Compositional Verification of a Communication Protocol for a Remotely Operated Vehicle*

Alwyn E. Goodloe[1] and César A. Muñoz[2]

[1] Alwyn.Goodloe@nianet.org
National Institute of Aerospace
100 Exploration Way, Hampton, VA 23666, USA
[2] Cesar.A.Munoz@nasa.gov
National Aeronautics and Space Administration
Langley Research Center, Hampton, VA 23681, USA

Abstract. This paper presents the specification and verification in the Prototype Verification System (PVS) of a protocol intended to facilitate communication in an experimental remotely operated vehicle used by NASA researchers. The protocol is defined as a stack-layered composition of simpler protocols. It 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 satisfies a distinct requirement. It is formally proven that the protocol components satisfy certain delivery guarantees. Compositional techniques are used to prove these guarantees also hold in the composed system. 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.

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. When flying, commands from the ground-based pilot are broadcast to the aircraft and telemetry data from the aircraft are broadcast to the ground station. Hence, communication between the air and ground components is critical for the safe operation of the vehicle. This paper presents the formal verification, in the Prototype Verification System (PVS) [15], of a communications protocol designed for use in AirSTAR [2], a dynamically scaled experimental aircraft designed and built by NASA's Langley Research Center (LaRC) for use as a

* This work was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NNX08AE37A awarded to the National Institute of Aerospace. This work was done while the second author was resident at the National Institute of Aerospace. Authors are in alphabetical order.

testbed for research on software health management and flight control. This protocol is formed from the composition of several simpler protocols structured as a protocol stack. This paper focuses on the formal verification of delivery properties of the communication protocol.

The verification approach employs a compositional technique where the delivery property of the composed protocol is lifted from the delivery properties of its components. Most the 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 mathematical development presented in this paper has been formally verified in PVS. This development is electronically available from `http://research.nianet.org/fm-at-nia/IVHM`.

2 Protocol Requirements

A ROA platform consists of an airborne vehicle and a 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. Developments in both the design and application of ROAs have led to a number of innovations in wireless communication in this domain. For instance, a flock of ROAs may employ adhoc networking to imbue the collection with a routing capability allowing for sophisticated communication. AirSTAR, on the other hand, 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 and this is unlikely to change over the life of the aircraft. The aircraft currently uses a very simple communication scheme in which all broadcast messages are treated alike. Flight 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 protocol should guarantee that all telemetry data broadcast is eventually delivered. This requirement is called the *guaranteed delivery requirement.*

Since the requirements of weak and guaranteed delivery are in some sense orthogonal to each other, the protocol has been structured as two different protocols: the *weak delivery protocol* (WDP) and the *guaranteed delivery protocol* (GDP). The differences between WDP and GDP are similar to the differences between the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP) of the Internet protocol suite. However, WDP and GDP are considerably smaller, simpler, and more verifiable than UDP and TCP, which are considered to be too complex to be used in AirSTAR. In addition to these two protocols, other protocols are needed to support the communication between the aircraft and the ground station. In particular, a *link layer* is considered that performs error detection and multiplexes WDP and GDP messages into the physical communication medium.

This paper focuses on *functional correctness* of delivery properties. The correctness criteria for guaranteed delivery is that messages are received in the order they are sent. A liveness property says that the messages will eventually arrive. In the case of weak delivery, the correctness criteria states that every message received was in the sequence of messages that were sent. The protocol underwent several iterations and, for that reason, a methodology that accommodated such evolution has been employed.

3 Protocol Stack

A collection of protocols 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 protocol when sending a message. The next layer down corresponds to the transport layer and it is here that the core of the GDP and WDP 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. Note that the GDP and WDP 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 two communication channels over which messages are sent and received.

The proposed protocol stack is illustrated in Figure 1. The protocol stack can be viewed 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 GDP and WDP lie at the same layer, but they behave differently as they satisfy different requirements. These may be viewed as disjoint components occupying the same layer in the stack, but possess no shared state. On the other hand, the two layers interact with the same link layer. Consequently, the link layer is shared between the WDP and GDP components. Each protocol in the stack

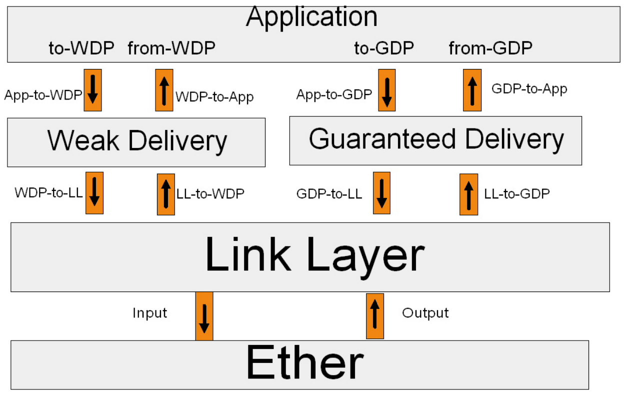

Fig. 1. Protocol stack

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 depicted in Figure 1, where each queue is represented as a small rectangle with an arrow pointing in 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`. Note that the ether is not a protocol layer, but a model of the transport medium.

4 Protocol Specification

A specification of the protocol stack described in the previous section has been constructed using PVS, which provides a rich specification language and a powerful theorem prover. The use of a theorem prover, as opposed to a model checker, 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.

4.1 Ether

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

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

where `LinkFrame` is defined in Section 4.2. 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 type `EtherAction` as follows:

```
DropIn(linkframe:LinkFrame) : DropIn? +
DupIn(linkframe:LinkFrame) : DupIn? +
NoiseIn(linkframe:LinkFrame) : NoiseIn?
```

where the constructor is defined left of the colon and a recognizer for the type defined to the right of the colon. The ether state machine, in effect, perturbs the ether 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 a frame on the inbound was :

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

which returns a new ether state with the value `linkframe` removed from the ether's input channel. Note the back-quote symbol is the PVS field access operator. The state machines are functional models, but above this layer the model is relational. In this case, the relation

```
ether?(s,n:Ether) : bool = ∃(a: EtherAction) : n = next(s,a),
```

non-deterministically selects a valid action for the state machine to execute.

4.2 Link Layer

The link layer is intended to serve as an interface between the protocol stack and the communication medium, since the physical layer is abstracted away, as well as to provide common services needed by the protocols that lay at the next higher layer. The link layer also performs a check-sum error detection. Furthermore, the link layer multiplexes messages sent from the WDP and GDP layers wrapping them in a common header, and demultiplexes them on the receiving side 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:

```
LinkFrame = cs: CheckSum × 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 way. The type `LinkInterface` is a 4-tuple formed from the four queues `GDP-to-LL`, `WDP-to-LL`, `LL-to-GDP`, and `LL-to-WDP`. The type `Link` is a tuple formed from the `Ether` and the `LinkInterface`. Hence, the type `Link` represents the state of all information entering and leaving the link layer.

The link layer functionality is represented by a transition function that, given the current `Link` and the action to perform, yields the next state, where the possible actions are: *send* a WDP message, *send* GDP message, and *receive* a message. If sending a WDP or GDP 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 state machine receive actions are expressed in PVS as follows:

```
Receive(linkframe) : IF  member(linkframe,s‘ether‘output)  THEN
  IF  ¬checksum?(linkframe) THEN
   s WITH [  ‘ether‘output := remove(linkframe,s‘ether‘output]
  ELSE CASES linkframe‘frame OF
    GDP(gdpframe) :  s WITH [
      ‘link‘ll_to_gdp := enqueue(gdpframe,s‘link‘ll_to_gdp),
      ‘ether‘output := remove(linkframe,s‘ether‘output) ],

    WDP(wdpframe) : s WITH [
      ‘link‘ll_to_wdp := enqueue(wdpframe,s‘link‘ll_to_wdp),
      ‘ether‘output := remove(linkframe,s‘ether‘output) ]
     ENDCASES
   ENDIF
```

4.3 Weak Delivery Protocol

The Weak Delivery Protocol is extremely simple and so is its model. The type `WDP` is a 5-tuple formed from the two sequences `to-WDP` and `from-WDP`, the two queues `App-to-WDP` and `WDP-to-App`, and the `LinkInterface`. Sending a message is modeled as removing a message from the `App-to-WDP` queue and adding it to the `WDP-to-LL` queue. Receiving a message is modeled as removing a message from `LL-to-WDP` queue and adding it to the `WDP-to-App` queue.

4.4 Guaranteed Delivery Protocol

The Guaranteed Delivery Protocol shall satisfy the guaranteed delivery requirement. Following the standard solution to this problem, GDP is designed as a sliding-window protocol [19]. Discussions with the AirSTAR engineers revealed a communication pattern that led to a sliding-window protocol with block acknowledgment developed by Gouda [7, 8]. Although an informal proof may be found in the literature, the one presented in this paper appears to be the first attempt at a formal mechanical proof of the protocol.

Each GDP message has a sequence number that acts as an identifier. The receiver replies with a message acknowledging the receipt of a contiguous block of sequence of numbers. The sender and receiver maintain bounded windows, also 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 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 sequence numbers, although in the actual specification some amount of machinery is needed to map an unbounded range of sequence numbers to a bounded buffer.

The sender maintains the following pointers. The variable `ns` is a pointer to the sequence number of the next data item to be sent and the variable `na` is a pointer 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 ($\texttt{na}$,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 maintains the following pointers. 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 $\texttt{lr} \leq k < \texttt{nd}+\texttt{rw}$, in any order, and ignores messages out of this range. When the receiver has received the contiguous block of sequence numbers ($\texttt{nd}$,n) the pointer `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 n, where $\texttt{la} \leq n \leq \texttt{nd}-1$, have been received and delivered, but not yet acknowledged. Periodically, GDP sends the block acknowledgment for sequence numbers $(\texttt{la}, \texttt{n} - 1)$ and `la` is reset to `nd`.

4.5 Application Layer

The sender and receiver processes at the application layer are each composed of two state machines. At the sender, one machine maintains a pointer to the next message in `to-GDP` to be sent, copies that message to `App-to-GDP`, and increments the pointer. The other machine behaves similarly by copying messages from `to-WDP` to `App-to-WDP`. The receiver processes move messages from `GDP-to-App` to `from-GDP` and from `WDP-to-App` to `from-WDP`.

4.6 Composing Models

For each one of the WDP and GDP protocols, it will be assumed that there are two processes: a sender process and a receiver process. The WDP sender and receiver processes are called `WDPSender?` and `WDPReceiver?`, respectively. Similarly, the GDP sender and receiver processes are called `GDPSender?` and `GDPReceiver?`, respectively. These processes behave in a non-deterministic way. Hence, each one of them is defined as a relation between the current state and one of the possible next states. For instance, `GDPSender?`, which relates the current state of the GDP sender process and a possible next state, is defined as either a `GDPSenderNext` transition, a `LinkNext` transition, or a `EtherNext` transition, where the fields that are not modified by the transitions remain unchanged. The transitions are a function from the current state and an action to perform to the next state. In order to model the non-deterministic selection of actions, existential quantifiers are used to generate actions for each transition. The relation `GDPSender?` is formally expressed as follows:

$$
\begin{aligned}
&\texttt{GDPSender?}(s, n : \texttt{GDPSender}) = \\
&(\ \exists\ a : \texttt{GDPSenderAction}.\ n = \texttt{GPDSenderNext}(s, a)\ ) \\
\bigvee\ & \\
&(\ \exists\ a : \texttt{LinkAction}.\ n_l = \texttt{LinkNext}(s_l, a) \\
&\quad \wedge\ n`\texttt{App\_to\_GDP} = s`\texttt{App\_to\_GDP} \wedge\ s`\texttt{winsender}\ =\ n`\texttt{winsender}\ ) \\
\bigvee\ & \\
&(\ \exists\ a : \texttt{EtherAction}.\ n_e = \texttt{EtherNext}(s_e, a) \\
&\quad \wedge\ n`\texttt{link} = s`\texttt{link}\ \wedge\ n`\texttt{App\_to\_GDP} = s`\texttt{App\_to\_GDP} \\
&\quad \wedge\ s`\texttt{winsender}\ =\ n`\texttt{winsender}\ ),
\end{aligned}
$$

where s and n stand for the current and next `GDPSender` state, respectively, and the back-quote symbol is the field access operator. The projections of states s, n into `Link` and `Ether` states are denoted by sub-indices l and e, respectively. The GDP receiver and WDP processes have a similar model.

5 Protocol Verification

This paper focuses on the functional correctness of WDP and GDP. The functional correctness of a system is usually expressed by *invariant safety and liveness properties*, i.e., predicates that hold in every reachable state of the system. 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.

There are many relationships that are local to either the sender or the receiver. For instance, the property that states 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., $\mathtt{na} \leq \mathtt{ns}$, 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., $\mathtt{la} \leq \mathtt{nd}$, only concerns the receiver. As these properties can be described solely in terms of the states of the GDP sender or the GDP receiver processes, they can be easily encoded using the PVS's subtype and dependent type system. These generate type correctness conditions, which most of the time can be automatically proved by the PVS type checker. The remainder of this section will focus on properties that relate the sender and receiver processes and consequently require more complex reasoning.

Consider the case of a system of two nodes *exclusively* running the WDP protocol. The state of this system, represented by the type `WDPSystem`, is a n-tuple composed of the union of the fields in `WDPSender` and `WDPReceiver` such that the input and output channels of the ether interface in the sender are connected, respectively, to the output and input channels of the ether interface in the receiver. The invariant predicate that expresses the correctness property of WDP is defined as follows:

$$\mathtt{wdp\_sound}(s : \mathtt{WDPSystem}) \equiv \mathtt{from\text{-}WDP}_s \subseteq \mathtt{to\text{-}WDP}_s,$$

where s refers to a reachable state. Henceforth, variables are sub-indicated with the state to which they belong, e.g., $\mathtt{from\text{-}WDP}_s$ refers to state of the sequence `from-WDP` in state s and $\mathtt{to\text{-}WDP}_s$ refers to the state of the `to-WDP` sequence in s. This invariant states that all WDP messages that the receiver node delivers to the application layer were indeed sent by the sender's application layer.

In the case of a system of two nodes *exclusively* running the GDP protocol, the state, represented by the type `GDPSystem`, is a n-tuple composed of the union of the fields in `GDPSender` and `GDPReceiver`. The ether in the sender and receiver sides are connected in a similar way as in the WDP. The invariant predicate that expresses the correctness property of GDP is defined as follows:

$$\mathtt{gdp\_sound}(s : \mathtt{GDPSystem}) \equiv \mathtt{from\text{-}GDP}_s \preceq \mathtt{to\text{-}GDP}_s,$$

where $\preceq$ is the prefix relation between sequences. This invariant 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.

For GDP, a traditional fairness property [16] is considered, which states that all messages in the `to-GDP`queue are eventually sent. That is, for every message in `to-GDP`, it is eventually the case that a state is recorded where each message has been sent. Since it is an invariant that `ns` always points to the next item to be sent, the fairness property can be stated as saying that given any run of the protocol, for every sequence number m the run records a state where $\mathtt{ns} > m$. This is stated formally as follows:

$$\mathrm{fair}[(\mathrm{run})] = \lambda(r : (run)) :\ \forall(m : \mathrm{Nat}) :\ \exists(n : \mathrm{Nat}) : r_n`\mathtt{ns} > m.$$

The predicate live states that all data is eventually delivered. Formally, this is expressed as a predicate on the runs of the protocol as follows:

$$\text{live}[(\text{run})] = \lambda(r : (run)) :\ \forall(m : \text{Nat}) :\ \exists(n : \text{Nat}) : r_n\text{`}\mathtt{nd} > m.$$

The liveness property is then given as:

$$\text{liveness}[(\text{run})] = \lambda(r : (run)) :\ \text{fair}(r) \Rightarrow \text{live}(r).$$

The primary verification objective of this work is to formally prove that the predicates, wdp_sound, gdp_sound, and liveness are indeed invariants when both WDP *and* GDP run simultaneously in each node. This system is the *asynchronous composition* of WDP and GDP and will be denoted by WDP || GDP. To verify wdp_sound, gdp_sound, and liveness in the composed system, a compositional approach is proposed where each invariant is independently proved for its respective system, i.e., wdp_sound is an invariant of WDP and gdp_sound is an invariant of GDP, and then a general framework is provided that enables the lifting of an invariant property on one system, e.g., gdp_sound on GDP, to an invariant on a composition of systems, e.g., gdp_sound on WDP || GDP.

5.1 Proving Invariants on WDP and GDP, Independently

Proving invariants on transition systems, such as WDP or GDP, are routine in the theorem proving community. It usually entails the transformation of the initial invariant to a weaker form that can be proved by induction. A simple set of theories developed by Rusu [16] is used for proving invariants on discrete transition systems by natural induction on the length of the system traces. The nontrivial task of finding auxiliary invariants that enable the inductive proof of the original invariant is subject to the ingenuity of the human prover.

For WDP and GDP the problem is made harder by the fact the full protocol stack and all possible interleavings between the sender and receiver processes have to be considered. As seen in Section 4.6, the sender and receiver components of each protocol are formed from the disjunction of a number of relations representing the layers of the stack. This means that that an invariant must be shown to hold under each transition in each layer. Consequently, each proof requires the discharge of a large number of cases. For each one of these cases, it has to be proven that if an invariant is satisfied at step n, it is also satisfied at step $n + 1$. This is a considerable amount of work even though many of the cases can be easily discharged by using general properties of bags, queues, and buffers.

To automate the verification task, a set of proof strategies that are applied to discrete transition systems defined using Rusu's PVS theories has been defined. The use of such strategies form the basis of a methodology that will allow Rusu's techniques to scale to industrial-size problems. The strategies basically unfold the transition relations and discharge the easy cases of inductive proofs. For instance, to prove an invariant on GDP, the strategy unroll-gdp is invoked. This strategy, in turn, invokes strategies unroll-gdp-sender and unroll-gdp-receiver as well as strategies to unroll the application layer sender and receiver processes. The `unroll-gdp-sender` strategy, for example, expands the relational definitions, instantiating and skolemizing quantifiers as needed, until it finally expands the definitions of the state machines. In the case of the state machine `GDPSenderNext`, the strategy "lifts" the conditionals so as to expose the guarded cases, which, in turn, are discharged using PVS's assert decision procedure. Additional support strategies are employed that apply properties of structures such as bags, FIFO queues, and bounded buffers to simplify expressions to the point where basic decision procedures can be applied to complete the proof. Even in the cases where the strategies do not succeed, they generate enough information to assist a developer in finding weaker invariants.

To perform the proof of `wdp_sound`, the first command is the strategy `discharge-inv`, which automatically proves all but two inductive cases. The first case is discharged by simply unfolding a definition. The second unproven case suggests the need for an invariant saying that all frames in `WDP-to-App` are in `to-WDP`:

$$\texttt{WDP-to-App}_s \subseteq \texttt{to-WDP}_s.$$

To prove this an additional auxiliary invariant is needed that states that WDP frames in the link layer and in the ether belong to `to-WDP`. Once this invariant is added as a lemma to the theory, the proof is finished by using the strategy `use-inv`. To prove the auxiliary invariant, the same approach is used, which suggests the new invariant:

$$\texttt{App-to-WDP}_s \subseteq \texttt{to-WDP}_s.$$

This new invariant is automatically discharged by `discharge-inv`.

The proof of `gdp_sound` is considerably more complicated, but the general method is the same. The strategy `discharge-inv` is used to eliminate the easy cases and new invariants are added to discharge the unproven cases via `use-inv`. This approach is iterated on the new invariants. In total, six auxiliary invariants have been added to the GDP theory, including the following relations between the sender's and receiver's windows:

- The counter of received messages is less than or equal to the counter of sent messages: $\texttt{lr}_s \leq \texttt{ns}_s$
- The counter of delivered messages is less than or equal to the counter of sent messages: $\texttt{nd}_s \leq \texttt{ns}_s$
- The largest sequence number for which an acknowledgment has been received is less than or equal to the counter of the sent acknowledgments

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

  where the function `last_true` returns the difference between $\texttt{na}_s$ and the largest sequence number for which an acknowledgment has been received.

The stack structure considerably affects the size of the proofs as invariants have to be checked at different layers. Although most of the manual tasks are routine, scale becomes a prohibitive factor that will get worse in larger models. If heavy-weight formal methods are to be used in industrial practice, they must accommodate an iterative design process. Manually proving the GDP process after even a simple design change can take much of a day and the prospect of repeatedly doing so for each design iteration is not practical. The strategies are written in a lisp-like PVS scripting language and are composed of 937 lines of code. To maintain a high degree of automation, changes to the model are reflected in the strategy code, which is an integral part of the iterative design methodology. All the strategies and proofs can be found at the aforementioned web site.

5.2 Proving Invariants on the Asynchronous Composition of WDP and GDP

In the previous section, it has been proven that `wdp_sound` is an invariant of WDP and that `gdp_sound` is an invariant of GDP. However, the verification objective 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 proven that `wdp_sound` and `gdp_sound` are invariants of WDP || GDP using the method explained in the previous section. However, this approach does not profit from the invariants that have been already proven for WDP and GDP independently, and therefore the have to be proven again for all possible interleavings of WDP and 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 on one system can be *lifted* to the composed system. To this end, it is considered that the state of a transition system consists of a private state and a shared state. The state of the composed system has a copy of the private states of each transition system but only one shared state common to both of them. When the composed system performs a transition of one system, the private state of the other system remains unchanged.

A transition system is defined as follows. Let V be a finite set of typed variables and Θ an initial condition defined on the variables. Define state S_V as a type-consistent valuation of the variables. A transition is a relation $\rightarrow$ in $S_V \times S_V$. A transition system is defined as the tuple $T = (S_V, \Theta, \rightarrow)$. Given two transition systems $T_1 = (S_{V_1}, \Theta_1, \rightarrow_{T_1})$ and $T_2 = (S_{V_2}, \Theta_2, \rightarrow_{T_2})$, , define $T_1 \parallel T_2 = (S_{V_1 \cup V_2}, \Theta_{T_1 \parallel T_2}, \rightarrow_{T_1 \parallel T_2})$ as follows: the state space $S_{V_1 \cup V_2}$ is a valuation of the variables in V_1 and V_2. Let $s \in S_{V_1 \cup V_2},$, define the restriction operators $s{\downarrow_{T_i}}$ and $s{\downarrow_{[T_i]}}$, for $i = \{1, 2\}$ such that the first operator projects the composed state to the state of T_i, which only includes the private and shared state of T_i and the second operator only projects the private part of T_i.

The composed initial state is defined as

$$\Theta_{T_1 \parallel T_2} \; = \; \{s : S_{V_1 \cup V_2} \mid s{\downarrow_{T_1}} \in \Theta_{T_1} \wedge s{\downarrow_{T_2}} \in \Theta_{T_2}\},$$

and the composed transition relation is defined as

$$
\begin{aligned}
s \rightarrow_{T_1 \| T_2} s' \;=\; \{(s,s') : S_{V_1 \cup V_2} \times S_{V_1 \cup V_2} \mid & \; s{\downarrow}_{T_1} \rightarrow_{T_1} s'{\downarrow}_{T_1} \wedge s{\downarrow}_{[T_2]} = s'{\downarrow}_{[T_2]} \\
\vee & \; s{\downarrow}_{T_2} \rightarrow_{T_2} s'{\downarrow}_{T_2} \wedge s{\downarrow}_{[T_1]} = s'{\downarrow}_{[T_1]}\}.
\end{aligned}
$$

An *abstraction* α of a transition system T is a simulation relation that maps states into states such that

1. if s_0 is an initial state in T, then $\alpha(s_0)$ is also an initial state of T, and
2. if $s_n \rightarrow_T s_{n+1}$ then $\alpha(s_n) \rightarrow_T \alpha(s_{n+1})$.

The following theorem is sufficient to prove that an invariant on the left-hand side of the parallel operator is also an invariant of the composed system.

Theorem 1 (Invariant Left-Lifting). *Let P be an invariant of a transition system T_1. The predicate $P_{T_1 \| T_2}$, where $P_{T_1 \| T_2}(s : S_{V_1 \cup V_2}) \equiv P(s{\downarrow}_{T_1})$, is an invariant of the transition system $T_1 \parallel T_2$ if there is an* abstraction *α of T_1 such that the following conditions are met:*

1. *α is fixed under P, i.e., $P(\alpha(s{\downarrow}_{T_1}))$ implies $P(s{\downarrow}_{T_1})$, and*
2. *under the abstraction α, T_2 does not interfere with T_1, i.e., given $s_n, s_{n+1} : S_{V_1 \cup V_2}$, if $s_n{\downarrow}_{T_2} \rightarrow_{T_2} s_{n+1}{\downarrow}_{T_2}$ then $\alpha(s_n{\downarrow}_{T_1}) \rightarrow_{T_1} \alpha(s_{n+1}{\downarrow}_{T_1})$.*

Proof (Sketch of PVS Proof). Consider an arbitrary trace $s_0, \ldots, s_n$ in $T_1 \parallel T_2$. It is shown that P holds in s_n. First, it is shown that $\alpha(s_0{\downarrow}_{T_1}), \ldots, \alpha(s_n{\downarrow}_{T_1})$ is a trace in T_1. There are two cases:

1. The transition (s_i, s_{i+1}) is transition in T_1. In this case, $\alpha(s_i{\downarrow}_{T_1}) \rightarrow_{T_1} \alpha(s_{i+1}{\downarrow}_{T_1})$ since α is an abstraction of T_1.
2. The transition (s_i, s_{i+1}) is a transition in T_2. In this case, $\alpha(s_i{\downarrow}_{T_1}) \rightarrow_{T_1} \alpha(s_{i+1}{\downarrow}_{T_1})$ since T_2 does not interfere with T_1.

Therefore, $\alpha(s_0{\downarrow}_{T_1}), \ldots, \alpha(s_n{\downarrow}_{T_1})$ is a trace in T_1. Since P is an invariant on T_1, P holds in $\alpha(s_i{\downarrow}_{T_1})$, for $i \leq n$. Since α is fixed under P, P holds in $s_i{\downarrow}_{T_1}$ as well. The result then follows from the fact that $P_{T_1 \| T_2}(s_i)$ is defined as $P(s_i{\downarrow}_{T_1})$. □

A symmetric theorem for the right transition system can be proved in a similar way. Both theorems have been mechanically proven in PVS and both the formalization and proof can be found online.

For the case of the distributed system `WDP` ∥ `GDP`, the queues `App-to-WDP` and `WDP-to-App` are private to `WDP`. Although the sequences `to-WDP` and `from-WDP` reside in the application layer, 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.

The fact that wdp_sound is an invariant of WDP $\|$ GDP is a consequence of the invariant lifting theorems.

Theorem 2 (WDP **Soundness).** *WDP_sound is an invariant on WDP $\|$ GDP.*

For the proof of WDP, the abstractions that are needed are filters that remove, respectively, GDP packets from the link layer and the ether interface.

Proof (Sketch of PVS Proof). 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 proven that α_w is an abstraction of WDP, that WDP_sound is fixed to α_w, and that, under α_w, GDP does not interfere with WDP. Therefore, by the fact that the invariant WDP_sound holds on WDP and theorem 1, WDP_sound is an invariant on WDP $\|$ GDP. □

The hypotheses to the theorem are automatically discharged by strategies that have been developed to prove that a given function is an abstraction, that an abstraction is fixed to an invariant, and that the noninterference condition holds. The statement and proof that gdp_sound is an invariant of WDP $\|$ GDP is similar.

6 Related Work and Conclusion

Numerous variations of the basic sliding window protocol have been subjected to hand verification techniques. Stenning [18] is likely to have been the first to discuss the correctness of such protocols. Snepscheut [6] and Hoogerwoord [10] are representative of this work. Process algebras have also been used to manually verify one-bit sliding window protocols [3, 20]. Badban et al [1] considers a protocol with arbitrary, but finite window size while others assume an unbounded window size. Model checking has been applied to verifying a number of sliding window protocols e.g. [9, 12, 17], but to prevent state explosion the window size has to be kept to a relatively small size.

Others have applied automated theorem provers to verify sliding window protocols. Cardell-Oliver used HOL to verify safety properties [4]. A timed model was given in [5] and a safety property is verified using PVS. Rusu [16] proved safety and liveness of a protocol with unbounded window size in PVS.

Concurrently executing programs are complex artifacts making it difficult to reason about their correctness. For parallel programs with shared variables, the classical theory of Owicki and Gries [14] was the first breakthrough for reasoning about the correctness of parallel programs having shared variables, but the theory is not compositional. Assume-Guarantee methods modify the theory to be compositional [11,21]. Nieto [13] formalized rely-guarantee in Isabelle. The approach proposed here is not as general as these techniques, but was targeted toward the system under analysis, yet is largely mechanizable as has been shown here.

A small communication protocol stack intended to be used by remotely operated vehicles has been presented. The 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,novel 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. As an added feature, the strategies are robust to changes in the protocol specification. Therefore, protocol modifications usually require only minor changes in the soundness proofs rather than having to redo all the proofs by hand. The techniques presented in this paper complements the techniques in [16] by allowing them to be applied to larger systems where the designs evolve over time.

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. 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.

Acknowledgements

The author would like to thank the AirSTAR team and in particular David Cox for their technical support, and Eric Cooper, Paul Miner and the anonymous referees for their comments that help to improve the presentation of this work.

References

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

2. R. Bailey, R. Hostetler, K. Barnes, C. Belcastro, and 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, 2005.
3. J. Brunekreff. Sliding window protocols. In *Algebraic Specification of Protocols*, number 36 in Cambridge Tracts in Theoretical Computer Science, pages 71–112. 1993.
4. Rachel Mary Cardell-Oliver. *The Formal Verification of Hard Real-Time Systems*. PhD thesis, University of Cambridge, 1992.
5. D. Chkliaev, J. Hooman, and 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, pages 113–127. Springer-Verlag, 2003.
6. J.L.A. Van de Snepscheut. The sliding-window protocol revisited. *Formal Aspects of Computing*, 7:3–17, 1995.
7. M. Gouda. *Elements of Network Protocols*. Wiley-Interscience, 1998.
8. M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.
9. G. Holzmann. The model checker Spin. *IEEE Transactionsactions of Software Engineerng*, 23(4):279–295, 1997.
10. R. Hoogerwoord. A formal derviation of a sliding window protocol. Technical University of Eindhoven, 2006.
11. C. Jones. Tentative steps toward a method for interfering programs. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.
12. 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, pages 48–59. Springer-Verlag, 1997.
13. L. Nieto. The rely-guarantee method in Isabelle/HOL. In *Programming Languages and Systems*, Lecture Notes in Computer Science 2618, pages 348–362. Springer-Verlag, 2003.
14. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
15. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.
16. V. Rusu. Verifying a Sliding-Window Using PVS. In *Formal Techniques for Networked and Distributed Systems (FORTE01)*, pages 251–266. Kluwer Academic, 2001.
17. K. Stahl, K. Baukus, K Lakhnech, and Y Steffen. Divide, abstract, and model check. In *Proceedings of the 6th International SPIN Workshop*, Lecture Notes in Computer Science 1680, pages 57–76, 1999.
18. N. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.
19. A. Tannenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.
20. F. Vaandrager. Verification of two communication protocol by means of process algebra. Technical report, CWI, 1986.
21. Q. Xu, W. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.