process of proving invariant properties of the communication protocol presented in this paper considerably differs from performing the proof either informally or formally by manual means. This is because the stack structure results in an explosion in the size of the proof as invariants must be shown to hold at each level of the stack. Consequently, most analyses abstracts away lower layers of the stack in such a way as to elide interactions among the layers that may be critical to the architecture of the network. The automated approach presented in this paper relieves the user of the burden of repeatedly performing basic proof steps. It also aids the user in the process of finding the right invariants that makes the proof goes through. In addition, it transforms the process of writing a proof primarily into one of writing proof strategy scripts. One of the strengths of this approach is that it supports iterative design. The prospect of repeatedly manually reproving the soundness properties after each design change is unappealing. In contrast, using the approach presented in this paper, changes in the protocol typically require local changes to the proof scripts in order to recreate the soundness proofs.

Finally, since the protocol is specified in the declarative specification language of PVS, it is amenable to rapid prototyping. Indeed, using recently added PVS features, Java code that implements the functional and deterministic aspects of the protocol was automatically generated [34]. An actual implementation will likely be structured somewhat differently for efficiency. However, it is expected that the semantics will be preserved, allowing this prototype to serve as a semantic benchmark for the implementation.

References

[1] R. Bailey, R. Hostetler, K. Barnes, C. Belcastro, C. Belcastro, Experimental validation subscale aircraft ground facilities and integrated test capability, in: Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, San Francisco, California.

[2] T. Jordan, J. Foster, R. Bailey, C. Belcstro, AirSTAR: a UAV platform for flight dynamics and control system testing, in: Proceeding of the 25th

AIAA Aerodynamic Measurement Technology and Ground Testing Conference.

[3] A. Murch, A flight control system architecture for the NASA AirSTAR flight test infrastructure, 2008. AIAA Guidance, Navigation and Control Conference and Exhibit.

[4] S. Owre, J. Rushby, N. Shankar, PVS: A prototype verification system, in: D. Kapur (Ed.), Proc. 11th Int. Conf. on Automated Deduction, volume 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1992, pp. 748–752.

[5] A. Tannenbaum, Computer Networks, Prentice Hall, third edition, 1996.

[6] M. Gouda, Elements of Network Protocols, Wiley-Interscience, 1998.

[7] M. Brown, M. Gouda, R. Miller, Block acknowledgement: Redesigning the window protocol, IEEE Transactions on Communications 39 (1991) 524–532.

[8] V. Rusu, Verifying a Sliding-Window Using PVS, in: Formal Techniques for Networked and Distributed Systems (FORTE01), Kluwer Academic, 2001, pp. 251–266.

[9] M. Archer, B. D. Vito, C. Muñoz, Developing user strategies in PVS: A tutorial, in: Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03, NASA/CP-2003-212448, NASA LaRC,Hampton VA 23681-2199, USA, pp. 16–42.

[10] N. Stenning, A data transfer protocol, Computer Networks 1 (1976) 99–110.

[11] A. Shankar, S. Lam, Construction of network protocols by stepwise refinement, in: J. de Bakker, W.-P. de Rover (Eds.), Proceedings of Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, Lecture Notes in Computer Science 430, Springer-Verlag, 1989, pp. 669–695.

[12] J. V. de Snepscheut, The sliding-window protocol revisited, Formal Aspects of Computing 7 (1995) 3–17.

[13] R. Hoogerwoord, A formal derviation of a sliding window protocol, 2006. Technical University of Eindhoven.

[14] F. Vaandrager, Verification of Two Communication Protocol by Means of Process Algebra, Technical Report, CWI, 1986.

[15] J. Brunekreff, Sliding window protocols, in: Algebraic Specification of Protocols, number 36 in Cambridge Tracts in Theoretical Computer Science, 1993, pp. 71–112.

[16] K. Paliwoda, J. Sanders, An incremental specification of the sliding-window protocol, Distributed Computing (1991).

[17] B. Badban, W. Fokkink, J. Groote, J. Pang, J. van de Pol, Verification of a sliding window protocol in μCRL and PVS, Formal Aspects of Computing 17 (2005) 342–388.

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

[19] G. Holzmann, The model checker Spin, IEEE Transactionsactions of Software Engineerng 23 (1997) 279–295.

[20] R. Kaivola, Using compositional preorders in the verification of a sliding window protocol, in: Proceedings of the 9th Conference on Computer Aided Verification, Lecture Notes in Computer Science 1254, Springer-Verlag, 1997, pp. 48–59.

[21] K. Stahl, K. Baukus, K. Lakhnech, Y. Steffen, Divide, abstract, and model check, in: Proceedings of the 6th International SPIN Workshop, Lecture Notes in Computer Science 1680, pp. 57–76.

[22] R. M. Cardell-Oliver, The Formal Verification of Hard Real-Time Systems, Ph.D. thesis, University of Cambridge, 1992.

[23] D. Chkliaev, J. Hooman, E. de Vink, Verification and improvement of the sliding windonw protocol, in: Proceedings of the 9th Conference on Tools and Algorithms for the Construction of Analysis of Systems (TACAS'03), Lecture Notes in Computer Science 2619, Springer-Verlag, 2003, pp. 113–127.

[24] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Informatica 6 (1976) 319–340.

[25] T. Nipkow, L. P. Nieto, Owicki/Gries in Isabelle/HOL, in: Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science 1577, Springer-Verlag, 1999, pp. 188–203.

[26] A. Mooij, W. Wesselink, Incremental verificaiton of Owicki/Gries proof outlines using pvs., in: Proceedings of the 7th International Conference on Formal Engineering Methods, Lecture Notes in Computer Science 3785, Springer-Verlag, 2005, pp. 390–404.

[27] J. Koudijs, Automated Verification of Owicki/Gries proof outlines: comparing PVS and Isabelle, Master's thesis, T.U. Eindhoven, 2006.

[28] C. Jones, Tentative steps toward a method for interfering programs, ACM Transactions of Programming Languages and Systems (TOPLAS) 5 (1983) 596–619.

[29] Q. Xu, W. de Roever, J. He, The rely-guarantee method for verifying shared variable concurrent programs, Formal Aspects of Computing 9 (1997) 149–174.

[30] L. Nieto, The rely-guarantee method in Isabelle/HOL, in: Programming Languages and Systems, Lecture Notes in Computer Science 2618, Springer-Verlag, 2003, pp. 348–362.

[31] V. Vafeiadis, M. Parkinson, A marriage of rely/guarantee and separation logic, in: Proceedings of 18th International Conference on Concurrency Theory (CONCUR), Lecture Notes in Computer Science 4136, Springer-Verlag, 2007, pp. 256–271.

[32] J. Rushby, Formal Verification of McMilian's Compositional Assume-Guarantee Rule, Technical Report, SRI, 2001.

[33] M. Charpentier, Composing invariants, Science of Computer Programming 60 (2006) 221–243.

[34] L. Lensink, C. Muñoz, A. Goodloe, From Verified Models to Verifiable Code, Technical Memorandum NASA/TM-2009-215943, NASA, Langley Research Center, Hampton VA 23681-2199, USA, 2009.