Protocol verification made simple: a tutorial

Mohamed G. Gouda *

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188, USA

Dedicated to the memory of Professor Louis E. Rosier (1951–1991): It ain't over.

Abstract


We describe how to define communication protocols, how to express protocol properties, and how to verify that the execution of a given protocol definition satisfies a given protocol property. The presentation is rarely opinionated, sometimes conversational, mostly rigorous, but always simple.

Keywords: network protocols; verification; formal method; invariant; progress; fault-tolerance.

1. A case of false advertisement

The title of this paper is somewhat misleading: it implies that protocol verification is “complex” and needs to be “simplified”. This is simply not true: protocol verification is already simple; it only needs to be presented as it is. Unfortunately, most literature in the area may have led you to believe otherwise. The wide discrepancy between the actual simplicity of protocol verification and its alleged complexity can be traced back to three factors: education vacuum, unchecked rumors, and economic motives. Our objectives of discussing these factors here is to try to explain how protocol verification has acquired its undeserved reputation of being difficult, complex, and impractical.

Education vacuum: As it happens, very few courses have ever been offered in the area of protocol verification. Very few books have ever been written on the subject. Researchers who aspire to work on the area have to rely on themselves in selecting and reading a paper here, or studying a paper there, without a grand plan to guide their study. In the absence of formal courses and good books on the subject, casual observers of the area conclude that the area is difficult, and researchers who may have been lured into the area decide to work on something else.

Unchecked rumors: It has been my observation over the years that attacks on protocol verification have come, almost exclusively, from researchers who never studied the area, and never practiced its subject. These attacks do not reflect deep, insightful knowledge of protocol verification; nor they reflect a first hand experience of its practices. If anything, these attacks reflect an ambiguous discomfort of an unknown, and a well-seated reluctance to know it. Like unchecked rumors, these attacks have spread far and fast: everyone have heard of them, but no one knows when or how they have originated. (Indeed, I cannot recall a major effort in protocol verification that has failed, and its failure could have prompted these attacks.)

Economic motives: As long as society accepts the myth that protocol verification is impractical, industrial institutions that develop and produce

* Gouda is supported in part by the ONR Grant N00014-89-J-1913, and by a grant from the Texas Advanced Technology Program 1992–1994.
protocols will not be required to verify their products. And as long as these institutions are not required to verify their protocols, they will not verify them in order to maximize their short-term profits. The net outcome is that society will continue to accept protocols that are wrong, expensive, and inefficient. Protocol bugs will continue to be an integral part of our lives. If an electronic mail message that you send is lost due to a protocol bug, please accept our apologies. After all what else can be done, if protocol verification is out of the question.

I do not know how to counter unchecked rumors or the economic motives against protocol verification. This tutorial, however, is intended as a small first step in filling the education vacuum surrounding protocol verification. If you always wanted to study protocol verification but was afraid to ask, here is your chance. Sit back, relax, and read on.

2. What is protocol verification anyway

It is instructive to view a protocol verification method as a procedure with two inputs and one output. The procedure takes a protocol definition on one input, and a statement describing some property of protocol execution on the other input, then computes a yes/not-sure answer. A “yes” answer means that the following proposition has been established: The input property is indeed a property of the input protocol. A “not-sure” answer means that some effort to establish this proposition has failed. There are two possible causes for such a failure: either the proposition is false and so cannot be established, or it is true but more efforts are needed in order to establish it. Note that a procedure that almost always returns “not-sure” may qualify as a verification procedure, but it is hardly an effective one. You cannot legislation against it, but you do not want to buy it.

Henceforth, we call a statement that describes a property of protocol execution a protocol property.

Two concerns need to be addressed in developing a verification procedure. First, in order that the procedure can “understand” and “analyze” its two inputs, both inputs have to be expressed in some predefined notation. Second, to establish that the input protocol property truly describes every execution of the input protocol definition, the procedure needs to check that some predefined proof obligations are met. If the procedure checks that all proof obligations are met, it returns a “yes” answer. Otherwise, it returns a “not-sure” answer.

The above discussion identifies the three main components of every method of protocol verification: some notation X for defining protocols, some notation Y for expressing protocol properties, and a set of proof obligations that need to be met in order to establish that a protocol property in notation Y truly describes every execution of a protocol definition in notation X.

As an example, consider a protocol verification method that takes a protocol definition on one input, and the protocol property “no deadlock state can be reached” on the other input. The proof obligation for the method is to generate the reachable states of the protocol, one by one, and...
check that none of them is a deadlock state. If all reachable states of the protocol can be generated and confirmed as non-deadlock states, the method returns a "yes" answer. Otherwise, the method returns a "not-sure" answer. (Note that this particular verification method can be modified slightly to return a "no" answer once a reachable deadlock state is generated. On the other hand, one can argue that there is no great distinction between a "not-sure" answer and a "no" answer: the protocol in question should not be used in either case.)

By the way, the proof obligation of checking that every reachable state of the input protocol satisfies some property is easy to implement. Hence, it has been adopted in many protocol verification methods in the literature. However, by adopting this particular proof obligation, the applicability of a verification method becomes severely limited. In particular, a verification method that adopts this proof obligation cannot be used to verify protocols with infinite or very large number of reachable states. More importantly, such a method cannot be used to verify protocols that are defined using some parameters, until these parameters are assigned specific values. Because of the inherent limitations of this proof obligation, the verification method that we discuss in this paper has a different set of proof obligations.

In the next three sections, we describe the three components of one particular method of protocol verification. A notation for defining protocols is given in Section 3. A notation for expressing protocol properties is given in Section 4. A corresponding set of proof obligations is presented in Section 5. Nothing distinguishes this particular method of protocol verification from other methods, except perhaps its sheer simplicity.

3. Protocol definition: keep it simple

The task of verifying a protocol can be simplified a great deal if one starts with a "nice" definition of the protocol. Unfortunately, I cannot give you a clear definition to what I mean by "nice". Nevertheless, I want you to recognize that not all definitions of the same protocol are equally nice: some definitions are easier to verify than others. (This is the same as saying that there are many ways to express the same meaning, but some expressions are easier to explain and be understood than others.) Thus, the next time you have difficulty verifying some protocol definition, consider changing the definition. (I have utilized this advice a number of times with great success.)

Here are some "tips" that have helped me and can help you come up with nice protocol definitions:

- **Discrimination**: Do not try to verify a protocol that is not defined by you or by someone you trust to have a reasonable experience in protocol verification. (Do not get entangled in the intellectual mess caused by the carelessness of others or their lack of experience.)
- **Abstraction**: When you define a protocol, get rid of every protocol feature that is not related to the features of most interest to you. (For example, when you define a sliding window protocol, the sequence numbers of data messages and acknowledgements are important, but all other message fields can be abstracted away as they do not contribute to the sliding window mechanism.)
- **Notation**: Use the simplest notation that you can think of to define your protocol. (Complex notation invites the three c's: complexity, complication, and confusion.)

The rest of this section is devoted to presenting a simple notation for defining communication protocols. The presentation concentrates mostly on the semantics of the notation rather than on any specific syntax. (The syntax that I do prefer is presented in Section 8, and illustrated by some examples in Section 9.)

A protocol is defined by a set of processes. Each process is defined by a set of local variables and a set of actions. Before we can define an action, we first need to define five concepts: channels between processes, message sequences in channels, send statements, receive statements, and protocol states.

For each pair \((p, q)\) of distinct processes in a protocol, there is a shared variable called the channel from \(p\) to \(q\), or \(C.p.q\) for short. This variable can also be referred to as an output channel of process \(p\) or as an input channel of process \(q\). Variable \(C.p.q\) is shared because, as discussed below, its value can be updated by executing specific actions of processes \(p\) and \(q\).
The value of $C_{p,q}$ is a finite sequence of messages of the form:

$m_1; m_2; \ldots; m_k$

where $m_1$ is the head message in the sequence, $m_k$ is the tail message in the sequence, and ";" is the sequence concatenation operator.

The value of $C_{p,q}$ can be updated by executing a send statement "send $m$ to $q$" in some action of process $p$; this execution causes a message $m$ to be added at the tail of the message sequence. The value of $C_{p,q}$ can also be updated by executing a receive statement "rcv $m$ from $p$" in some action of process $q$; this execution causes the head message $m$ to be removed from the message sequence. Execution of a receive statement is blocked if the message sequence is empty or if the head message in the sequence is not the same as the one identified in the receive statement. Because the message sequence is unbounded (though finite), execution of a send statement cannot be blocked.

A state of protocol $P$ is defined by a value (i.e., message sequence) for each channel in $P$, and by a value for each local variable of each process in $P$.

An action of a process $p$ in protocol $P$ is a pair (guard, step) such that the following three conditions are satisfied:

(1) Guard: The guard is a boolean expression that refers only to the head messages in the input channels of process $p$ and to the local variables of $p$.

(2) Step: The step consists of one statement that is defined recursively as one of the following: skip, assignment, nondeterministic selection, bounded loop, send, or receive statement, or a sequence of two statements. Each statement refers only to the local variables of process $p$.

(3) Atomicity: If the guard is true at some state of protocol $P$, then the step can be executed starting from this state without any blocking to its receive statements.

The atomicity condition needs some explanation. Executing a step consists of executing a finite sequence of statements because each loop statement in the step is bounded. Thus, if the execution of every receive statement in a step is guaranteed not to be blocked (as required by the atomicity condition), then execution of the step is guaranteed to terminate in finite time, and the action can be viewed as "atomic".

Satisfying the atomicity condition is not hard at all. In fact, we propose in Section 8 a specific syntax for an action that guarantees that the action satisfies this condition.

An action is enabled at some protocol state iff the action's guard is true at that state. If an action is enabled at some protocol state, then the action can be executed at this state. Executing an action at a state consists of executing the action's step starting from that state until the step terminates.

A computation of protocol $P$ is a sequence (state.1; action.1; state.2; action.2; \ldots) such that the following three conditions are satisfied:

(1) Execution: Each state.\(i\) is a state of $P$, and each action.\(i\) is an action in $P$ that is enabled at state.\(i\). Moreover, executing action.\(i\) starting from state.\(i\) can yield state.\(i+1\).

(2) Maximality: If the sequence is finite, then no action in $P$ is enabled at the last state in the sequence.

(3) Fairness: If the sequence is infinite, and if for some $i$, some action in $P$ is enabled at every state.\(j\) where $j > i$, then that action is action.\(k\) in the sequence, for some $k > i$.

Informally, a computation of protocol $P$ describes an "execution" or "run" of $P$ starting from an arbitrary state of $P$. The run is maximal: either it is indefinite, or it reaches a state after which no further execution of the protocol is possible. The run is also fair: each action in $P$ that is continuously enabled along the run is eventually executed.

In the definition of a computation, actions of different processes are assumed to execute in sequence, rather than in parallel as one might expect. The reason for this rather counter-intuitive definition is simplicity. If actions of different processes are assumed to execute in parallel, the task of verifying a protocol would become much harder because many more cases (related to the many ways in which parallel actions can overlap in time) would need to be considered in the verification. The preference of sequential computations over parallel ones can also be justified by an implicit understanding that if one proves a protocol correct assuming sequential computations, then the same protocol would be correct assuming parallel computations. (The fact that
this understanding is implicit and has not been proven explicitly indicates that some research is needed on this subject. See [1] and [8] for more discussion of this subject.)

4. Protocol properties: austerity helps

The question that we address in this section is how many and what types of protocol properties do we need to allow in our method? Let us suppose that we are interested in some $m$ property types. For each property type, we need to master a method for verifying that every execution (i.e., computation) of a protocol satisfies a property of that type. It would be nice, then, to keep $m$ as small as possible so that we do not have to master too many verification methods. (Austerity helps!)

In this presentation, we choose $m = 2$, and henceforth restrict our attention to only two types of protocol properties. This choice, $m = 2$, is largely based on our long experience with the properties of sequential programs. Indeed, the properties of sequential programs are of two types: annotation and termination. An annotation property of a program states that whenever an execution of the program reaches some specified point in the program text, the values of program variables satisfy some specified predicate when the program execution reaches the specified point. The termination property of a program states that any execution of the program is guaranteed to terminate. Influenced by these properties of sequential programs, we choose analogous properties for protocols: closure and recurrence. These properties are defined below, but first we need to define the notion of state predicate.

A state predicate of protocol $P$ is a function that has a binary value, true or false, at each state of $P$. A state of $P$ is an $r$-state iff the value of state predicate $r$ is true at that state.

Let $r$ be a state predicate of protocol $P$. $r$ is a closure in $P$ iff at least one state of $P$ is an $r$-state, and every computation of $P$ that starts with an $r$-state is infinite and all its states are $r$-states.

From this definition, if $r$ is a closure of some protocol $P$, then the execution of $P$ can be started at some $r$-state (because some state of $P$ is an $r$-state). Moreover, if the execution of $P$ does start at an $r$-state, then the execution is guaranteed to progress indefinitely and every state that is encountered during the execution is guaranteed to be an $r$-state. In other words, $r$ defines a nonempty and closed domain of indefinite execution for protocol $P$.

A closure property of $P$ specifies an execution domain for $P$ where at least one action of $P$ is to be executed infinitely often. To specify which actions in $P$ are to be executed infinitely often in the execution domain, we need a second property: recurrence. But before we present the recurrence property, we need first to introduce the notion of a transition.

A pair $(b, t)$ is a transition in protocol $P$ iff $b$ is a predicate that refers only to the local variables of some process $p$ in $P$, and $t$ is an action of the same process $p$.

A transition $(b, t)$ in protocol $P$ is enabled at some state of $P$ iff both predicate $b$ and the guard of action $t$ are true at that state. A transition that is enabled at some state can be executed at this state. Executing a transition $(b, t)$ at a state consists of executing the step of $t$ starting from this state. Note that executing a transition $(true, t)$ is identical to executing action $t$.

A set of transitions is reduced iff every pair of distinct transitions in the set have distinct actions. Each set of transitions can be reduced by replacing every pair of transitions $(b_1, t)$ and $(b_2, t)$ in the set by a single transition $(b_1 \lor b_2, t)$ where $\lor$ stands for "logical or".

Let $T$ be a reduced set of transitions in protocol $P$, and let $r$ be a closure in $P$. $T$ recurs within $r$ in $P$ iff every computation (state.1; action.1; ...) of $P$, where state.1 is an $r$-state, satisfies the following condition. There is a transition $(b, t)$ in $T$ such that for every $i$, there is $j > i$, where $(b, t)$ is enabled at state.$j$ and action.$j$ is $t$.

From this definition, if $T$ recurs within $r$ in protocol $P$ and if the execution of $P$ starts at an $r$-state, then at least one transition in $T$ is guaranteed to be executed infinitely often.

To summarize, the properties of a protocol $P$ consist of one closure property:

$r$ is a closure in $P$

and a number of recurrence properties:

$T.1$ recurs within $r$ in $P$

$\ldots$

$T.k$ recurs within $r$ in $P$
The closure property describes a closed domain $r$ of indefinite execution for protocol $P$. Each recurrence property describes a set $T_i$ of transitions of which one transition is guaranteed to be executed infinitely often within the execution domain $r$.

5. How to verify it

In this section, we discuss proof obligations that are sufficient to establish that a protocol $P$ satisfies a closure property ($r$ is a closure in $P$) and a recurrence property ($T$ recurs within $r$ in $P$). We start with the proof obligations for a closure property.

In order to establish that $r$ is a closure in $P$, one needs to establish the following three obligations:

1. **Witness:** Exhibit a state of $P$ at which $r$ is true; i.e. exhibit an $r$-state of $P$.
2. **Liveness:** Prove that at least one action in $P$ is enabled at each $r$-state of $P$.
3. **Stability:** Prove that if an action in $P$ is enabled at an $r$-state, then executing this action starting from this state is guaranteed to terminate at an $r$-state of $P$.

The witness obligation shows that at least one state of $P$ is an $r$-state. The liveness obligation guarantees that the last state in any finite computation is not an $r$-state. Finally, the stability obligation ensures that all states in a computation that starts with an $r$-state are $r$-states. Therefore, these three obligations together are sufficient to establish that $r$ is a closure in $P$.

Stating the proof obligations to establish that $T$ recurs within $r$ in $P$, one needs to exhibit a ranking function $f$ over $r$ and fulfill the following three obligations:

1. **No-retreat:** Prove that if any transition in $\sim T$ is enabled at an $r$-state of $P$, then executing this transition starting from this state does not increase $f$.
2. **Progress:** Prove that for any integer $k > 0$, there is a transition in $T$ that is enabled at every $r$-state where $f = k$, or there is a transition in $\sim T$ that is enabled at every $r$-state where $f = k$ and its execution starting from any such state decreases $f$.
3. **Conclusion:** Prove that there is a transition in $T$ that is enabled at every $r$-state where $f = 0$.

The no-retreat obligation shows that the value of $f$ cannot increase as long as no transition in $T$ is executed. The no-retreat and progress obligations together ensure that if $f > 0$ then some transition in $T$ is eventually executed, or the value of $f$ is eventually decreased. The no-retreat and conclusion obligations together guarantee that if $f = 0$ then one transition in $T$ is eventually executed.

The above definition of a ranking function can be extended into a function $f$ that assigns to each $r$-state $S$, a sequence of $n$ natural numbers, for some $n > 0$. Thus, $f.S = (x.1; \ldots; x.n)$ where each $x.i$ is a natural number. For this extended $f$, the condition $f.S < f.S'$ holds iff

$$f.S = (x.1; \ldots; x.n),$$
$$f.S' = (y.1; \ldots; y.n),$$
there is an $i$ in $1,\ldots,n$ such that $x.i > y.i$, and for every $j < i$, $x.j = y.j$.

It is straightforward to extend the earlier notion of a transition “decreasing” or “increasing” a ranking function $f$ to accommodate the extended $f$. Taking this extension into account, the above three obligations of no-retreat, progress, and conclusion remain the same.
6. The trouble with communication

Our view of communication has so far been idealistic. When one process \( p \) sends a sequence of messages to another process \( q \), the sent messages are stored in channel \( C.p.q \) in the same order they were sent until they have been received, also in the same order, by process \( q \). Unfortunately, "real life" does not abide by this idealistic view. In real life, sent messages can be reordered while in the channel. They can also be corrupted, i.e., transformed into erroneous messages in the channel, or can be lost, i.e., disappear completely from the channel. Thus, protocols should be designed to tolerate these common forms of communication faults.

In this section, we discuss how to expand our protocol properties in Section 4 in order to be able to state that a protocol can tolerate communication faults. We also expand our proof obligations in Section 5 to be able to prove that a protocol is fault-tolerant.

We refer to the actions defined in Section 3 as protocol actions, and so distinguish them from fault actions which we are about to define.

A fault action is a pair \((\text{guard}, \text{step})\) such that exactly one of the following three conditions is satisfied.

1. **Reorder:** The guard is a predicate whose value is true iff a channel in the protocol has two adjacent, distinct messages \("m1;m2"\), and the step is a statement that reorder these two messages into \("m2;m1"\).

2. **Corruption:** The guard is a predicate whose value is true iff a channel in the protocol has a message \( m \) other than the special message \("error"\), and the step is a statement that replaces message \( m \) with \"error\" in the channel.

3. **Loss:** The guard is a predicate whose value is true iff a channel in the protocol has a message \( m \), and the step is a statement that removes message \( m \) from the channel.

Fault actions are enabled and executed in exactly the same way as protocol actions are enabled and executed as defined in Section 2.

The concept of a computation of protocol \( P \) is extended such that a finite number of fault actions are executed along every computation of \( P \). To effect this extension, the execution condition in the definition of a computation (in Section 3) is modified such that each action \( i \) in the computation is either an action in protocol \( P \) or a fault action. The other two conditions in the definition, namely maximality and fairness, remain as they are and continue to refer to the actions of protocol \( P \) only.

Definitions of the two properties closure and recurrence remain as they are in Section 4. (Note, however, that each of these definitions now has a slightly different meaning due to the change in the definition of a computation.)

Of the six proof obligations in Section 5, only one, stability, needs to be strengthened; the other five remain as they are. The strengthened stability obligation is as follows:

**Stability under faults:** Prove that if an action of \( P \) or a fault action is enabled at an \( r \)-state of \( P \), then executing this action starting from this state is guaranteed to terminate at an \( r \)-state of \( P \).

This proof obligation along with the five remaining proof obligations in Section 5 are sufficient to establish that protocol \( P \) satisfies its closure and recurrence properties even if message reorder, corruption, or loss do occur.

7. It is time to discuss timeouts

In the previous section we have discussed how to verify that a protocol \( P \) is fault-tolerant, i.e. satisfies its properties even if communication faults do occur. It remains now to discuss how to define such a fault-tolerant protocol in the first place.

In general, protocols can be made fault-tolerant by three "standard" techniques: the addition of sequence numbers to messages, discarding received erroneous messages, and timeouts. The first technique can be used to detect and correct all occurrences of message reorder. The second technique can be used to transform each occurrence of message corruption to an occurrence of message loss. The third technique can be used to detect and correct all occurrences of message loss (provided that there is an upper bound on the time-to-live for every sent message).

The first two techniques can be defined using the protocol definition notation discussed in Section 3. Timeouts, however, cannot be defined using this notation. Therefore, some extension of the notation is in order.
The direct approach to accommodate timeouts in a protocol definition is to introduce some kind of a "global clock" to the protocol definition. The function of the clock is to keep track of "time passing" so that timeout actions can become enabled after enough time has passed. Unfortunately, if you try this direct approach, as we did, you will soon discover that it tends to complicate protocol definitions and their verifications. This situation has led us to the following indirect approach.

We extend the definition of a protocol action in Section 3 to include what we call timeout actions. Like a regular protocol action, a timeout action is a pair (guard, step), and its step is defined exactly like that of a regular protocol action. However, the guard of a timeout action is a state predicate of the protocol, as defined in Section 4, rather than a local predicate, as defined in the guard condition in the definition of a protocol action in Section 3.

Timeout actions are powerful. They can be used to detect the occurrences of global conditions in their protocols. For example, it is possible to define a timeout action that is enabled when and only when all channels in the protocol are empty and all other protocol actions are disabled. (This global condition usually indicates the occurrence of a message loss.)

Timeout actions are abstract. Their definitions describe when can the actions be executed and what are the effects of their executions, but they do not describe how the actions can be implemented. In particular, the definition of a timeout action includes a global condition (i.e., a set of protocol states) whose occurrence triggers the execution of the timeout action. The definition, however, does not describe how the occurrence of this global condition can be detected using real-time clocks. Describing how to implement timeout actions using real-time clocks, and verifying the correctness of these implementations is beyond the scope of our verification method. (This is the price we have to pay to keep our verification method clean and simple.)

8. What is in a syntax

The simple semantics of our notation deserves a simple syntax; simplicity is always our most coveted concern. The syntax defining a process in a protocol is as follows.

```
process ⟨process name⟩
var  ⟨var name⟩:⟨var type⟩, . . . ,
     ⟨var name⟩:⟨var type⟩
begin ⟨action⟩[ ] . . . [ ]⟨action⟩
end
```

Note that declarations of different local variables are separated by commas, while definitions of different actions are separated by boxes "[ ]".

We adopt a PASCAL-like syntax to declare the local variables of a process. In the most part, local variables are of five types: boolean, enumerated, range, integer, and array. We allow arrays to be infinite; see the first protocol example in Section 9.

An action has the following syntax.

```
⟨guard⟩ → ⟨step⟩
```

For simplicity, we adopt the restriction that a step has at most one receive statement. The guard and step of an action should also satisfy the three conditions of guard, step, and atomicity in Section 3.

In order for an action to satisfy the atomicity condition, if the action's step has a receive statement (rcv m from q), then the action's guard should test that m is indeed the head message in the channel from process q. A possible syntax for such an action is as follows:

```
head m from q → rcv m from q; S
```

where head m from q is a predicate whose value is true when and only when m is the head mes-
sage in the channel from process \( q \). Recall that \( S \) is a statement that contains no receive statements because of our restriction that each step has at most one receive statement. For convenience, we use the following syntax as a short-hand for this action form:

\[
\text{rcv } m \text{ from } q \rightarrow S
\]

This short form guarantees that that the atomicity condition is met.

We are now ready to present some protocol examples and discuss how to verify them along the method outlined in this paper.

9. For example ...

We discuss in this section three examples. In each example, we use the notation in Sections 3, 7, and 8 to define some protocol. We then use the concepts of closure and recurrence in Section 4 to state some properties of that protocol. Finally, we show that the proof obligations in Sections 5 and 6 are met, thus establish that the protocol indeed satisfies its intended properties.

Protocol 1

This protocol consists of two processes \( p \) and \( q \). Process \( p \) has an infinite bit-array named \( \text{"data"} \), and process \( q \) has an infinite bit-array named \( \text{"rcvd"} \). Process \( p \) sends the bits in \( \text{"data"} \) one by one to process \( q \) which stores them in \( \text{"rcvd"} \). Each \( \text{"data"} \) bit is sent as a sequence of two bits: a 0 bit is sent as a sequence 0;1 and a 1 bit is sent as a sequence 1;0. On receiving two successive bits, process \( q \) decodes the two bits into one bit then stores the bit in array \( \text{"rcvd"} \). Process \( p \) is defined as follows.

\[
\text{process } p
\]

\[
\text{var } \begin{array}{l}
data: \text{array [integer]} \text{ of 0..1,} 
 i: \text{integer,}
\end{array}
\begin{array}{l}
\text{begin}
\text{true } \rightarrow \text{ if } data[i] = 0 \rightarrow \text{ send 0 to } q; 
\text{send 1 to } q 
\text{ }\text{}\text{ [ } data[i] = 1 \rightarrow \text{ send 1 to } q; 
\text{send 0 to } q 
\text{ fi; } i:= i+1
\end{array}
\]

Process \( p \) has only one action that is always enabled because its guard is true. On executing this action, the next \( \text{"data"} \) bit is checked: if it is 0, \( p \) sends 0 followed by 1; otherwise \( p \) sends 1 followed by 0. Process \( q \) is defined as follows.

\[
\text{process } q
\]

\[
\text{var } \begin{array}{l}
\text{rcvd: array [integer]} \text{ of 0..1,} 
 j: \text{integer,}
\text{first: boolean,} 
 b: 0..1
\end{array}
\begin{array}{l}
\text{begin}
\text{rcv } b \text{ from } p \rightarrow \text{ if first } \rightarrow \text{rcvd}[j], j, \text{first}
\text{ }\text{}\text{ [ } b, j + 1, \text{false}
\text{ }\text{}\text{ [ } j \rightarrow \text{first } \rightarrow \text{first } := \text{true}
\text{ fi; } j:= j + 1
\end{array}
\]

Process \( q \) has only one action. This action is enabled when and only when there is at least one bit in the channel \( \text{C.p.q} \). On executing this action, the head bit in \( \text{C.p.q} \) is received and the local boolean variable \( \text{"first"} \) is checked to determine whether this bit is the first one in a two-bit sequence encoding one \( \text{"data"} \) bit. If it is, the bit is stored in the next location of array \( \text{"rcvd"} \); otherwise, the bit is discarded.

Define \( r1 \) to be the following state predicate of Protocol 1:

\[
r1 = (\text{first } \land \text{dbl.c } \land
\text{seq.(data, } i-1) = \text{seq.(rcvd, } j-1); \text{even.c})
\land (\text{v } \land \text{~first } \land \text{~dbl.c } \land
\text{seq.(data, } i-1) = \text{seq.(rcvd, } j-1); \text{odd.c})
\]

where

\[
c = \text{the bit sequence } \text{C.p.q} \text{ in the channel from } p \text{ to } q
\]

\[
dbl.c = \text{true}
\text{if } c \text{ has zero or even number of bits}
\]

\[
dbl.c = \text{false}
\text{if } c \text{ has odd number of bits}
\]

\[
even.c = \text{empty sequence}
\text{if } c = \text{empty sequence } \vee c = b1
\]

\[
even.c = b2; \text{even.x}
\text{if } c = b1; b2; x
\]
odd.c = empty sequence
   if c = empty sequence
      odd.c = b1
      if c = b1
      odd.c = b1; odd.x
         if c = b1; b2; x

seq.(y, k - 1) = empty sequence
if y is an infinite bit array and k = 0
   seq.(y, k - 1) = y[0]; y[1]; ...; y[k - 1]
if y is an infinite bit array and k > 0
Define T1 to be the following set of transitions of Protocol 1:
T1 = {(first, t.q)}
where t.q denotes the (only) action of process q.
Next, we prove the following two assertions:

r1 is a closure in Protocol 1
T1 recurs within r1 in Protocol 1

To prove that r1 is a closure in Protocol 1, we need to verify that the three proof obligations of witness, liveness, and stability are fulfilled. The protocol state:

\((i = 0 \land C.p.q = \text{the empty sequence} \land j = 0 \land \text{first})\)

satisfies r1; hence the witness obligation is fulfilled. The action of process p is enabled at every protocol state; thus the liveness obligation is fulfilled. Finally, the action of process p cannot falsify any of the two disjuncts of r1, and the action of process q does establish one of the disjuncts of r1; thus the stability obligation is fulfilled.

In order to prove that \(T1 = \{(\text{first}, t.q)\}\) recurs within r1 in Protocol 1, we exhibit the following ranking function.

\(f1 = 0 \quad \text{if } r1 \land \text{first} \land \#C.p.q > 0\)
1 \quad \text{if } r1 \land \text{first} \land \#C.p.q = 0
2 \quad \text{if } r1 \land \sim \text{first} \land \#C.p.q > 1
3 \quad \text{if } r1 \land \sim \text{first} \land \#C.p.q = 1

where \#C.p.q is the number of messages (in this case bits) in the channel from process p to process q.

It remains now to show that \(f1\) fulfills the proof obligations of no-retreat, progress, and conclusion. The complement of \(T\) is the set \(\sim T = (\text{true}, t.p), (\sim \text{first}, t.q)\). It is straightforward to show that if any transition in \(\sim T\) is executed starting from a state where \(f = k\), then the result is a state where \(f = k\) or \(f < k\); thus the no-retreat obligation is fulfilled. The progress obligation is fulfilled by the following three facts. First, transition \((\text{true}, t.p)\) in \(\sim T\) is enabled at every state where \(f = 3\) and its execution at any such state yields a state where \(f = 2\). Second, transition \((\sim \text{first}, t.q)\) in \(\sim T\) is enabled at every state where \(f = 2\) and its execution at any such state yields a state where \(f = 0\). Third, transition \((\text{true}, t.p)\) in \(\sim T\) is enabled at every state where \(f = 1\) and its execution at any such state yields a state where \(f = 0\). The conclusion obligation is fulfilled by the fact that transition \((\text{true}, t.q)\) in \(T\) is enabled at every state where \(f = 0\).

Protocol 2

Like the previous protocol, Protocol 2 consists of two processes p and q. Each process sends a request message to the other process then waits for a reply message before sending the next request message. Process p is defined as follows. (Process q is defined in a symmetrical way.)

process p
var ready : boolean
begin
   ready \rightarrow ready := false; send request to q
   [ ] rcv reply from q \rightarrow ready := true
   [ ] rcv request from q \rightarrow send reply to q
end

Define r2 to be the following state predicate of Protocol 2.

\(r2 = (X.p + \text{request} \#C.p.q + \text{reply} \#C.q.p = 1) \land (X.q + \text{request} \#C.q.p + \text{reply} \#C.p.q = 1)\),
where
\(X.p = 0\) if \sim ready.p
1 if ready.p
\text{request} \#C.p.q = \text{number of request messages in } C.p.q
\text{reply} \#C.q.p = \text{number of reply messages in } C.q.p

Define T2 to be the following set of transitions of Protocol 2.

\(T2 = \{(\text{true}, t.p.0)\}\)
where \(t.p.0\) denotes the first action in process p.

It is straightforward to show the following assertions.
r2 is a closure in Protocol 2
T2 recurs within r2 in Protocol 2

We leave it to the reader to try to prove these two assertions, but here is a hint: the following ranking function can be used in proving the second assertion:

\[
f_2 = \begin{cases} 
0 & \text{if } r2 \land \text{ready}.p \\
1 & \text{if } r2 \land \text{head}.(C.q.p) = \text{reply} \\
2 & \text{if } r2 \land C.q.p = \text{request}; \text{reply} \\
3 & \text{if } r2 \land \text{head}.(C.p.q) = \text{request} \\
4 & \text{if } r2 \land C.p.q = \text{reply}; \text{request}
\end{cases}
\]

where head.(C.q.p) denotes the head message in the channel from process q to process p.

Protocol 3

It is required to modify Protocol 2 such that the modified protocol, Protocol 3, tolerates message reorder, loss, and corruption. As it turns out, Protocol 2 tolerates message reorder because r2 is stable under message reorder actions; i.e., executing a reorder action starting from an r2-state is guaranteed to terminate at an r2-state.

In order to make r2 stable under message loss actions, it should be modified as follows.

\[
r3 = (X.p + \text{request} \# C.p.q + \text{reply} \# C.q.p \leq 1) \land (X.q + \text{request} \# C.q.p + \text{reply} \# C.p.q \leq 1)
\]

In other words, r3 is obtained from r2 by replacing each equal to ≤ .

Unfortunately r3 does not fulfill the liveness obligation for Protocol 2. This is because no action in Protocol 2 is enabled at the following r3-state.

\[
(X.p + \text{request} \# C.p.q + \text{reply} \# C.q.p = 0) \land (X.q + \text{request} \# C.q.p + \text{reply} \# C.p.q = 0)
\]

To make r3 fulfill the liveness obligation under message loss, we add the following timeout action to process p and a symmetrical timeout action to process q.

\[
\text{timeout } X.p + \text{request} \# C.p.q + \text{reply} \# C.q.p = 0 \rightarrow \text{send request to q}
\]

If message corruption can occur, then r3 still does not fulfill the liveness obligation because no action in the modified protocol is enabled for execution at any r3-state where

\[
\text{head}.(C.p.q) = \text{error} \land \text{head}.(C.q.p) = \text{error}
\]

To make r3 fulfill the liveness obligation under message corruption, we add the following action to process p and a symmetrical action to process q.

rcv error from q → skip

These actions transform every occurrence of message corruption into an occurrence of message loss (which can then be dealt with using the timeout actions).

10. Where do you go from here

I have described in this tutorial useful “tools” for defining and verifying protocols: a notation for defining protocols, a notation for stating protocol properties, and a sufficient set of proof obligations for verifying that a given protocol satisfies a given property. Unfortunately, a tool is not useful until it is frequently used. And it cannot be frequently used until the would be user feels comfortable about using it. And this cannot happen until the would be user has used the tool many times before (possibly with great difficulty in the beginning). In summary, to learn how to use a tool, you have to use it! This may sound like a catch, but that is precisely how you learnt to drive your car.

My advise to you is to practice ... practice ... practice. Choose a protocol from the literature, and define it using the notation that I have described here, then try to state and verify its closure and recurrence properties. Make sure that the protocols you choose at the beginning are the simplest that you can think of; then work yourself slowly into harder and harder protocols. All along remember that the objective of these exercises is to make you familiar and comfortable with these formal tools and how to apply them, rather than to check the correctness of “industrial-strength” protocols. (That will come later.) Finally, do not get discouraged by rumors and allegations that protocol verification is hard and can never be put to use: nothing can be that pretty and that useless at the same time.
The protocol verification examples in this tutorial are intentionally simple. For more involved examples, the reader is encouraged to look at three recent papers [2,4,5]. In each of these papers, the same verification method that we have outlined here has been used to verify some non-trivial and interesting protocols.

Our protocol verification method does not address three important topics: how to define the "interface" between a protocol and the rest of the world, how to combine two or more protocols together to form a multifunctional protocol, and how to state and verify the realtime requirements of a protocol. The research in these important topics is still going-on, and final answers are far from being settled. Anyway, a reader who is curious about these topics can find some answers in [3] and [7].

Finally a warning! This paper is a tutorial. It is intended only to educate the reader about one particular method of protocol verification. Although this method comes highly recommended (I have used this method in the past, and plan on using it for sometime in the future), the reader should be aware that there are many other methods of protocol verification. For detailed surveys of some of these methods, let me recommend [6] and [9]. Best wishes.

Acknowledgements

Many people have taught me so much for this paper to be as short. For that, I would like to thank E.W. Dijkstra, W.H.J. Feijen, D. Gries, S.S. Lam, J. Misra, and A. Shaheen. I would like also to thank my students A. Arora and S. Chodrow for listening and sometimes protesting. I would like also to thank my three anonymous referees; their cruelty helped and has been appreciated. Last but by no means least, I would like to thank Bjorn Pehrson, Bengt Jonsson and Joachim Parrow for inducing me to finally write this paper and get it over with. Without their support, this paper would not have been possible.

References