VERIFICATION OF COMMUNICATIONS PROTOCOLS AND ABSTRACT PROCESS MODELS

Benedetto Lorenzo Di Vito

Technical Report 25 August 1982

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

Institute for Computing Science
The University of Texas at Austin
Austin, Texas 78712

acknowledgements

I would like to heartily thank my co-supervisors, Don Good and Simon Lam, for their guidance and advice during the course of my research. I am particularly indebted to Don Good for providing me with financial support through the Institute for Computing Science. The other members of my dissertation committee, Jim Browne, Mani Chandy and Jay Misra, were a valuable source of wisdom and encouragement. I am fortunate to have had such a distinguished committee.

Several other individuals have contributed in various ways to the completion of my thesis. Carl Sunshine and Mohamed Gouda graciously offered critiques of parts of this work while it was in progress. I have had helpful discussions with Bob Boyer and J Moore on the use of their theorem prover. Several friends and colleagues at the Institute for Computing Science provided assistance by reading draft papers and listening to half-baked ideas. They include Mike Smith, Rich Cohen, Bill Bevier and Bill Young. To all of these people I express my gratitude.

My wife Lynn deserves special thanks for her support, both financial and emotional. For the last few years she has endured the life of a dissertation widow, which happily is now coming to an end. Not to be forgotten are my parents, Tony and Mary. The important lessons of life are still learned from one's parents, and the values they instilled in me have played a substantial role in this achievement.

Abstract

Communications protocols are crucial for the reliable exchange of information in distributed systems. In this dissertation, we consider the problem of formally specifying and verifying properties of protocol systems. Such systems are modeled by hierarchies of concurrent processes, where interprocess communication is achieved by message passing rather than through arbitrary shared variables. Based on this model, a methodology is developed for mechanically assisted protocol analysis.

The Gypsy methodology for concurrent program verification is the point of departure for much of this work. Specialized methods applicable to protocols are derived from the Gypsy methods. Behavior of protocol modules is specified in a fairly abstract manner using a state transition paradigm, thus avoiding a highly procedural form of specification. Protocol services are specified by means of assertions over message histories. Proof techniques are introduced for verifying safety properties of the process models. In addition, a specification and assertion language is developed. This language emphasizes features and operations useful for expressing protocol oriented concepts and constructing proofs about them.

An important aspect of this work is use of machine assisted analysis, most notably the use of mechanical theorem proving. A strategy for applying a particular automatic theorem prover, the Boyer-Moore prover, to protocol verification problems is put forth. A consequence of this strategy is the accumulation of a large body of proved lemmas, constituting a rudimentary deductive theory for protocols. With this theory, the methodology has successfully been applied to a pair of sample transport protocols. These include the Stenning protocol and an abstraction of the data transfer function of TCP.

Chapter 1
INTRODUCTION

As interest in distributed systems continues to grow, research on data communications technology is becoming increasingly important. Due to the inherent complexity of communications systems, reliable construction of such systems is a difficult task. The need for effective analysis and design techniques is paramount. Without them our ability to develop systems having predictable behavior would be seriously limited. The analysis problems facing us are formidable, but not by any means insurmountable.

At the heart of any communications system is its protocol architecture. Protocols are the procedures that control the orderly exchange of information in a network [McQuillan 78]. Recently, the importance of applying formal methods to the problems of communications protocol analysis and design has been recognized. A number of approaches to protocol specification and verification have been proposed, using a large variety of models and techniques. Surveys of this work are readily available [Sunshine 79, Bochmann 80]. Several distinct lines of research have subsequently emerged, although it is safe to say that we have not yet heard the final word.

In this thesis, we report on a new methodology that has been developed to address the problem of protocol analysis. It is based on techniques for verifying systems of concurrent processes that communicate by message passing. Only verification of safety properties is considered at this time; we hope to consider liveness properties in the future. Included is a novel method for stating abstract behavioral specifications of protocol modules. It avoids the highly procedural forms of specification that are in common use today. Also, a significant aspect of the methodology is the use of mechanical theorem proving tools to actually carry out the proofs. We view this as important not so much for the obvious advantages of automation, but rather for the elimination of errors that can result from doing tedious proofs by hand.

Overall we would characterize the methodology as being an integration of many tools and techniques from a diversity of sources. Among them are verification of concurrent and sequential processes, applicative language design, state transition models, decision table techniques, deductive theory building and mechanical theorem proving. Particularly important is the integration of techniques for modeling concurrent processes and state transition systems into a unified framework. The manner in which these ideas have been forged together reflect our belief that an effective methodology is the result of a delicate balance between theoretical and practical considerations.

Much of this work is based on concepts found in Gypsy [Good 77, Good 78]. The Gypsy methodology, developed at the University of Texas at Austin, is a highly successful methodology for specifying, verifying and implementing concurrent programs. An initial attempt at protocol verification was performed as a straightforward application of the Gypsy methods and automated tools [DiVito 81]. With additional help from the AFFIRM verification system [Musser 80, Gerhart 80], the experiment was successfully completed. Nevertheless, several aspects of this approach were somewhat less than satisfying, most notably the procedural form of specifying protocol behavior. This situation led to the search for a new methodology that was more suited to the special needs of protocol work. This thesis is the end result of that search.

Essentially what was done was to start with some of the better ideas of Gypsy and combine them with a few additional ones to create a new integrated methodology. Unfortunately, this meant that we could not make

direct use of Gypsy's automated tools. We did, however, take the opportunity to try other theorem provers and found that we could make good use of the Boyer-Moore theorem prover [Boyer 79]. At the present time we are using fully mechanical theorem proving tools and a partially automated verification condition generator. Other work is currently done by hand, but the construction of additional tools to help with specification and analysis is certainly a doable task. Because our work has been mainly exploratory research, development of tools has been done only to demonstrate feasibility.

While the theory of verification is often quite straightforward, it is usually very difficult to apply in practice. For this reason, we follow a basic principle of problem solving: if the general problem is too difficult, try to solve restricted versions of it. The most obvious step toward applying this principle has already been taken, namely the narrowing of our focus to a class of protocol problems. In the rest of our work, we will apply this principle repeatedly. A major theme of our general approach is to impose restrictions and then exploit the special properties that accrue from those restrictions. In exchange for sacrificing generality we hope to increase our chances of obtaining results. Verification is sufficiently difficult to justify extensive specialization.

Although the primary focus of this research has been analysis of protocols, the methods developed are clearly applicable to more general problems. The techniques are based on a restricted form of concurrent process model. There certainly are other distributed system applications that could make use of such a model and take advantage of our methods. Almost any process acting as a "server," whose behavior can be described as event-driven, could be readily specified and analyzed by our techniques. A fundamental limitation in practice would be the complexity of the required processing, which manifests itself through a need for nontrivial data structures and operations.

Let us now outline the remaining chapters of this dissertation. In Chapter 2 we present the basic approach for modeling and specifying protocol systems using concurrent processes. The Gypsy model of concurrency underlies this approach. The corresponding verification techniques are described in Chapter 3. Modular verification is emphasized as a practical goal.

Chapter 4 deals with the design of a specification (assertion) language. Stressed are generic language features and the use of data structures that facilitate proof by automatic theorem provers.

A primary objective of the methodology has been to develop tools and techniques for mechanized analysis. Chapter 5 describes the use of existing tools and the development of new tools for analysis.

The methodology has been successfully applied to prove properties about a pair of transport protocols. One is the protocol introduced by Stenning [Stenning 76] and the other is based on the data transfer functions of TCP [Postel 80]. Chapter 6 presents the details of this work.

Finally, in Chapter 7 we discuss some proposed extensions to the basic specification and verification methods. The motivation for these is the handling of connection management features and other state dependent kinds of processing.

Chapter 2
MODELING AND SPECIFICATION

Many models of communication protocol systems are currently in use today [Sunshine 81]. These models vary considerably in their degree of formalism and level of abstraction. We propose new modeling and specification techniques with a high degree of formalism that enable modeling of systems at an intermediate level of abstraction. We feel that this approach leads to a sound and effective methodology for verification while still allowing construction of fairly realistic models of protocols.

2.1 Philosophy of Specification

Specification is a highly loaded word, which is subject to many preconceived interpretations. For this reason, we begin this chapter by explicitly stating our notions of specification.

Specification techniques can only be meaningfully discussed in the context of an expressed purpose. Only when definite reasons for using formal specifications are stated is it possible to evaluate the usefulness of specification techniques. We can cite three principal reasons for formalizing protocol specifications.

1. *Definition*. An unambiguous definition of a protocol is important as a means of communication. This includes both communication between a protocol designer and others as well as between a designer and himself. Naturally, a companion informal specification is necessary to aid understanding, but by itself is too imprecise.

2. *Analysis*. Having completed the design of a protocol, a designer would typically like to perform various kinds of analyses on it. These involve discovering what properties a protocol has or ensuring that certain intended properties are actually present. This latter type of analysis is termed verification and is the main subject of our research. Formal specifications are the objects upon which these analyses are performed.

3. *Implementation*. After a protocol has been sufficiently well analyzed, its various components will be implemented. There can potentially be many different implementations for a single component. The formal specification is the standard against which these implementations are measured. Each must be in conformance with the standard to ensure that proper communication takes place.

When specification goals conflict, as is often the case, compromises must be made to adequately achieve all goals. It is difficult to arrive at a specification method that serves all the purposes stated above equally well. Simultaneously satisfying (2) and (3) is particularly troublesome. Formalisms that lend themselves to analysis tend to be very abstract. On the other hand, specifications from which implementations can be easily derived must be more concrete. Achieving a proper balance is therefore crucial to the overall effectiveness and usefulness of a specification methodology.

It is also important that a specification places appropriate constraints on the range of possible implementations. Many authors use an "abstract program" approach to model protocols. This involves exhibiting procedures in a conventional high level programming language to serve as a "specification." We

regard this approach as unsatisfactory because of the amount of inessential, low level detail that is introduced. In effect, it overconstrains implementations.

Conversely, a highly abstract model may leave an implementor with too much leeway. Often such models are used to facilitate analysis with respect to a certain property or class of properties. This is fine if our only goal is analysis. If, however, we wish to state a true specification, then we need to be concerned with the larger problem. A protocol designer might want events processed in a certain way to ensure that an acceptable level of performance is achieved or to ensure that one protocol entity does not unduly consume the resources of another. In essence, an implementor should only have the freedom to make decisions that affect a local execution environment; decisions that can affect remote execution environments or the overall operation of communication are the province of the protocol designer. Thus, a specification technique designed to meet the previously stated goals should not permit implementations to be underconstrained.

Sunshine has proposed a general model for specifying protocol systems [Sunshine 79], which we adhere to in the work that follows. In this model, a protocol layer is viewed as an *abstract machine*. This protocol machine provides services to the next higher level protocol layer. Services can be obtained by issuing commands to the protocol machine. A *service specification* is stated that describes the effects that these commands have on the behavior of the machine. The general characteristics of the machine model are deliberately left vague so as to accommodate different specification styles. Commonly chosen are methods based on state transition models. Our preference is for a process oriented model.

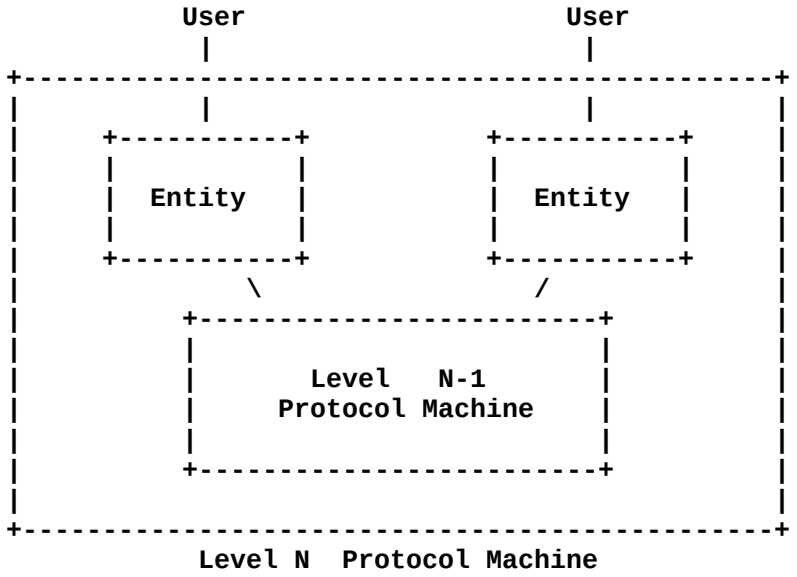


Figure 2-1: General model of protocol systems.

In the general framework, a level N protocol machine is decomposed into a set of communicating entities and a level N-1 protocol machine (Fig. 2-1). The entities are the protocol modules that actually do the work for protocol layer N. These entities realize the service primitives for level N and communicate among themselves by calling on the services of the level N-1 protocol machine. A *protocol specification* is stated that describes the effects of these events internal to level N. Demonstrating that a set of modules, which is in compliance with the protocol specification, satisfies its service specification is the protocol verification problem.

Our approach to modeling protocols is consistent with that outlined above. A protocol machine is modeled as an *abstract process*. Such a process is an active entity with explicit inputs and outputs. A process communicates asynchronously with its external environment by passing messages through its inputs and outputs. Message passing is the realization of the service primitives in the protocol machine model.

As the underlying model of computation, the abstract process model determines our notions of service specification and protocol specification. The process model uses no concept of global state, abstract or

otherwise. Instead, semantics is specified by relating the flows of messages through the process inputs and outputs. It is in this sense that we have a service specification. An important consequence of this arrangement is that we can usually express any properties we wish to show about a protocol directly as its service specification. This is in contrast to many state machine models where it is often necessary to prove properties about a specification to be convinced that it captures the intended behavior. In our approach, the service specification *is* the set of properties of interest.

A process representing protocol layer N is composed of a number of subprocesses. Some of these represent level N entities or modules and one or more represent the level N-1 protocol machines(s). A protocol specification in this model consists of abstract definitions for the level N protocol modules. Process behavior is specified using a state transition approach for such definitions. In this way we have integrated state transition concepts with parallel process concepts.

The precise definition of our modeling and specification techniques forms the rest of this chapter. Although our emphasis is on the modeling of protocol systems, it should be clear that these methods could also be useful for other distributed system applications.

2.2 Abstract Process Model

Our process model is similar to that of Gypsy [Good 77, Good 78]. It allows specification of hierarchical process structures with communication by message passing. With these hierarchies viewed as trees, their leaves correspond to the protocol entities, for example, transport stations for a transport layer protocol. An entire hierarchy corresponds to a protocol layer, or protocol machine in the protocol machine model. We proceed with a detailed explanation of the process model for protocol systems.

2.2.1 Basic concepts

Active computing entities are known as *sequential processes*. Each such process is assumed to be resident on a single processor and communicates with its environment (other processes) via message buffers. A message buffer is a finite size queue[1] that connects two sequential processes, one sending to it and the other receiving from it. No other means of data sharing is allowed; all variables within a process are strictly local. Note that a buffer may be of size zero, in which case the message passing regime would be similar to that of Communicating Sequential Process (CSP) models [Hoare 78]. A sequential process is assumed to be nonterminating.

We can build hierarchical process models using *concurrent processes*. A concurrent process consists of several subprocesses plus a number of local buffers to interconnect them. The subprocesses may themselves be either sequential or concurrent. Thus a concurrent process may be viewed as a tree structured object, where the leaves represent sequential processes (executing entities) and the nonleaf nodes represent concurrent processes. A concurrent process should be regarded as a virtual structure because its components are distributed and it does not embody a single locus of control; it is simply an organized collection of lower level processes and buffers that are aggregated for specification purposes.

A protocol machine can be represented by a concurrent process as shown in Fig. 2-2. This example shows a simple data transfer protocol, such as the Stenning protocol [Stenning 76]. The transport service is modeled as a concurrent process with three subprocesses. The sender and receiver are the protocol modules, which would be defined as sequential processes; the medium process represents the next lower level protocol machine, another concurrent process. The single input and single output of the protocol machine are depicted as message buffers source and sink. Four internal buffers are provided for the sender and receiver to interact with the transmission medium, and hence, with each other. Note that a buffer may be an input or output at several different levels in a process hierarchy. In this example, for instance, the source buffer is an input both to the

[1]We assume buffers are bounded but unbounded buffers would work as well. Our methods for proving safety properties do not depend on the use of unbounded buffers so we prefer to use the more realistic bounded buffer model.

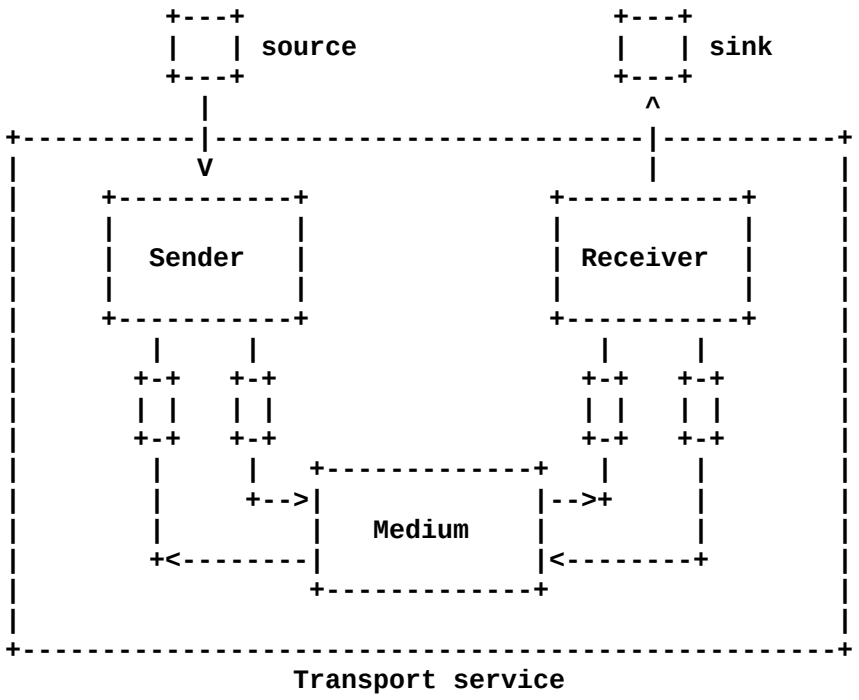


Figure 2-2: Protocol machine as a concurrent process.

sender and to its parent process, the transport service. It is said to be an external input of the transport service process.

Objects that flow through a message buffer are typed data objects, that is, all messages are elements of a single data type. A language for defining data types, objects and operations on them will be presented in Chapter 4. For now we just point out that each buffer will have associated with it the type of its data objects. Buffers are the one and only one mechanism for interprocess communication in our model. They represent shared data structures and consequently require that the operations of sending to a buffer and receiving from a buffer be atomic. In other words, it is assumed that send and receive operations exclude each other so that no inconsistent state can result.

2.2.2 Concurrent process definitions

A necessary evil of any formal method is well defined notation. Pictures of process structures are good for imparting an intuitive understanding of the model, but if we ever hope to achieve mechanized analysis we must have some machine readable formalism. To this end, we posit some simple notation for precisely defining our process models. No tools for reading or analyzing this language have yet been implemented, but it is clear that it would be a straightforward task to do so.

The general style for our process definition language is borrowed from the Gypsy programming language. Gypsy in turn has a Pascal-like syntax for most of its features. We continue in this tradition. This gives our definitions a more operational flavor than many of the abstract models of protocols. But they are also less operational than the strict programming language models. This is no accident -- we feel we have achieved a good compromise between abstractness and realism.

The transport service process may be given a concurrent process definition, as shown in Fig. 2-3. This is a realization of the structure shown in Fig. 2-2. There are three major parts to a concurrent process definition.

1. The process header identifies the input and output buffers and the types of objects that flow through them.

```
process transport_service (input  source: message;
                           output sink: message) =
  begin
     buffers (sndr_pkt, rcvr_pkt: packet;
              sndr_ack, rcvr_ack: natural);
     cobegin
        sender (source, sndr_ack, sndr_pkt);
        receiver (rcvr_pkt, sink, rcvr_ack);
        medium (sndr_pkt, rcvr_pkt, rcvr_ack, sndr_ack);
     end
  end
```

Figure 2-3: Transport service process definition.

2. The internal buffers are declared together with their element data types. These local buffer names will be meaningful only within this process definition.

3. The subprocess structure is introduced by means of a "cobegin" statement. Each subprocess is "called" by passing buffers as its parameters. These may be either internal or external buffers.

The general form for a concurrent process definition is shown in Fig. 2-4. Note that multiple instances of a subprocess may be called in the cobegin statement.

```
process P (inputs  IB1: IT1; ...; IBm: ITm;
           outputs OB1: OT1; ...; OBn: OTn) =
  begin
     buffers (B1: BT1; ...; Bk: BTk);
     cobegin
        P1 (LB1,1, ..., LB1,np(1));
        . . .
        Pq (LBq,1, ..., LBq,np(q));
     end
  end
```

Figure 2-4: General form of concurrent process definition.

The implied semantics of the process definition is that all subprocesses named in the cobegin statement are started simultaneously and thereafter run forever. We are not concerned with modeling processes that terminate, although termination is certainly a capability that could be added. If any of the subprocesses is itself a concurrent process, its cobegin statement is similarly "executed." The result is that an entire process hierarchy begins operation at one time and no part of it ever terminates. Practically speaking, the component processes are physically distributed and cannot actually be started synchronously. This is not a real problem, though, since we can imagine that the start up instant coincides with the earliest start time of any of the subprocesses. Conceptually we regard the others as having started at this point and being inactive during the interval just before they began execution.

We will shortly describe how sequential process definitions are constructed. These form the actual protocol specifications. The other subprocess that appears in the example above is the transmission medium process, the lower level protocol machine. Here the only necessary step is to declare its input/output interface. The medium process may be partially defined by

```
process medium (input  lf_in:  T1;
                output rt_out: T1;
                input  rt_in:  T2;
                output lf_out: T2) = pending
```

The above declaration contains only a process header and states that the actual definition is pending. This expresses the fact that we are unconcerned with the internal structure of the medium. All we care about is that it have the inputs and outputs cited in the process header.

Observe from Figures 2-2 and 2-3 that there is nothing distinguishing the form of the sender and receiver processes from that of the medium process. We intend the sender and receiver to be sequential processes, but at the external process interface this information is hidden. It is precisely this characteristic of the model that allows us to easily build hierarchical process structures. More important is that this aspect of the model leads to a modular proof organization, which will be of great benefit in dealing with complex behaviors.

2.2.3 Restrictions and limitations

There are some simple semantic restrictions on the model that have yet to be stated. These include:

- Buffer types must match. The sending process, the receiving process and the buffer declaration itself all refer to the type of data object that may be passed. These must all agree for a process structure to be well formed.
- Within one level of a process hierarchy, only one process may access each end of a buffer. More precisely, input access to a buffer may be given to only one port of one subprocess. The same is true of output access. Proof rules for concurrency are simplified by this restriction.
- All buffers must be explicitly declared either as inputs, outputs or internal buffers. There is no analog of the global variable. Nothing is implicitly inherited from the levels above.

The concurrent process model is simple and powerful, but also limited. Perhaps its most significant limitation is that all interprocess communication is via message passing. This means that all service primitives of protocols are modeled by this one mechanism. Real implementations of protocol modules will use a variety of techniques: procedure calls, supervisor calls, traps, interrupts, etc. If a command requires the use of a procedure call mechanism, for example to return possible error codes or other status information, then it will be more difficult to model with message passing primitives.

As often happens, though, the greatest weakness of a method is also its greatest strength. The power we get in return for accepting this limitation is considerable. The success of the Gypsy methodology is evidence in support of this claim. Our model is even simpler and more specialized than Gypsy. Therefore we expect to reap big rewards. Primarily these have come from two mutually reinforcing sources: the technique for specifying sequential process behavior, to be described in the next section, and some very simple and elegant verification methods.

2.3 Protocol Specifications

A protocol is often defined as the set of rules that govern interaction among communicating entities. Precise expression of these rules is the goal of protocol specification. It is often difficult to capture the essence of this interaction without going into too much detail and infringing on the territory of the implementor. Nevertheless, as argued previously, a certain amount of detail is necessary. In order to avoid the excessive detail introduced by procedural forms of specification, we make use of a state transition approach. The resulting method, termed *abstract behavioral specification*, can fulfill the needs of precise definition and analysis, while still serving as a basis for implementation.

Consider a sequential process of the kind described in Section 2.2.1. It is possible to give specifications for such a process by regarding its behavior as being *event driven*. By this we mean that the life of a process consists of a continuous series of event processing cycles. Each cycle begins with the occurrence of an event,

followed by its processing, after which the process goes back to wait for the next event. Two kinds of events are recognized: the reception of a message from one of the input buffers and an internally generated timeout event. The quiescent state for a process is to be waiting for a message from one or more of its input buffers.

The event driven view of process behavior leads to the adoption of a state transition paradigm for the specification of a protocol module. Local state information is assumed to be maintained by the process; this is collectively referred to as the *state vector*. Processing of an event causes a transition to occur, which updates the state vector. Similar models of protocol systems are in common use today. However, there are some important distinguishing features of our model.

First of all, the state vector is strictly local to a single process; there is no notion of a global state vector. This preserves the modular character of the specification and proof methods. Since a process communicates with its external environment through explicit output buffers, we must account for this interface in the behavioral specification. Accordingly, the actions of event processing are divided into two categories.

1. *Response*. Messages may be sent to the output buffers; this constitutes a modification of the external environment.
2. *Transition*. The state vector is updated; this constitutes a modification of the internal environment.

The exact mechanism used to state a detailed specification, which is based on the use of decision tables, is the other major distinguishing feature of our method. Before introducing it, we must elaborate on the overall structure of the sequential process model.

```
process sender (inputs source: message;
                       ack_in: natural;
                output pkt_out: packet) =
begin
   state vector (unack, next: natural;
                 queue: mapping of packet;
                 timing: boolean;
                 to_time: natural)
      initially (0, 0, null, false, 0);
   events
      next - unack < send_window =>
         on receipt of mess from source
            handle by source_hdlr;
      true =>
         on receipt of ack from ack_in
            handle by ack_hdlr;
      timing =>
         after to_time handle by timeout_hdlr;
   end
end
```

Figure 2-5: Example of sequential process definition.

2.3.1 Sequential process definitions

Fig. 2-5 contains a sample process definition, the sender process of the Stenning protocol. This represents a fairly abstract process definition, having a form that may be regarded as a process schema. There are three major parts to a sequential process definition.

1. The process header identifies the input and output buffers and the types of objects that flow through them. This is completely analogous to the concurrent process case.

2. The nature of the state vector is revealed by declaring it as a record-like abstract data object. Its components are named and typed using the abstract types provided in the specification language. Its initial value is also declared.

3. A list of event processing statements is provided. Each statement corresponds to a particular class of event, either a message reception or a timeout. Each statement refers to an *event handler*, which is a separate language structure refined at the next lower level.

Thus the process schema contains declarations for both data and control. The list of event processing statements may be thought of as a CSP-like guarded command structure with enabling conditions [Hoare 78], embedded within a nonterminating loop. Each statement begins with an enabling condition that determines whether the given event is eligible for selection on a given cycle. In this way, it is possible to select a subset of the input buffers upon which to wait for the next event. Messages arriving to a buffer with a corresponding false enabling condition will be queued but not yet received. Similarly, a timeout event with a false enabling condition means that the timer is currently turned off for that event. Events that are enabled will be waited upon in parallel. This construct is also similar to an Ada "select" statement, where the guards correspond to when-conditions and the receive and timeout statements correspond to accept and delay alternatives.

```
process P (inputs  IB1: IT1; ...; IBm: ITm;
           outputs OB1: OT1; ...; OBn: OTn) =
  begin
     state vector (SV1: ST1; ...; SVk: STk)
        initially (IV1, ..., IVk);
     events
        A1 => on receipt of V1 from IB1 handle by RH1;
        . . .
        Am => on receipt of Vm from IBm handle by RHm;
        B1 => after T1 handle by TH1;
        . . .
        Bq => after Tq handle by THq;
     end
  end
```

Figure 2-6: General form of sequential process definition.

The general form of a sequential process definition is shown in Fig. 2-6. There is exactly one event processing statement for each input buffer. On the other hand, there can be any number of timeout events. Each timeout event has an expression T_i that refers to the absolute time after which it can fire. For definiteness, we assume time is in units of milliseconds. The boolean expressions $A_1,...,A_m$ and $B_1,...,B_q$ are used to select a subset of events eligible for execution on any given event processing cycle. State vector components may be freely referenced in such guard expressions. Receive event statements contain implicit declarations for the variables V_i, assumed to represent objects of type IT_i. These variables are only accessible in the corresponding receive event handlers, RH_i. In addition, all state vector components are accessible to all event handlers. Observe that if all enabling conditions should simultaneously become false, the process will have effectively terminated, in the sense that no further events will ever be handled.

2.3.2 Event handlers

Next we describe how the details of event handling are specified. Each handler will in fact be represented by a special form of decision table. The decision table will indicate the appropriate response and transition actions to be performed under various conditions pertinent to the given event. This technique offers a way to state the specification using a precise and concise notation, while still allowing it to be done in a functional, nonprocedural manner. It also provides the advantages of structuring the specification in such a way that a limited analysis for consistency and completeness may be performed, as well as yielding a very simple

procedure for generating verification conditions.

The task of an event handler is to specify the two kinds of actions mentioned earlier: transition and response. These actions should be functions only of the current state and the value of the message just received (for receive events). We would like to specify these actions by stating only their final effects, not how they are achieved. Therefore we reject any solution based on the use of procedural programming statements such as assignments and loops. Preferable is a solution that is functional in nature, for example, something akin to the output and next state functions of classical finite state automata.

Naturally, for updating the state vector a straightforward next state function is the obvious choice. To simplify matters, we actually separate it into k individual functions, one for each state vector component. Thus, we have for receive events

$$NS_{ij}(SV_1, \ldots, SV_k, V_j) = expression_{ij};\ i=1,\ldots,k;\ j=1,\ldots,m$$

and for timeout events

$$NS_{ij}(SV_1, \ldots, SV_k) = expression_{ij};\ i=1,\ldots,k;\ j=1,\ldots,q.$$

A designer would simply need to provide an appropriate set of expressions to specify the state transitions of his protocol.

Specifying response actions, the analog of the output function for an FSA, is less clear but could be handled in a similar way. The problem is that there is no single output value. There is, in fact, multiplicity in two dimensions: multiple output buffers and multiple messages sent over each. One possibility is to have a single function whose value is the sequence of all messages to be sent out. Quite likely, though, we do not need to specify the relative order of sends between two output buffers. Moreover, this might err a bit on the side of overspecification. Consequently, we content ourselves with independent output functions for the output buffers. This allows us to specify the sequence of messages that go out on a given output, knowing that these send operations may be interleaved with those of the other outputs. Thus, we have as response functions for receive events

$$R_{ij}(SV_1, \ldots, SV_k, V_j) = expression_{ij};\ i=1,\ldots,n;\ j=1,\ldots,m$$

and for timeout events

$$R_{ij}(SV_1, \ldots, SV_k) = expression_{ij};\ i=1,\ldots,n;\ j=1,\ldots,q.$$

While these functions theoretically suffice to define behavior, from a methodological point of view they are deficient. In practice, these functions can become numerous. Furthermore, they will tend to contain complex expressions, such as

$NS_{ij}(SV_1, \ldots, SV_k, V_j) =$

```
if P
  then if Q
         then if R then A else B fi
         else C
       fi
  else if S then D else E fi
fi
```

In the midst of even a modest number of such functions a designer might lose sight of relationships among them and how they interact to achieve the total effect. What is needed is a way to impose more structure on the specification in order to make these orthogonal relationships more apparent.

Decision table notation satisfies this need quite well. It permits us to identify those actions that are performed under identical conditions. Ultimately, it forces the designer to point out all conditions under which he believes processing should occur. Mechanical analysis of the tables can then reveal cases that may have

been overlooked. In addition, decision tables are about as compact a notation for expressing complex logic as one can imagine. There should be little doubt that the complexity of real life protocols warrants this kind of brevity.

	Stub	Rules			
Conditions	x < y	T	F	F	F
	n + 5 = i * j	-	T	F	F
	in_limits(a,b,c)	-	-	F	T
Actions	x := y	X	-	X	X
	reset (s, t)	-	X	X	-
	n := next(i,j)	-	X	-	X

Figure 2-7: Example of limited entry decision table.

Before going on to describe our particular formalism, we review some basic concepts of decision tables for the reader who may be unfamiliar with them [Metzner 77, Pooch 74]. In their most common application, decision tables were used to express conditional processing for procedural programming languages. The limited entry format was used most often as the basic syntactic form. An example is shown in Fig. 2-7.

The rules (columns) of such a table represent separate alternatives for processing. One rule is selected for execution based on the evaluation of certain conditions. For a rule to be selected, its truth value assignment must include the boolean vector that results upon condition evaluation. When a rule is selected, the actions marked with an X are performed in sequence. At most one rule is selected; it is possible that none is selected, in which case no actions are performed.

It is easy to adapt this kind of structure for use with our transition and response functions. We simply change the form of action entries to represent expression evaluation rather than the mere performance of imperative statements. The required actions are of two types:

1. Sending a sequence of messages to a particular output buffer.
2. Updating the value of a state vector component.

	Stub	Rules
Condition part	Boolean expressions	T, F, or don’t care
Action part	Output buffers	Response
	State vector components	Transition

Figure 2-8: General form of event handler decision table.

Fig. 2-8 shows the general form we will use for our specialized decision tables. In practice, we cannot

physically place an expression in the small space for an entry so we use single letter abbreviations that are defined immediately below the table. This form is shown in Fig. 2-9.

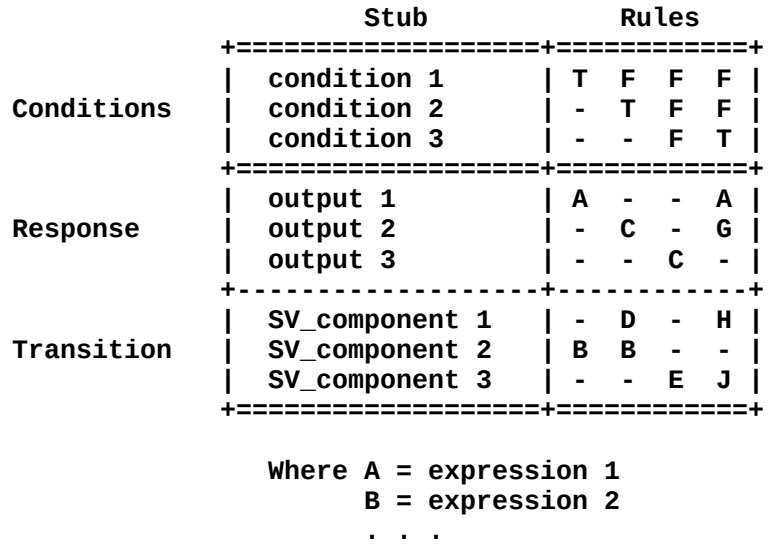


Figure 2-9: Extended entry decision table format.

The semantics of event handling can now be explained as follows.

- Based on the outcomes of condition evaluation, one rule is selected for execution. If more than one are eligible, one is chosen nondeterministically. If none is eligible no actions are performed; this is referred to as an implicit ELSE rule. Decision table rules may be thought of as guarded commands [Dijkstra 75].
- For each response row in the table, the entry for the selected rule must evaluate to a sequence of objects of the type associated with the corresponding output buffer. This sequence of messages is sent to the designated buffer. A null entry denotes that no messages are sent.
- For each transition row, the entry for the selected rule must evaluate to an object of the type associated with the corresponding state vector component. This is the new value to be assumed by that component. A null entry indicates that the state vector component does not change value.
- There is no order of execution implied by the position of rows in the table. All expressions in the entries refer to the old values of the state vector elements.

Note that for response actions there may not be enough space in the output buffers to hold all messages being sent. In this case, we assume that the maximum number of messages is sent to each buffer and the process then becomes blocked awaiting the opportunity to discharge the rest of its messages. An example of one of our protocol decision tables for the receiver process is shown in Fig. 2-10.

One more feature is included in the sequential process model. In order to compute values of time needed for initiating timeouts, a process must be able to determine the current time. This is accomplished by having an implicitly declared variable "time" that is set to the current time when an event occurs. It may be referred to by any of the expressions appearing in the event handlers. It may not be referenced by any event selection guards or any assertions.

```
pkt_hdlr                          1 2 3 4
     +================================+=========+
     | pkt.seqno = next               | F - T T |
     | pkt.seqno > next               | F T - - |
     | pkt.seqno - next < rcv_window  | - T - - |
     | pkt.seqno in domain(queue)     | - F - - |
     | (next+1) in domain(queue)      | - - F T |
     +================================+=========+
     | sink                           | - - B D |
     | ack_out                        | A - C E |
     +--------------------------------+---------+
     | next                           | - - H J |
     | queue                          | - G - K |
     +================================+=========+

     Where
       A = [next]
       B = [pkt.mssg]
       C = [next+1]
       D =    pkt.mssg
           :> apply (".mssg", range (consec (queue)))
       E = [reach(queue)]
       G = queue with ([pkt.seqno] := pkt)
       H = next + 1
       J = reach(queue)
       K = upper(queue, reach(queue))
```

Figure 2-10: Example of protocol decision table.

2.4 Service Specifications

Up to this point, we have not introduced any concepts explicitly for the purpose of doing verification. In fact, the techniques described thus far would be useful for protocol specification regardless of whether we intended to carry out any verification. Now we would like to extend our formal machinery to be able to prove properties about the protocols we model. The main property of interest is some formal statement of the services provided by a protocol.

Our method for expressing a service specification is based on the use of assertions about the input/output characteristics of a concurrent process. Generally speaking, what we try to do is relate message flows at the process outputs to those at the process inputs. Message delivery properties, which are our main concern, are conveniently expressed in this manner. However, service primitives that do not result in any output messages being generated, in other words whose sole purpose is to cause some internal state change, have effects that are more difficult to capture. Connection management commands often behave in this way. Nevertheless, they must exert an influence that will be felt at some future point in the life of the process, and therefore, will be reflected in the future messages that flow through the outputs. To deal with the full generality of such behavior, we will need some extensions that are beyond the basic methods described in the following sections. These extensions are taken up in Chapter 7.

2.4.1 Histories

The primary device used for expressing this kind of service specification is known as a *history*. This is an instance of the more general class of *auxiliary variables*, a well established tool for the verification of concurrent programs. Our notion of history is one commonly used in parallel process models [Howard 76, Good 79]. A message history (or buffer history) records all of the messages received from an input buffer or those sent to an output buffer during the life of a process.

The history itself is merely a sequence of data objects, which are of the same type as the messages flowing through the given buffer. For instance, if we have a buffer with element type "msg," then its corresponding input and output histories have the structure

type msg_history = sequence of msg

Note that a history is a finite but unbounded data object.

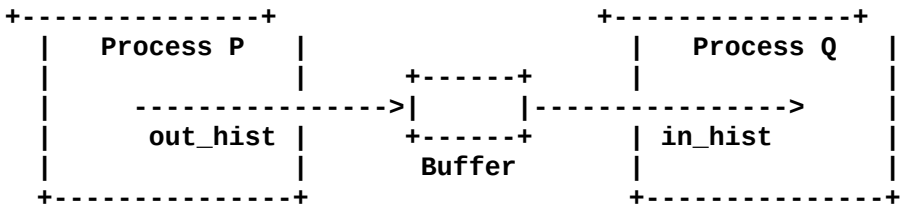


Figure 2-11: History variables.

As auxiliary variables, histories are assumed to reside within sequential processes. Because of the restrictions on buffer access, only one sequential process can have a buffer as its input or output. This is a simplification of the situation in Gypsy, where multiple processes can have access to a single buffer. It means we have no need for the distinction between local and global histories that Gypsy has [Good 79]. Instead, a single auxiliary variable for each end of a buffer contains all the pertinent information (Fig. 2-11). When we say a history is an output history, we mean output with respect to the sending process. From the buffer's point of view, the designations input and output will appear to be reversed. This is just an artifact of the duality inherent in our bipartite process and buffer model.

When referring to history variables, we give them the same name as the corresponding formal parameters of the containing process. This suffices to unambiguously name the histories of the external buffers of a process. However, when the context includes more than one process, naming conflicts may occur. This possibility exists whenever we discuss the decomposition of a concurrent process. As a result, we will establish some simple conventions to deal with this problem.

Fig. 2-12 illustrates the naming conventions for histories in the context of a concurrent process P having subprocesses Q and R. These conventions and the history variable access rules are outlined below.

- For buffers external to the concurrent process P, we use their associated buffer names (formal parameters of P) as history names. No ambiguity is possible so there is no need to qualify these names any further. Note that these buffers are external to the subprocesses as well, in which case we use the same history names because they are in fact the same history variables. It is understood that histories reside within sequential processes; only through the parameter passing mechanism are they made visible to higher levels in the process hierarchy. For example, in Fig. 2-12, "source" is a history variable accessible to both P and its subprocess Q.
- For buffers that are strictly internal to a concurrent process, like A and B in the figure, we use history names that are derived from the internal buffer names. In particular, the history names are "B_in" and "B_out" for a buffer named B. These buffer names will generally differ from the formal parameter names of the subprocesses. Access to these history variables does not rise any higher in the process hierarchy, that is, they are not visible above the concurrent process P.

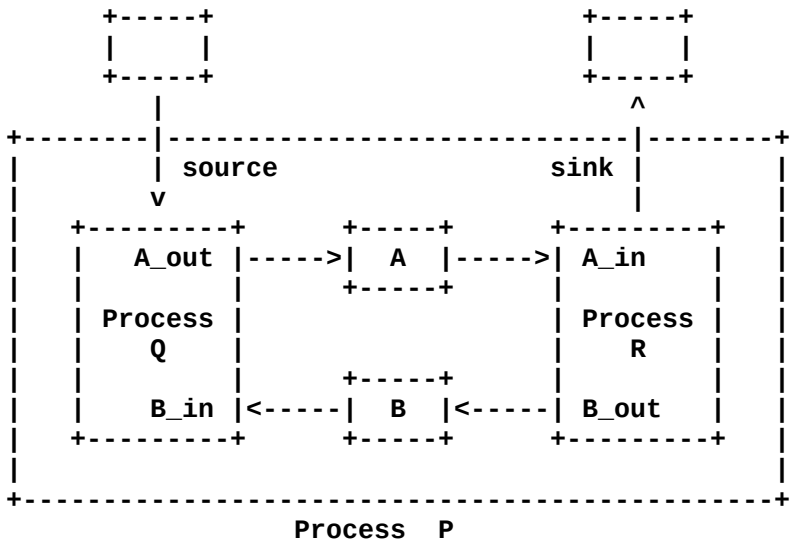


Figure 2-12: History naming conventions.

The actual manipulation of history variables, which is implicit in the proof rules, follows conventional usage. Each history is initially null. Upon receipt of a message M from an input buffer, its history is updated by

in_hist := in_hist <: M

where "<:" is the operator for appending an element to the right end of a sequence. For the output case, we will be sending out entire sequences of messages in a single action. Hence output histories are updated by

out_hist := out_hist @ S

where "@" is the operator for concatenating sequences.

2.4.2 External invariants

Histories are the variables used for expressing assertions about processes. To prove safety properties, we will want to express such assertions as invariants. In addition, we want to extend the modular nature of our process model to its verification techniques. For this reason, we make use of *external invariants* in our methods. An external invariant may only access the history variables of buffers external to the process in question. The access rules laid down in the previous section will help ensure that this is the case.

Invariant assertions on histories typically state relations that must hold between pairs of histories or pairs of expressions involving histories. A simple example would be

sink initial source

where "initial" is an infix relational operator meaning initial subsequence. A more complicated invariant would consist of a conjunction of such formulas.

We must define the meaning of the external invariant a bit more carefully. Specifically, we must identify those points in time at which the assertion must hold. Consider, for the moment, a sequential process. Intuitively, we want an invariant to hold when the process is *idle*, waiting for events to occur. Verification will demonstrate that the invariant is maintained after an event occurs and is handled by the sequential process. This assertion is not required to hold while an event is being processed. In this regard it is analogous to the loop invariant of the inductive assertion method for sequential program verification. We can think of the invariant as being an instantaneous assertion that holds at the waiting point of the event processing loop.

External invariants are also stated for concurrent processes. In order to achieve proof modularity, external invariants have the same form for all processes, that is, histories are the only variables allowed. It is also necessary, though, to define when an external invariant holds for a concurrent process. To do this we simply extend the notion of idleness. Recall that a sequential process is idle when it is not processing events. A concurrent process is deemed to be idle when all of its subprocesses are simultaneously idle, which amounts to a recursive definition of the concept. Now we can simply say that for any process its external invariant holds when the process is idle.

A major advantage of this form of assertion is that it holds at fewer points in time and needs to cover fewer intermediate states. It is therefore possible to write stronger assertions because they apply at more specialized points in time. Our concept of invariant differs from the Gypsy concept of blockage assertion, which must hold when a process is waiting either to send or to receive. Similarly, it differs from other kinds of invariants that must hold after every separate I/O operation.

Chapter 3
VERIFICATION METHODS

Given the process model developed in Chapter 2, it is possible to develop complementary verification methods of great simplicity and power. Of prime importance is the modularity of these verification techniques. Modularity is the key to achieving proofs that do not grow unmanageably with the size of a problem. Modular verification of concurrent processes is an important feature of the Gypsy methodology [Good 79]. We embrace this aspect of the Gypsy paradigm in the work that follows.

3.1 Outline of Methods

All processes in our model, both sequential and concurrent, are given external invariants of the type described in Section 2.4.2. These are the assertions used in the verification of hierarchical process structures. Naturally, the methods for proving that an invariant holds are different for the two kinds of processes. We will begin by discussing the verification method for sequential processes.

Principles of sequential program verification are well understood. The Floyd-Hoare method, which is most prevalent, makes use of an axiom system to characterize the semantics of program statements [Floyd 67, Hoare 69]. A program is then annotated with assertions at various points, with the meaning that the assertions hold when control passes through those points. From this annotation a set of *verification conditions* (VCs) is constructed; these are logical formulas whose truth must be established before the program can be said to be verified. The use of mathematical induction to prove invariant assertions about loops is implicit in this method.

Our task is eased considerably by the way we model sequential processes. By working with a process schema, which essentially has a fixed structure, we can develop a proof method with a parallel fixed structure. The result is a verification schema, which makes use of a fixed set of assertions and a fixed set of verification conditions. A designer need only supply the parameters for this schema and then carry out the actual proofs. Given this method of operation, the human verifier is not really concerned with proof rules or axioms since we do not have combinative statements in the conventional programming language sense. Nevertheless, we do state proof rules in Section 3.2 to formally define our method. A typical user, however, will simply use a verification condition generator that has been constructed to follow the schema presented in Section 3.3.

Ultimately, we wish to prove that the external invariant of each process holds. In order to do this for a sequential process, we introduce an intermediate assertion, the *internal invariant*. It differs from the external invariant only in the data objects that it may reference. An internal invariant may refer to the state vector components of a sequential process as well as to its buffer histories. Otherwise, it is required to hold at precisely the same points in time as the external invariant. This assertion may be regarded as the analog of the loop invariant in the inductive assertion method for sequential program verification.

The basic strategy for verifying a sequential process is depicted in Fig. 3-1. The internal invariant acts as a bridge between the behavioral specification (process definition) and the external invariant. We first show that the internal invariant holds for the process definition of interest. We then prove that the internal invariant implies the external invariant. It is the external invariant that goes on to participate in the proofs of the

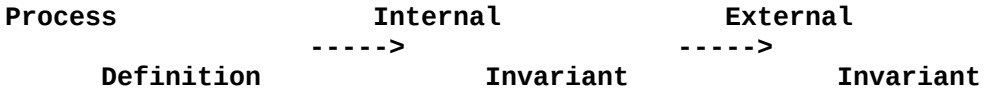


Figure 3-1: Structure of sequential process proof.

concurrent process at the next higher level. Recall that the external invariant may only refer to history variables, so the variables of this assertion are a subset of those of the internal invariant.

Verification of a concurrent process has the same goal, namely establishing that its external invariant holds. Naturally, the internal structure of such a process must play a role in the proof. However, we only examine the topmost level of the hierarchy, ignoring the internal details of the subprocesses. The constraints on external invariants make this possible. The only data objects that may be referenced by such an assertion are the histories of the input/output buffers of the process in question. No knowledge of the internal structure of a process is used.

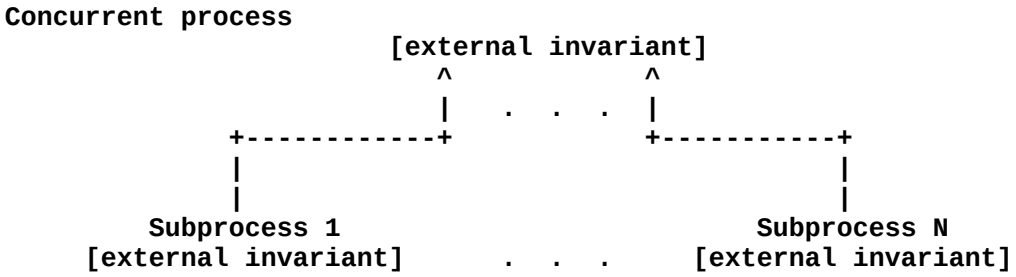


Figure 3-2: Proof structure for concurrent process.

This is the technique that enables a modular proof structure to be obtained. When verifying a concurrent process, we only make use of the external invariants of its subprocesses. The conjunction of these invariants, plus knowledge about the topology of process interconnection through message buffers, must together imply the external invariant of the parent process. One level of the concurrent process hierarchy is thereby verified by this procedure. The overall proof structure is depicted in Fig. 3-2.

3.2 The Deductive System

Having described the verification methods in brief, we will proceed with a formal development of the deductive system. Our basic technique for formalizing verification is based on Hoare style axiomatization [Hoare 69]. In this approach, a logic of verification is defined by postulating a set of axioms and inference rules about program structures. These axioms and rules capture the semantics of various programming primitives in terms of their effects on assertions. This allows one to reason about programs by transforming a verification problem into a set of logical formulas involving assertions over the program variables.

The technique just described is normally used for verifying programs of a traditional sequential programming language. Our situation differs from that in two respects: use of a concurrent process model and use of a state transition model. Nevertheless, we can still develop a similar formalism based on a system of inference rules. In fact, we benefit from the abstract, fixed structure of our process model. As a result of not casting our problem in terms of conventional programs, we can exploit the more specialized nature of the process model. The entire deductive system for verifying both sequential and concurrent processes consists of just three inference rules.

Before introducing the proof rules, we need to lay down a few preliminary definitions. They should establish a precise working vocabulary for the rest of this chapter. We begin by restating some concepts that

have already been discussed less formally.

Definition 3.1: An *assertion* for a process P is a logical formula involving the variables of P and no others. These may be either real variables or auxiliary variables.

Definition 3.2: A sequential process is *idle* when it is not in the midst of processing an event. A process that is temporarily blocked on a send operation is not considered idle. A concurrent process is idle when all of its subprocesses are simultaneously idle.

Definition 3.3: An *internal invariant* for a sequential process P is an assertion that may refer to the state vector components of P or the histories of P's buffer parameters. It only has access to the input history of an input buffer or the output history of an output buffer. It must hold whenever P is idle.

Definition 3.4: An *external invariant* for any process P is an assertion that may refer only to histories of P's buffer parameters. Again, it only has access to the input history or output history, as appropriate. It must hold when P is idle.

In order to express that an assertion Q is an internal or external invariant of a process P, we use a pair of boolean functions:

int(P, Q)
ext(P, Q)

It is also useful to have available the notion of *predicate transformer* [Dijkstra 75]. This is a function that maps one predicate or assertion into another. We will have use for Dijkstra's weakest precondition function.

Definition 3.5: The *weakest precondition* of a statement (or action) S for an assertion Q is the weakest predicate that must be true before the execution of S in order to ensure that Q is true upon termination of S. Notationally, it is expressed as wp(S, Q).

Definition 3.6: It is often necessary to refer to a *substitution instance* of a formula or term. The expression

$$Q \text{ with } (v_1 := e_1; \ldots; v_n := e_n)$$

denotes the formula or term obtained from Q by substituting expression e_i for variable v_i, for i=1,...,n.

3.2.1 Sequential process proof rules

Verification of a sequential process involves showing that its external invariant holds when the process is idle. This proof revolves around the internal invariant, consisting of two subproofs:

1. Show that the internal invariant holds whenever the process is idle.
2. Show that the internal invariant implies the external invariant.

We will begin with the second rule since it is quite simple. Assume P is a sequential process.

```
int(P, I);   I -> E
                    --------------------                    (R1)
                         ext(P, E)
```

We are using a common notation for proof rules. Its meaning is if each conjunct above the line is true then the predicate below the line is true. The validity of this rule is immediate since both I and E are required to hold at the same points of P's execution. Of course, E must not violate the access rules for external invariants.

The other rule is not so simple. It must account for the details of event processing to show that invariance is maintained. For the moment, we will abstract away from the details of event handlers by resorting to the weakest precondition function. Consider the schematic form for a sequential process shown in Fig. 3-3. Here we have represented the actions of event processing by the identifiers RS, for receive statement, and TS, for timeout statement.

```
process P (inputs  IB1: IT1; ...; IBm: ITm;
           outputs OB1: OT1; ...; OBn: OTn) =
             begin
                state vector (SV1: ST1; ...; SVk: STk)
                   initially (IV1, ..., IVk);
                events
                   A1 => RS1;
                   . . .
                   Am => RSm;
                   B1 => TS1;
                   . . .
                   Bq => TSq;
                end
             end
```

Figure 3-3: Schematic form for a sequential process.

Given this characterization of the process, we can state the second proof rule as follows.

$$\frac{\begin{array}{l} I \text{ with } (IH_1 := null;...; IH_m := null; \\ \qquad OH_1 := null;...; OH_n := null; \\ \qquad SV_1 := IV_1;...; SV_k := IV_k); \\ I \;\&\; A_1 \rightarrow wp(RS_1, I);\ ...;\ I \;\&\; A_m \rightarrow wp(RS_m, I); \\ I \;\&\; B_1 \rightarrow wp(TS_1, I);\ ...;\ I \;\&\; B_q \rightarrow wp(TS_q, I) \end{array}}{int(P, I)} \qquad \textbf{(R2)}$$

This rule requires that we show I to hold on the initial state and that its invariance is maintained after execution of whichever statement is selected to process an event.

Now we may decompose the RS and TS statements further to derive their weakest precondition functions. Each of these statements consists of a receive or timeout event followed by the handling of the event according to an appropriate decision table. Let the event handler be denoted by EH. Since a timeout event itself does not change any variables of a process, it follows that

$$wp(TS_i, Q) = wp(EH_i, Q)$$

On the other hand, a receive event will cause one of the input histories to be updated:

$$wp(RS_i, Q) = wp(EH_i, Q) \text{ with } (IH_i := IH_i <: V_i)$$

Next we must decompose the event handler construction. An event handler can be regarded as nothing more than a guarded command list, with its conditions for selection being the guards and the actions specified by the lower half of the table constituting the commands. The only real difference is that in Dijkstra's original formulation of the guarded command list, at least one guard must be true or program abortion occurs. In our case, the decision table has an implicit ELSE rule so that we conceptually have an additional guarded command whose guard is the negation of all the others and whose command is the null action.

Using the transformation outlined above, let us represent an event handler EH by the following guarded command structure.

$$[b_1 \Rightarrow AC_1;\ ...;\ b_r \Rightarrow AC_r;\ (\text{not } b_1 \;\&\; ... \;\&\; \text{not } b_r) \Rightarrow skip]$$

where r is the number of decision table rules, b_i denotes the conjunction of conditions required for rule i to be selected, and AC_i represents the aggregated actions of rule i. Under this formulation, we define the weakest precondition for EH as

$wp(EH, Q) = \quad (b_1 \rightarrow wp(AC_1, Q))$

& . . .
& (b_r -> wp(AC_r, Q))
& (not b_1 & ... & not b_r -> Q) (1)

All that remains now is to define the weakest preconditions of the AC_i in 1. The two types of actions are sending messages to the output buffers and updating state vector elements. Sending messages causes output histories to be updated. Therefore, the effects of the aggregated actions AC for a rule are given by

```
wp(AC, Q) = Q with (OH1 := OH1 @ RSP1; ...;
                    OHn := OHn @ RSPn;
                    SV1 := TR1; ...;
                    SVk := TRk)
```

where the $\{RSP_i\}$ are response expressions from the proper decision table rule and the $\{TR_i\}$ are transition expressions.

This completes our discussion of proof rules for sequential processes. Later we will use these rules to generate a schematic set of verification conditions.

3.2.2 Concurrent process proof rules

Verification of a concurrent process is performed in modular fashion, following the hierarchical structure of the concurrent process itself. Proving that the external invariant holds requires only the external invariants of the subprocesses; no knowledge of the internal structure of these subprocesses is used [Good 79]. The required proof is the result of a single inference rule.

The gist of this rule is that the conjunction of the subprocess invariants must imply the invariant of their parent process in order for us to conclude that the upper invariant holds. There is, however, one detail that must be dealt with before stating the rule. Because the subprocess invariants are stated independently, there will be no obvious correlation among history names at the next higher level, where this proof is to take place. The use of formal process parameters (external buffers) forces us to consider the binding of actual to formal buffer parameters. For this reason, we must impose a consistent renaming of history variables in the subprocess invariants.

Let H be a history variable in the external invariant of any subprocess of P. H will be renamed as follows.

- If H corresponds to one of P's external buffers, it is given the same name as P's formal parameter.
- If H corresponds to the sending end of an internal buffer B, it is renamed B_out.
- If H corresponds to the receiving end of an internal buffer B, it is renamed B_in.

Denote by rename(P, Q) the assertion that results from renaming the external invariant Q for the environment of parent process P.

Now we can precisely state the final proof rule. Let P_1,...,P_n be the subprocesses of P.

```
        ext(Pi, Ei), i=1,...,n;
  { AND_{i=1}^{n} rename(P, Ei)
     & AND_{i=1}^{k} Bi_in initial Bi_out } -> E
 ------------------------------------------------        (R3)
                    ext(P, E)
```

The renaming of history variables ensures that all assertions range over an appropriate set of variables so the indicated implication can be proved. During this proof relationships between histories are provided by terms of the form

B_in initial B_out

These terms state that the sequence of messages received from a buffer is an initial segment of those sent to it. We will informally argue the validity of this verification method at the end of this chapter.

3.3 Verification Conditions

Armed with a set of proof rules, we can now make the procedure for verifying processes more explicit. In particular, we can provide a generic set of verification conditions, the proofs of which suffice to establish the external invariants for generic processes. It is then a simple matter to fill in the parameters of this VC schema to obtain the actual VCs for a specific process. Proofs of the specific VC instances will constitute verification of the specific process. The notation to be presented is rather cumbersome for direct human use. It is intended to serve as the specification from which an automatic verification condition generator might be built.

3.3.1 Sequential process VCs

Two proof rules are provided for verifying a sequential process. Rule R1 is simple, but rule R2 contains some complex antecedents. It would be better to split up all the required implications to make the overall proof process more manageable. We would then have a number of separate proofs to perform before we could say that the internal invariant holds.

The antecedents of rule R2 can be split and normalized by expanding the wp functions and applying two logical identities for implications:

```
P -> (Q & R)   eqv  (P -> Q) & (P -> R)
               P -> (Q -> R)  eqv  (P & Q) -> R
```

This gives rise to two basic kinds of verification conditions.

1. The internal invariant must be shown to hold with the initial value of the state vector and null buffer histories.
2. It must be shown that the internal invariant is maintained after an event is processed. This involves proving that for each event class, and for each decision table rule within that class, the actions performed will preserve the invariance of the assertion. This must also be shown for the ELSE rules of the decision tables.

In order to present the form of verification conditions, we need to introduce some appropriate notation. We assume the general form for a sequential process shown in Fig. 2-6 on page 9. The notation used is as follows.

$IB_1,...,IB_m$	input buffers (and histories)
$OB_1,...,OB_n$	output buffers (and histories)
$SV_1,...,SV_k$	state vector elements
$IV_1,...,IV_k$	initial state vector values
$V_1,...,V_m$	receive variables
$A_1,...,A_m$	enabling guard for receive events
$B_1,...,B_q$	enabling guard for timeout events

Recall that we are using buffer names to stand for their corresponding histories.

Additionally, it is necessary to refer to various quantities and expressions derived from entries in the event handler decision tables, as depicted in Fig. 2-9 on page 12. For this we use:

```
nrr(e)                    no. of rules in table e (receive)
nrt(e)                    no. of rules in table e (timeout)
ncr(e)                    no. of condition rows in
                            table e (receive)
nct(e)                    no. of condition rows in
                            table e (timeout)
```

$RC_{e1},...,RC_{e,ncr(e)}$	condition expressions in stub of table e (receive)
$TC_{e1},...,TC_{e,nct(e)}$	condition expressions in stub of table e (timeout)
RDT(e,s,i,j)	entry for receive event decision table e, section s, row i, rule j
TDT(e,s,i,j)	entry for timeout event decision table e, section s, row i, rule j
CD_{ei}(xDT(e,C,i,j))	derived condition expression for event e, row i, rule j
RS(xDT(e,R,i,j))	derived response expression for event e, row i, rule j
TR_i(xDT(e,T,i,j))	derived transition expression for event e, row i, rule j

The last three expression classes are defined as macro-like objects that are expanded when actual VCs are constructed.

CD_{ei}(entry) = if entry = 'T' then RC_{ei}
else if entry = 'F' then not RC_{ei}
else true
fi fi

RS(entry) = if entry = '-' then null else value(entry) fi

TR_i(entry) = if entry = '-' then SV_i else value(entry) fi

Here value(entry) refers to the expression that is the value of its abbreviated entry appearing in the table. Recall that abbreviations are defined immediately beneath a table.

Finally, invariant assertions themselves are represented by boolean functions with predetermined arguments. For a process named P, its internal and external invariants are named P_int and P_ext.

P_int ($IB_1,...,IB_m,OB_1,...,OB_n,SV_1,...,SV_k$)

P_ext ($IB_1,...,IB_m,OB_1,...,OB_n$)

Using the notation defined above, Table 3-1 displays the general form of verification conditions that must be proved to establish the external invariant of a sequential process. In accordance with our view of a sequential process as a process schema, we may similarly regard Table 3-1 as a verification condition schema. Only the parameters need be filled in; the structure remains the same from one problem to the next.

The purpose of the verification conditions in Table 3-1 is explained below.

a. This VC establishes that the internal invariant holds initially.

b. For each receive event and for each rule in its corresponding decision table, we must show that the event processing specified by that rule preserves the invariant. The hypotheses contain the conditions under which a rule is selected and the conclusion reflects the effects of event processing according to the rule.

c. For each receive event, we must show the invariant is preserved when no rule is selected, or equivalently, when the ELSE rule is selected. The conclusion shows that the only change is due to the message reception.

d. For each timeout event and for each rule in its corresponding decision table, the processing according to that rule must preserve the invariant. This case is similar to (b) except that there is

a) $P\_int\ (null, \ldots, null, null, \ldots, null, IV_1, \ldots, IV_k)$

b)
$$P\_int\ (IB_1,\ldots,IB_m,OB_1,\ldots,OB_n,SV_1,\ldots,SV_k)$$
$$\& A_i\ \&\ CD_{i1}(RDT(i,C,1,j))\&\ldots$$
$$\&CD_{i,ncr(i)}(RDT(i,C,ncr(i),j))$$
$$\rightarrow P\_int\ (IB_1,\ldots,IB_i{<:}V_i,\ldots,IB_m,$$
$$OB_1@RS(RDT(i,R,1,j)),\ldots,OB_n@RS(RDT(i,R,n,j)),$$
$$TR_1(RDT(i,T,1,j)),\ \ldots,\ TR_k(RDT(i,T,k,j)))$$
$$\text{for } i=1,\ldots,m;\ j=1,\ldots,nrr(i)$$

c)
$$P\_int\ (IB_1,\ldots,IB_m,OB_1,\ldots,OB_n,SV_1,\ldots,SV_k)$$
$$\& A_i$$
$$\&\ not\ (CD_{i1}(RDT(i,C,1,1))\&\ldots$$
$$\&CD_{i,ncr(i)}(RDT(i,C,ncr(i),1)))$$
$$\&\ .\ .\ .$$
$$\&\ not\ (CD_{i,1}(RDT(i,C,1,nrr(i))\ \&\ \ldots$$
$$\&\ CD_{i,ncr(i)}(RDT(i,C,ncr(i),nrr(i))))$$
$$\rightarrow P\_int\ (IB_1,\ldots,IB_i{<:}V_i,\ldots,IB_m,$$
$$OB_1,\ldots,OB_n,SV_1,\ldots,SV_k)\qquad \text{for } i=1,\ldots,m$$

d)
$$P\_int\ (IB_1,\ldots,IB_m,OB_1,\ldots,OB_n,SV_1,\ldots,SV_k)$$
$$\& B_i\ \&\ CD_{i1}(TDT(i,C,1,j))\&\ldots$$
$$\&CD_{i,nct(i)}(TDT(i,C,nct(i),j))$$
$$\rightarrow P\_int\ (IB_1,\ldots,IB_m,$$
$$OB_1@RS(TDT(i,R,1,j)),\ldots,OB_n@RS(TDT(i,R,n,j)),$$
$$TR_1(TDT(i,T,1,j)),\ \ldots,\ TR_k(TDT(i,T,k,j)))$$
$$\text{for } i=1,\ldots,q;\ j=1,\ldots,nrt(i)$$

e)
$$P\_int\ (IB_1,\ldots,IB_m,OB_1,\ldots,OB_n,SV_1,\ldots,SV_k)$$
$$\rightarrow P\_ext\ (IB_1,\ldots,IB_m,OB_1,\ldots,OB_n)$$

Table 3-1: Verification conditions for a sequential process.

no message reception. Because of this, there is no need to worry about the ELSE rules since they will have no effect whatsoever on the variables.

e. The last VC demonstrates that the internal invariant implies the external invariant (R1).

As an example of VC construction, Fig. 3-4 shows a VC of type (b). It corresponds to rule 4 of the decision table in Fig. 2-10 on page 13.

3.3.2 Concurrent process VCs

The main inference rule for a concurrent process is R3. It is simple and warrants no substantial transformation as does R2. Consider the general form for a concurrent process shown in Fig. 2-4 on page 6. The formulation for the concurrent process VC will be based on this form with additional notation as given below.

$IB_1, \ldots, IB_m$ input histories of external buffers of P

$OB_1, \ldots, OB_n$ output histories of external

```
     receiver_int (pkt_in, sink, ack_out, next, queue)
   & pkt.seqno = next
   & (next + 1) in domain(queue)
-> receiver_int (pkt_in <: pkt,
                 sink @ (   pkt.mssg
                         :> apply(".mssg",
                                  range(consec(queue)))),
                 ack_out @ [reach(queue)],
                 reach(queue),
                 upper (queue, reach(queue)) )
```

Figure 3-4: Sample sequential process VC.

```
                                buffers of P
    B1_in, ..., Bk_in         input histories of internal
                                buffers of P
    B1_out, ..., Bk_out       output histories of internal
                                buffers of P
```

In order to precisely state the VC, it is necessary to define a binding between buffer names at the parent process level and those at the subprocess level. This binding performs the renaming function discussed in Section 3.2.2. The following notation is used.

```
inb(P, i, j)     binding of jth input buffer of subprocess i
                   in the environment of parent process P
outb(P, i, j)    binding of jth output buffer of subprocess i
                   in the environment of parent process P
nib(P, i)        no. of input buffers for P's ith subprocess
nob(P, i)        no. of output buffers for P's ith subprocess
```

The result of inb(P, i, j) will be a name of the form IBr or Br_in, according to whether the subprocess was passed one of P's input buffers or one of P's internal buffers. Similarly, outb(P, i, j) produces a name of the form OBr or Br_out. Because of the rules for passing buffer parameters, these functions necessarily define a one-to-one correspondence between the "port" names at the two levels. Using this notation, the single VC required for a proof of a concurrent process P is displayed in Table 3-2.

```
P1_ext (inb(P,1,1),...,inb(P,1,nib(P,1)),
                       outb(P,1,1),...,outb(P,1,nob(P,1)))
            & . . .
            & Pq_ext (inb(P,q,1),...,inb(P,q,nib(P,q)),
                      outb(P,q,1),...,outb(P,q,nob(P,q)))
            & B1_in initial B1_out
            & . . .
            & Bk_in initial Bk_out
          -> P_ext (IB_1, ..., IB_m, OB_1, ..., OB_n)
```

Table 3-2: Verification condition for a concurrent process.

An illustration of this type of VC is shown in Fig. 3-5. This example is the VC for the process depicted in Figures 2-2 and 2-3 on pages 5 and 6.

```
sender_ext (source, sndr_ack_in, sndr_pkt_out)
           & receiver_ext (rcvr_pkt_in, sink, rcvr_ack_out)
           & medium_ext (sndr_pkt_in, rcvr_pkt_out,
                         rcvr_ack_in, sndr_ack_out)
           & sndr_pkt_in initial sndr_pkt_out
           & rcvr_pkt_in initial rcvr_pkt_out
           & sndr_ack_in initial sndr_ack_out
           & rcvr_ack_in initial rcvr_ack_out
         -> transport_service_ext (source, sink)
```

Figure 3-5: Sample concurrent process VC.

3.4 Validity of Deductive System

The proof system we have presented is really quite simple, in spite of the elaborate notation of preceding sections. Nevertheless, it is necessary to offer some justification for the proof methods. One possibility would be to give a detailed, formal proof of the consistency of the deductive system. Since the proof rules are really just rules of inference in a logic, we could apply techniques of mathematical logic to establish consistency. What would typically be done is to show that the formal system has a model, which would in turn imply the consistency of the logic. It would be necessary to devise a model of computation and give an interpretation function that characterizes what it means for an assertion to be true in the computational model. Consistency of the system would then follow if we could show that the proof rules are theorems with respect to the model of computation and interpretation function. Owicki carried out such a program to show the consistency of a proof system for parallel programs [Owicki 75].

While we speculate that it would be possible to do this for our proof system, we take a more modest approach and merely present some informal arguments for the validity of the proof rules. It really is the case that the proof system is straightforward and can be justified informally. This is due in large part to the restricted model of concurrent processing upon which the specification and verification techniques are built. A highly pedantic proof would not inspire that much more confidence in the verification methods.

3.4.1 Sequential proof rules

The proof rules for sequential processes are based on well-known axiomatic proof techniques for sequential programs. Rule R2 is used to establish that an internal invariant holds for a process. The internal invariant itself is the analog of a loop invariant in the inductive assertion method. An entire sequential process could be likened to the program schema shown in Fig. 3-6.

```
SV := IV;
            loop
              assert internal invariant;
              wait on buffers and timeouts,
                 when event occurs, receive message M;
              select alternative based on M;
              evaluate response expressions,
                 send out messages;
              evaluate transition expressions,
                 update SV;
            end;
```

Figure 3-6: Model of sequential process operation.

Using this as a model, rule R2 essentially captures the cases that would result from a path analysis of the sequential code:

1. Path from entry to assertion point.
2. Set of alternative paths that traverse the loop.

Proof rule R2 then embodies the implicit induction proof necessary to establish the assertion as an invariant of this loop. Rule R1 is trivial and requires no special justification.

3.4.2 Concurrent proof rules

There is only one rule, R3, for proving a concurrent process. The external invariant of a concurrent process must hold at certain stable points, namely when all its subprocesses are idle. This ensures that the external invariant of all subprocesses are in effect at those same points in time. If any subprocess is itself a concurrent process, then its subprocesses must be idle, and so on. Every sequential process at the bottom of the process hierarchy must then be idle at the designated points of time.

History variables are assumed to be resident (if they were to exist) in sequential processes. They are implicitly accessible to higher levels in the process hierarchy by virtue of the buffer parameters, which are explicitly passed. The important characteristic of history variables is that they are *not* shared variables. For each physical buffer, its input and output histories are accessible (for updating purposes) to exactly one sequential process. Since a process invariant accesses only its own histories and there is no interference possible from other processes, the proof of the external invariant of a sequential process is independent of the actions or the proof of any other process [Owicki 76].

Now consider a level one concurrent process P. Its subprocesses are sequential processes $P_1,...,P_n$. Suppose we use rule R3 to establish P's external invariant. In the course of this proof, the histories of $P_1,...,P_n$ are renamed to a distinct set of history variables. Because of the parameter passing rules for buffers, there is a one-to-one correspondence between the renamed variables and the variables as they are known to $P_1,...,P_n$. It can not happen that a history from P_i and a history from P_j, $i \neq j$, are renamed to the same variable. Given that each external invariant E_i is true, then, it is valid to conclude

$$AND_{i=1}^{n} \text{rename}(P, E_i)$$

because this is simply a conjunction of formulas over disjoint sets of variables. Therefore, a proof of E using R3 properly establishes the truth of E.

To generalize, all we would need to do is appeal to an argument based on a complete induction over the structure of a process hierarchy, which is a strict tree structure. This would establish the validity of rule R3 in the general case. We have sketched the main ideas of such a proof, but we will not pursue its details.

3.4.3 Sufficiency of external invariants

It should be clear by now that the proof system is consistent. The proof rules give us a method to establish the truth of external invariants. But these assertions hold at very restricted points in time. One might legitimately ask: Is an external invariant really sufficient to assure safe behavior of a concurrent process? Is there anomalous behavior that this form of assertion is unable to rule out?

We raise this concern because an external invariant does not hold after every individual send operation, only in between events. If we were to attempt this style of specification with conventional procedural programming statements, we would indeed have something to worry about. It would be possible to write procedures, prove them correct and still have undesirable behavior present. Suppose, for instance, that we have the following program fragment.

```
loop
                    assert A;
                    . . .
                    send bad data;
                    loop
                       if false then leave end;
```

```
            . . .
        end;
        . . .
    end;
```

Assume the external invariant is required to hold when the program is at the assert point.

Generating VCs for such a fragment would produce (among others) a VC of the form:

```
A & false & other path conditions
              -> A with (effects of sending bad data)
```

This formula is trivially true, so we would complete the proof of our program, but it could still send out the bad data before going into the infinite loop.

Fortunately, our sequential process model does not have this problem. Generally speaking, we need to consider any potential source of nontermination that might cause such an anomaly. There are only two cases to consider.

1. *Nonterminating actions*. Our sequential processes do not use conventional iteration statements such as while loops or repeat loops. All event processing is achieved by evaluating expressions. These expressions contain calls on recursive functions, which could be nonterminating if weak restrictions were imposed. In Chapter 4, we present a function definition mechanism that only admits well-defined, total functions. A proof of this fact will be given. Hence, a sequential process cannot introduce nontermination on its own.

2. *Deadlock*. The other possibility is that a process could become deadlocked during an attempt to send out response messages. Because of strict modularity, though, a sequential process only has information about its own behavior, and not about any other process. Hence, a proof of deadlock cannot be performed within the confines of a sequential process. It is safe to assume that send operations eventually terminate because it is impossible for a sequential process to prove otherwise.

We conclude, therefore, that the kind of anomalous behavior we feared is ruled out by the various features of our methodology.

Chapter 4
SPECIFICATION LANGUAGE

Machine assisted verification requires the use of a formal specification language with well defined semantics. Such a language is used for expressing assertions and verification conditions, and in our case, is also used for writing the expressions of process definitions. The language we propose is based on a kernel logic for defining and operating on inductively constructed objects. A number of predefined operations are provided. In addition, a facility for adding user defined functions is included. Sufficient constraints on this definition facility are imposed so that a user cannot introduce any ill-defined functions that might render the resulting theory inconsistent.

4.1 Motivation and Goals

There are basically two reasons for introducing this language. First, we wish to provide a powerful means for expressing specifications and assertions, which can be very complex in typical protocol applications. We would like to define our concepts in a generic way and state properties of these concepts in a generic way. This would aid in the development of reuseable theories and bodies of knowledge. Emphasis is placed on primitives that are useful both to support our verification methods and to support the manipulation of abstract data objects commonly used in protocol models.

Second, we would like to take advantage of existing verification systems and theorem provers that are based on recursive function methods. Among these are the Boyer-Moore theorem prover [Boyer 79] and the AFFIRM verification system [Musser 80], both of which allow axiomatization of recursive functions and induction proofs of their properties. Data objects in such systems are inductively constructed and hence finite. Quantifiers are avoided in favor of recursive functions, which can provide a finite quantification capability. The features of this language have been designed to facilitate mechanical proofs with such systems, especially the Boyer-Moore prover. Language features that might otherwise be desirable, but are difficult to capture within a quantifier-free, recursive function based theory, have been deliberately omitted from the language.

To better understand the design decisions of this language, let us consider the role that humans play in the verification process. We propose a simple division of labor for this activity.

1. *Theory building*. It is helpful to develop concepts that are generic to a particular class of protocols, or for that matter, any other problem domain. Formalization of such concepts, together with proofs of their properties, constitutes the development of a theory. The degree of specificity of these theories varies, so it is natural to think in terms of hierarchies of theories. The result of this work should be made available to verification end-users.
2. *Application verification*. The verifiers of protocols or other application problems are the actual end-users. They make use of theories that have been provided by others. In this way, the work of the theory builders can be used and reused by many different verifiers, thereby avoiding a lot of duplicate effort.

In the world of software production, there is an analogous division of labor between system programmers

and application programmers. This division serves a useful purpose and we should strive for a similar mode of operation in our verification endeavors. To this end, the specification language has been designed with a high degree of generics in mind. It is generic mechanisms that enable us to develop reuseable theories. A general concept can be used repeatedly for different problems, or can be instantiated in different ways for the same problem. Two generic mechanisms are included in our language design: parameterized data types and functional arguments, that is, passing names of functions as parameters in a function call. Constraints are placed on these features to keep their use within reasonable limits.

Our domain of discourse consists of typed data objects and functions on those data objects. Data types may be constructed from a fixed set of primitive types and a fixed set of data structures. There is no capability for defining arbitrary abstract data types using, for example, an algebraic specification technique. We feel that a small set of built-in structures suffices for the applications of interest. However, the addition of new built-in data structures at some later time is not precluded by this approach.

With the existence of parameterized data types, we can refer to objects whose type is only partially specified, enabling us to define operations on large classes of objects. If we were designing a programming language, such a feature would have to be carefully considered because of its potential impact on implementations. Because we are designing a specification language, though, we are free of such concerns and need only worry about the ramifications for proofs.

Operations on objects are performed by functions. These are functions in the mathematical sense, meaning they have no side effects. A number of predefined functions are included in the language. In general, these are inadequate to define all the concepts of interest, so a facility for introducing user defined functions is also provided. It allows the definition of a restricted class of recursive functions. In accordance with the division of labor we proposed earlier, it is intended that recursive functions be defined primarily by the theory builders, leaving the application verifiers to define primarily nonrecursive functions. This distinction is purely methodological, not theoretical. Recursive functions give rise to the need for induction proofs. It is hoped, therefore, that the application verifier can be spared from having to engage in much of this kind of proving, allowing him to concentrate on the problem at hand.

A powerful generic mechanism is allowing functions to have functional arguments. This is an additional way to parameterize a function and generalize the operation it performs. The combination of parameterized data types and functional arguments are mutually reinforcing, yielding a capability to define highly general concepts. It also makes possible the definition of functions that provide general purpose recursion schemes. This helps justify our strategy that a verifier's introduction of new recursive functions be minimized. Rules for passing functional arguments and for defining functions with functional arguments must be made strict enough to ensure that a user cannot introduce nonsense through careless use of this feature. Rules have been imposed that make it possible to prove that all user functions, even recursive ones, are guaranteed to be well defined.

Many ideas have been borrowed from existing languages and systems. These include the Gypsy specification language, the Boyer-Moore theory, the AFFIRM specification language, applicative programming languages such as LISP and APL, and other programming languages. It is important to remember, though, that what we are defining is an assertion language or expression language, not a programming language.

4.2 Data Objects

All data objects are assumed to be inductively constructed objects. This means that every object is either one of several *bottom objects*, or is the result of applying one of several *constructor functions* to other objects. The universe of all possible data objects generated by these functions is a countably infinite set. There is also a set of *destructor functions* that return the component objects from applications of the constructor functions.

A data type defines a subset of the universe of data objects. We say that a type T1 is a subtype of another type T2 if the set of objects defined by T1 is a subset of those defined by T2. We say that two types are equal if they define the same set of objects. Data types are classified as either simple types or structured types. A simple type completely defines its set of objects. The simple types are boolean, integer (and its subtype natural) and a type called "token," which is similar to Pascal-like enumeration types. Structured types define objects

that are made up of one or more component objects. The complete type specification of a structured type depends on the types of its components. The structured types of the language include record, sequence and mapping. Records and sequences should be familiar objects; mappings form a special subtype of sequences.

4.2.1 Type definitions

Types are specified by type expressions. These contain references to named types, either predefined or user defined. For example,

```
boolean
                    sequence of integer
```

are references to types using predefined names. User type definitions may be written as

type T = <type reference>

Type definitions may also contain parameters.

type T (t_1, ..., t_n) = <type reference>

Here the t_i are type parameters or type variables. They may only take other type names as actual parameters; data valued parameters are not allowed. The type parameters are used in the body of a type definition:

type seq2 (T) = sequence of sequence of T

References to a parameterized type are made in the obvious way:

type SSN = seq2 (natural)

Type definitions may not be recursive, implying that any type reference may be fully expanded to a tree structured type expression. Note that the predefined structured types are really just parameterized types with a special syntax for the type reference.

4.2.2 Simple types

The boolean type includes the two distinct bottom objects "true" and "false." They have the following operators.

not, and(&), or, imp(->), iff, eq(=), ne, id

The function "id" is the identity function. Note that all types have the eq, ne, and id operators defined for them so from now on we will not bother to repeat that fact for each new type introduced.

Integer is the type that represents the full set of integers, from negative infinity to positive infinity. The following well known operations on integers are defined.

+, -, *, div, mod, **, lt(<), le, gt(>), ge, min, max

Often it suffices to work with the natural numbers (nonnegative integers) rather that the full set of integers. This is the case for all examples we will use in this thesis. The type natural is defined to be this subtype of the integers. The same operators are provided, although their semantics will be different for natural numbers. Any operation that would ordinarily produce a negative result for a given combination of arguments is forced to produce a result of zero for those arguments.

The naturals are a simple kind of inductively generated object. For example,

```
bottom object              zero
                  constructor function      add1(n)
                  destructor function       sub1(n)
```

could be used to generate naturals. The unique representation for a number N would then be:

add1(add1(... add1(zero) ...))

where N applications of add1 are used. The axioms for the destructor functions are

```
sub1(zero)     =  zero
                      sub1(add1(N))  =  N
```

With these as primitives it is possible to define all the operators on naturals as if they were user defined recursive functions. We will, however, continue to assume they are predefined operators with their familiar infix notation.

A type "token" is included, which is meant to be similar to the notion of enumeration type. Syntactically, an object of this type is written as an identifier. The token type is different from an enumeration type, though, in that it includes the set of all possible identifiers, rather than being restricted to a small set of values. If such restrictions are desired, the user must impose them explicitly by writing predicates about the variables involved. Although it is possible to inductively define tokens and give operators for manipulating them, we do not do so. Our only interest in them is as mnemonic representations for alternative values. Consequently, the only operation defined on tokens is the equality operator.

4.2.3 Structured types

The record type is quite similar to the record type of Pascal or Gypsy. A record type is defined by:

```
record (fld_1 : type_1;
                                  . . .
                              fld_n : type_n)
```

where each field or component may be of a different type. The field names are not significant for determining type equality. In other words, two record types are equal if they have the same number of fields and their component types match, regardless of whether the field names match.

There are three operations available for record objects.

```
[rec: e1,...,en]   constructs a record with component
                        values e1,...,en
   R.f                 extracts the field named f from record R

   R with (.f := e)    returns a record value that is R with
                         its f field changed to value of
                         expression e
```

If it is clear from context that a record object is called for, then [rec: e_1,...,e_n] may be abbreviated [e_1,...,e_n].

The sequence type is similar to that of Gypsy. It also follows closely the axiomatization for sequences used in AFFIRM. A sequence type is defined by:

sequence of T

It represents an object that is a finite sequence of elements of type T. The following operations are provided.

```
[seq: e1, ..., en]                 explicit construction
       []          or    null            empty sequence
       S <: e      or    S apr e         append element right
       e :> S      or    e apl S         append element left
       S1 @ S2     or    S1 join S2      append sequence
```

As before, the "seq:" indicator is usually omitted.

Sequences are axiomatized by the bottom object "null" and the constructor function "apr" (<:). All the other operations above can be defined in terms of these. For example,

$$[e_1,...,e_n] = (...(\text{null} <: e_1) <: ...) <: e_n$$

Components of a sequence may be accessed using the functions:

```
first(S)    /    last(S)          first/last element of S
      nonfirst(S) / nonlast(S)          tail/ head of S
```

Last and nonlast are the axiomatic destructors for sequences; first and nonfirst are defined functions. The axioms for sequences are as follows.

```
nonlast (null)    =  null
                     nonlast (S <: e)  =  S
                     last (null)       =  default(T)
                     last (S <: e)     =  e
```

The result of trying to take the last element of a null sequence is the default object for component type T.

A number of predefined operations on sequences are available. Among the most important are:

```
size(S)             number of elements in S
             e in S             test for membership
             S1 initial S2      initial segment relation
             S1 subseq  S2      subsequence relation
             S1 follows S2      "subset" relation
```

An extremely useful data type is the mapping type. A mapping data object consists of a set of ordered pairs that define a correspondence between certain integers and elements of some type T. Hence the name mapping connotes the mathematical concept of mapping, except that we have restricted the allowable domain to be integers. Actually, it is convenient to restrict it even further to the natural numbers. A mapping type is defined by the type expression

mapping of T

for range values having type T.

A mapping is most naturally envisioned as a set of ordered pairs. Sets cause difficulties, however, when represented by inductively constructed objects. This is because the abstract value of a set ignores the ordering of its elements, so that many different concrete objects can represent the same abstract object. Proving facts about equalities of such abstract objects is consequently tedious. It is for this reason, and others as well, that we have restricted the domain values to be numeric. It is then possible to order the pairs of a mapping and make use of a sequence representation. This in turn guarantees that a one-to-one correspondence exists between abstract and concrete objects, yielding more tractable proofs. The price we pay for this is that not all possible concrete objects are valid abstract objects, so we are obliged to constrain the allowable set of concrete objects.

To define the representation for a mapping, let us begin by defining a type for its ordered pairs.

type pair (T) = record (dom: natural; rng: T)

The "pair" type consists of a domain value and a range value. A pair sequence type can then be defined as

type pair_seq (T) = sequence of pair (T)

This type is used as the concrete representation for a mapping type.

To be a legitimate mapping, an object of type pair_seq must satisfy the constraint that its domain values are strictly increasing with the sequence ordering. An example of a valid mapping value is:

M = [[1,a], [2,b], [4,c], [9,d]] (2)

It is possible to define operations on this concrete type that are ultimately intended for use on mappings, but whose definitions do not depend on the constraint mentioned above. These include:

```
domain(M)     sequence of domain values
```

```
range(M)      sequence of range values
```

Furthermore, any operations valid on general sequences are applicable to the pair sequence type and we therefore allow them for mappings. Unlike the usual methods for data abstraction, we make no attempt to hide the concrete representation for this abstract type. On the contrary, we want to be quite open about it so as to share common functions, and more importantly, to share common lemmas.

Now let us discuss some mapping specific operations. The basic constructor function for mappings uses the "with" style of notation introduced for records. The construction

M with ([i] := e)

produces a new mapping from M by placing the pair [i,e] at the appropriate position. If a pair with domain value i already exists in M, it is replaced by [i,e]. For the sample mapping shown in 2 we have

```
M with ([5] := x)  =  [[1,a], [2,b], [4,c], [5,x], [9,d]]
      M with ([2] := x)  =  [[1,a], [2,x], [4,c], [9,d]]
```

Note that "with" is not a constructor in the same sense as the "apr" (<:) operator for sequences. The with-function is defined in terms of sequence operations; there are no axioms for mappings.

Several functions are provided to extract values from mapping objects, in addition to the domain and range functions.

```
M[i]            range value for domain value i
           upper(M, i)     upper portion of mapping with
                              domain values at least i
           lower(M, i)     lower portion of mapping with
                              domain values at most i
```

4.3 Undefined Functions

Function calls serve to model both data objects and operations performed on them. All of the operations discussed thus far have been described as "predefined" functions. Actually, they can be divided into undefined and defined (or definable) functions. This distinction is for theoretical reasons; a user need not be overly concerned with it . The semantics of all these functions must be given in some way, though. For undefined functions, this means giving a set of axioms. For defined functions, a set of function definitions is required. We will shortly present a mechanism for introducing function definitions. This definition mechanism can and will be used to define the semantics of predefined functions that are otherwise unaxiomatized. It also provides the means by which users introduce new functions.

We have already introduced most of the undefined functions. These include the bottom functions and constructor functions for data objects. There are a few additional undefined functions.

The equality operator is an important function whose effects cannot be expressed in terms of other functions of the language. The axiom for equality is that for explicit values x and y, x = y if and only if x and y are identical expressions. An explicit value is an expression containing only bottom functions and constructor functions; in short, it is the canonical form for a data object.

We have not yet discussed the conditional expression, or if-expression. This is a special, undefined function having the form

```
if <boolean_expr>
                        then <true_expr>
                        else <false_expr>
                    fi
```

The value of the if-expression is <true_expr> if the test evaluates to true, and is <false_expr> if the test evaluates to false.

Using this small set of undefined functions and the function definition facility, it is possible to define all the other functions of the language. The logical connectives, for example, can all be defined in terms of true, false and if.

4.4 Function definitions

We envision the development of a theory to proceed in bottom-up fashion, as it does in the Boyer-Moore system. Functions are defined in terms of previously defined functions. Stating a function definition is just another way of axiomatizing a concept. A function definition gives rise to an axiom or set of axioms that declares how a function call is to be rewritten. The semantics of the function definition mechanism consists of a set of rules for determining when a definition can be allowed. The property that we would like to be true of such rules is that they only admit function definitions that are guaranteed to axiomatize well-defined mathematical functions. Any definitions that could potentially lead to logical inconsistencies should be rejected. Later in this chapter we will show that the rules for defining functions do in fact admit only well-defined, total functions.

We will begin by discussing definitions for nonrecursive functions. Then we will consider the definition of functions with a restricted kind of recursion. Finally, we will consider the introduction of functional arguments into definitions.

4.4.1 Nonrecursive functions

The general form for a function definition is:

<function header> = <expression>

A function header has the syntax

function <name> (<param 1>; ...; <param n>) : <result type>

where parameters are declared by

<identifier list> : <type expression>

Following is an example of such a definition.

```
function and (P, Q: boolean) : boolean =
                 if P then Q else false fi
```

Within the body of a function, any of the predefined functions may be called as well as any user functions that have already been defined. Indirect recursion is therefore prohibited by this rule. Type constraints must be observed in function calls. The type of an actual parameter must be a subtype of the type declared for the formal parameter.

When the type expressions of formal parameters contain type parameters, actual type parameters must be supplied. Consider the example below, which uses the type "pair_seq" defined on page 33.

```
function in_range (S: pair_seq(natural);
                                  e: natural): boolean =
                 e in range(S)
```

Here the type parameter of pair_seq is filled in with the type natural. This makes the type of the expression "range(S)" in the function body be "sequence of natural," thereby satisfying the type constraints for the "in" operator.

Often it is desirable to leave the type parameters of formal arguments unspecified, allowing the function to be generic to some extent. If there are multiple parameters, though, it may be necessary to require that some of the type parameters match. Consider the function "in_range" above, for example. We might want to generalize it by allowing S to be a pair sequence of any type T. But if we do, we must ensure that the argument e is an

object of the same type T.

The method for specifying generic types of arguments is illustrated below.

```
function in_range (S: pair_seq(?T);
                                   e: ?T): boolean =
                  e in range(S)
```

The notation "?T" indicates that T is a formal type parameter of the function. The actual argument for S may be any pair sequence and the actual argument for e may be any object whatsoever. The constraint, however, is that S must be a pair sequence of whatever type e is. This rules out function calls with unmatching argument types and allows the body of the function to be evaluated under whatever pair of objects are passed in a valid function call.

It may occasionally happen with generic arguments that the type of the result is itself a function of the types of the arguments. In this case, the result type expression may refer to type parameters of the arguments. They are written without the "?," however, because the result type is not arbitrary but rather derived, much like the value of the function itself. The following function definition illustrates this usage.

```
function tuple (x: ?T1; y: ?T2):
                              record (x: T1; y: T2) =
                  [rec: x, y]
```

Keep in mind that if a formal parameter is declared to be generic, then nothing is known about the objects that may be passed through that argument position. Its use in expressions is necessarily limited to function calls where it can act again as a generic object. It may not be passed as an argument that is supposed to have a specific type. Ultimately, the only primitive operation that can be applied to it is a test for equality with another object that is known to be of the same, though unspecified, type.

4.4.2 Recursive functions

Recursion is the means by which we define "interesting" functions. We limit ourselves to a special class of recursive functions -- those with recursive calls that have a simple *structure reducing property*. Every recursive function is required to have a single *measured variable* that can be used to justify its definition. It is justified if each recursive call passes in the measured variable position an object that is "smaller" than the original object. This restriction makes it a simple matter to ensure that a recursive function is well defined. This definition principle is strictly less general than many other schemes for defining recursive functions, such as that of the Boyer-Moore theory, but it does allow the definition of a large variety of functions for expressing data operations and assertions. Intuitively, we feel it is adequate for the needs of verifying abstract process models.

A measured variable is a formal parameter whose type is one of the inductively constructed data types. For the present version of the language, only integers and sequences are used in this way. (Actually, records theoretically fall into this category, but there is no way to meaningfully define recursion on records.) To define recursive functions, we provide notation that makes it very explicit how and why the recursion is well defined. Simple syntactic and semantic checks will suffice to establish that a function definition is justifiable.

To define a recursive function, a user must single out the measured variable. It must have type natural or sequence, where the component type of a sequence is irrelevant. The body of the function is written in a special way to convey this information and to provide a special kind of conditional expression for evaluating the function in one of two ways, depending on the value of the measured variable. An example of the definition scheme is shown in Fig. 4-1. This is a definition of the sequence append operator.

In this function, S2 is the measured variable. The "object" construction is a special kind of case expression. It contains two clauses, specifying the value of the function in terms of two alternative expressions. Since S2 is a sequence, its outermost operator will be either null or an application of the sequence constructor (<:). In the latter case, two bound variables are introduced to enable us to refer to nonlast(S2) and last(S2). When join is evaluated, the expression "S1" is its value for S2 = null, and the expression "join(S1, S) <: e" is its value for S2 = S <: e. Note the similarity between this kind of definition and an algebraic axiomatization, such as might

```
function join (S1, S2: sequence of ?T) : sequence of T =
          object S2 is
            null   :  S1;
            S <: e :  join(S1, S) <: e
          end
```

Figure 4-1: Sample recursive function definition.

be given in an AFFIRM style notation:

```
S1 join null  ==  S1
                S1 join (S apr e) == (S1 join S) apr e
```

In order to precisely state the semantic rules for admitting recursive function definitions, we present the general form for a recursive function in Fig. 4-2. The interpretation of this schema is as follows.

- X is the measured variable. It must have the type of an inductively constructed object (currently natural or sequence).
- The first case in the conditional is X = bottom object. Associated with this case is an expression that must be devoid of recursive calls. The variable X may appear in this expression, but it is unnecessary since its value is known to be that of the bottom object.
- The second case of the conditional is for X = cons(v_1,...,v_m). This is an application of the constructor function for X's type. Variables v_1,...,v_m are bound to the components that make up X. The expression associated with this case may contain recursive calls. In each such recursive call, X must not appear in the measured variable position. Instead, one of the v_i must be supplied in this position. X may appear in any nonrecursive call or in nonmeasured positions of a recursive call. Where it is allowed, its value may alternatively be expressed as cons(v_1,...,v_m).
- If a type with multiple constructors is introduced at some later point, it is only necessary to extend this schema by including one case for each constructor function.

These rules are sufficient to guarantee termination of a recursive function on any combination of argument values that satisfy the type constraints. It is important to note that nothing need be proved when a function is defined. The rules constitute very simple semantic checks of the kind that any language processor could perform.

```
function name (X: T; Y1: T1; ...; Yn: Tn) : RT =
            object X is
              bottom :  expr1 (X, Y1, ..., Yn);
              cons(v1,...,vm) : expr2 (v1,...,vm,X,Y1,...,Yn)
            end
```

Figure 4-2: General form of a recursive function definition.

4.4.3 Functional arguments

Schemes for passing functional arguments are characteristic of applicative languages. Formal function parameters do not take normal data objects as arguments. Instead they take the names of functions that have previously been defined. A mechanism is provided for using these names in the body of a function, either by causing the functions named to be applied or by passing them as function arguments in other function calls. There is a strict separation between function arguments and data arguments. Our use of function arguments is purely applicative. Functions take functions as arguments to produce data objects as results; they do not take functions as arguments to produce other functions as results.

There are basically three language mechanisms necessary for providing a functional argument scheme.

1. A method and a notation are needed for declaring formal function parameters.
2. A way is needed to apply formal function parameters in function bodies.
3. A notation for passing actual function parameters is required.

We propose features that extend the method for defining functions outlined in the preceding sections. In particular, it will be possible to define recursive functions having function arguments.

The means for declaring formal function arguments can be thought of as an extension of the data typing mechanism. The type of a formal data argument implicitly defines the set of objects that may be passed in a function call. By using an analogous concept, the "type" of a function, we may give function parameter declarations that similarly constrain the functions that may be passed as actual parameters. In this way, we can arrange that only functions that can meaningfully be applied in the body of a function will ever be passed as arguments. As a result, anomalies in the definition of functions due to the function argument feature are totally avoided. The effect of these restrictions is that the functions we can define will not be all those that might otherwise be mathematically well defined. Nevertheless, the restrictions are still ones we can live with. They allow a wide range of expressive power in defining useful functions.

A formal function parameter is declared by the notation

<identifier list> : <function template>

A *function template* is a special kind of expression for declaring the type of a function argument. Essentially it contains information to characterize the functionality of a function, that is, its domain and range. As an example, the template for a unary function that maps from naturals to booleans is:

fun (natural : boolean)

In the general case, we must accommodate in a function template parameters that might themselves be function arguments. Consequently, we generalize to the notion of "parameter type," which can represent either a data argument or a function argument. A parameter type expression is recursively defined as follows.

```
<par type>  ::= <data type>  |  <func type>
    <func type> ::= fun (<par type> {,<par type>}* :
                              <data type>)
```

It should be clear that a function template could be represented by a tree structure, with its leaves denoting data parameters and its nonleaf nodes denoting function parameters.

A formal function argument may be used in two ways within the body of a function: it may be applied to a set of actual parameters or it may be passed as an actual function parameter in other calls. To apply a function argument, it suffices to write the parameter name in the position normally occupied by an explicit function symbol in prefix notation. Depicted in Fig. 4-3 is a function definition that illustrates the technique. This function computes its result from a sequence S by applying the function f to each element and collecting the resulting objects into a new sequence.

```
function apply_to_all (f: fun(?T1 : ?T2);
                          S: sequence of ?T1) :
                    sequence of T2 =
        object S is
          null   :  null;
          v <: e :  apply_to_all(f, v) <: f(e)
        end
```

Figure 4-3: Function definition with function argument.

When calling a function with function arguments, there are two ways to supply actual function parameters. The first is with formal function parameters, as illustrated in the figure. The second is to pass the explicit name of a function that has already been defined. For this latter case, the name under which a function was defined is

simply written unmodified, as it is already an identifier. For predefined functions with special symbols, quotes are used around the symbol, as in "+". As a special case, ".fld" may be written to denote the function of extracting the field fld from a record object. For example, if S is a pair sequence, it follows that

domain(S) = apply_to_all(".dom", S)

In order to ensure that a user cannot introduce any ill-defined functions, a number of restrictions are placed on the use of the function argument mechanism.

- The only things that may be passed in data argument positions are data objects (expressions) and formal data parameters.
- The only things that may be passed in function argument positions are explicit function names (identifiers or quoted forms) and formal function parameters.
- To pass an explicit function name as an actual parameter, the function must already be defined. A function may not pass its own name as an actual within its body.
- Any actual function parameter must have a header that is compatible with the function template of the corresponding formal parameter. To be compatible means that the tree structure represented by the header and template are identical. Data types in the actual parameter, however, need only be subtypes of those in the formal parameter.
- In any recursive call, the actual function parameters must be the same as the corresponding formal function parameters. In other words, a recursive call does not change any of the function arguments in the chain of recursive calls.

As we will show at the end of this chapter, these rules are sufficient, even in the presence of recursion, to guarantee that any set of functions satisfying them will be well defined.

To illustrate the entire function definition mechanism, we present a set of important predefined functions. They make full use of all the facilities for parameterized types, recursive definitions and functional arguments. Some very general operations are performed by these functions, prompting us to regard them as axiomatizing some important *recursion schemes*. What is significant is that they are in a form that makes them useful as general purpose "function applicators." The first we have already seen as the "apply_to_all" function.

1. Apply

   This is the simplest and most basic of the group. It takes a unary function and applies it to each element of a sequence to produce another sequence of the same length. It is analogous to MAPCAR in LISP.

```
function apply (f : fun(?T1 : ?T2);
                S : sequence of ?T1) :
               sequence of T2 =
    object S is
      null   :  null;
      v <: e :  apply(f, v) <: f(e)
    end
```

   Informally, its effect is as below, where $S = [S_1,...,S_n]$.

$$apply(f, S) = [f(S_1),...,f(S_n)]$$

2. Couple

   Couple is like apply except that it applies a binary function to pairs of elements drawn from two different sequences to produce a new sequence. In normal use the sequences are of equal length. If the lengths are unequal, the length of the second sequence is used.

```
function couple (f : fun(?T1, ?T2 : ?T3);
                 S : sequence of ?T1;
                 T : sequence of ?T2) :
                sequence of T3 =
```

```
    object T is
      null   :  null;
      v <: e :  couple(f, nonlast(S), v)
                     <: f(last(S), e)
    end

couple(f, S, T) = [f(S1,T1),...,f(Sn,Tn)]
```

3. Adjoin

Adjoin is used to apply a binary function to each adjacent pair of elements in a sequence. It produces a sequence result having length one less than that of the sequence argument.

```
function adjoin (f : fun(?T1, ?T1 : ?T2);
                 S : sequence of ?T1) :
                sequence of T2 =
    object S is
      null   :  null;
      v <: e :  if v = null
                  then null
                  else adjoin(f, v) <: f(last(v), e)
                fi
    end

adjoin(f, S) = [f(S1,S2),...,f(Sn-1,Sn)]
```

4. Extend

Extend applies a binary function that has a sequence valued first argument and a single object as its second argument. For each element of the sequence, extend applies the function to the initial segment of all elements up to the current one and the element itself. It produces a sequence valued result.

```
function extend (f : fun(sequence of ?T1, ?T1 : ?T2);
                 S : sequence of ?T1) :
                  sequence of T2 =
   object S is
     null   :  null;
     v <: e :  extend(f, v) <: f(v, e)
   end

extend(f, S) = [f(null,S1),...,f(nonlast(S),Sn)]
```

5. Reduce

Reduce gets its name from the corresponding APL operator. Its purpose is to reduce a sequence to a single value. It starts by applying a binary function to some initial value and the first element of the sequence. The function is applied a second time to the result of the first application and the second element of the sequence. This continues until all the elements have been consumed. If the sequence is null it returns the initial value.

```
function reduce (f : fun(?T1, ?T2 : ?T1);
                 I : ?T1;
                 S : sequence of ?T2) : T1 =
    object S is
      null   :  I;
      v <: e :  f(reduce(f, I, v), e)
    end

reduce(f, I, S) = f(f(...f(I,S1)...),Sn)
```

6. Scan

Scan again is analogous to the APL operation with the same name. Its purpose is to produce a sequence of values by applying reduce to each successive initial segment of the sequence argument. In other words, it displays all the partial results that are computed along the way to evaluating reduce(f,I,S).

```
function scan (f : fun(?T1, ?T2 : ?T1);
               I : ?T1;
               S : sequence of ?T2) : sequence of T1 =
    object S is
      null   :  null;
      v <: e :  scan(f, I, v) <: reduce(f, I, S)
    end

scan(f,I,S) = [reduce(f,I,[S1]),...,reduce(f,I,S)]
```

A list of definitions for the functions used in this thesis can be found in Appendix A.

4.5 Lemmas

Function definitions give us the capability to express concepts for modeling and specification purposes. Properties about these concepts are expressed as lemmas. A lemma is simply a logical formula that states some fact about functions. Lemmas are derived properties. They must be shown to follow from the basic axioms, function definitions and previously proven lemmas.

Among the most important tools for proving lemmas are inference rules for applying structural induction. The proof of a formula proceeds by selecting an induction variable, instantiating a base case and an induction step with the formula and induction variable, and proving both of the resulting formulas. For a sequence valued induction variable S and a formula P, we must prove the two cases

```
1. P with (S := null)
                    2. P -> P with (S := S <: e)
```

Induction on an integer variable is similar. Not all lemmas need to be proved by induction, but most useful properties of recursive functions do.

The actual form for expressing a lemma is very similar to the form for a function definition. The variables and their types are declared in the same way that they are within a function header:

lemma <name> (<param list>) = <boolean expression>

As an example, we can restate an equality shown in the previous section as a lemma.

```
lemma domain_apply (S: pair_seq(?T)) =
                   domain(S) = apply(".dom", S)
```

The parameters of a lemma have type constraints that must be satisfied in the same way that they must be for functions. When using a lemma in the proof of a formula, its formal parameters are instantiated and their types checked against those of the actual parameters. If the type constraints are not met, then the application of the lemma is not valid.

4.6 Existence of Defined Functions

In the absence of restrictions on the use of functional arguments, it would be easy to introduce function definitions that lead to an inconsistent theory. Consider, for example, the following ill-defined functions.

```
function g (f: fun(T : boolean); x: T) : boolean = not f(x);

    function h (x: T) : boolean = g(h, x);
```

Upon expanding any call on h, a paradox results:

$$h(y) = g(h, y) = \text{not } h(y)$$

We will show that the admission of function definitions satisfying all semantic requirements cannot lead to an inconsistent theory. To establish this result we will demonstrate that for every function definition there does in fact exist a total function that is axiomatized by that definition, thereby showing that the axiomatization has a model. This in turn is established by showing how function definitions may be used to "compute" the result of a function call and showing that such computations must always terminate in a finite number of steps, yielding a unique result. Axioms for the primitive undefined functions also participate in this evaluation process.

Let us begin by precisely defining our terminology and basic concepts.

Definition 4.1: The *domain of discourse*, D, is the set of all legitimate objects of a theory. More specifically, D is the set of terms generated by all possible applications of the bottom and constructor functions, where type restrictions are observed. Each such term represents a distinct object and is referred to as an *explicit value*.

Definition 4.2: The set of *callable functions*, F, consists of names for the undefined functions and all defined functions that have been admitted to the theory. At any given point in time, F is finite, although it grows as new functions are defined and added to the current theory.

A function f with m function arguments and n data arguments has the functionality:

$$f: F^m \times D^n \rightarrow D$$

The types of its formal parameters will restrict the actual domain of f to some subset of $F^m \times D^n$. Our goal is to show that all functions are well defined and total over their respective domains. We will accomplish this in two steps: first we show it for the case with no function arguments and afterwards we will extend the proof to handle the addition of the function argument feature.

By giving an effective procedure for computing the value of a function over all elements of its domain, its existence will be assured if we can just show the computations always terminate in finite time with deterministic results. The method of computation will be a *term rewriting process*. The axioms and function definitions are used to derive a set of unconditional rewrite rules. Each step in the computation consists of the application of a single rewrite rule to rewrite some subterm of an expression. Thus, we will cast our problem in terms of the manipulation of symbolic expressions via rewrite rules. Some of the terminology and results that follow are borrowed from existing work on rewrite rule systems [Knuth 70, Musser 80].

4.6.1 Rewrite rule concepts

Definition 4.3: An *expression* or *term* is defined recursively as either a variable v, or an n-place function symbol applied to n expressions, $f(e_1,...,e_n)$. Variables and function symbols are assumed to be drawn from two distinct sets of names or symbols. Variables are therefore distinguishable from nullary functions. A *ground expression* (or ground term) is an expression containing no variables. Note that an explicit value, as defined earlier, is a special kind of ground expression.

Definition 4.4: A *substitution* S is a mapping that assigns expressions to variables. A *substitution instance* of an expression E, written E/S, is E with its variables replaced according to S.

Definition 4.5: A *rewrite rule* is a pair of expressions, which we write L -> R, such that every variable

appearing in R also appears in L. L is assumed to contain at least one function reference. An expression A *directly reduces* to B by applying rule L -> R, also written as A -> B, if there exist subexpressions a, b of A, B and a substitution S such that a = L/S, b = R/S and B results from A by replacing subexpression a in A by b. Given a set of rewrite rules RR, an expression E is *reducible* if it can be directly reduced using any of the rules in RR; it is *irreducible* otherwise.

Definition 4.6: The "directly reduces" relation is extended to its transitive closure by saying that A *reduces* to B, written A --> B, if there is a sequence of direct reductions A -> E_1 -> ... -> E_n -> B. A set of rewrite rules RR is said to have the *finite termination* property if no infinite sequence of direct reductions is possible under RR. Let A ==> B, called a *terminal reduction*, hold if A --> B and B is irreducible. A set of rules has the *unique termination* property if whenever A ==> B and A ==> C, B = C. If a set of rules has both the finite and unique termination properties, in which case the rules are said to be *convergent*, then the terminal reduction relation actually defines a total function over the set of expressions.

Since our purpose is only to show the existence of certain functions, we will not be concerned with general problems of rewrite rule systems. As a result, we define a restricted form of the term rewriting process sufficient to meet our needs. This will simplify our proofs somewhat.

Definition 4.7: An expression $f(e_1,...,e_n)$ is *eligible* (for rewriting) if $e_1,...,e_n$ are explicit values and f is not a bottom or constructor function. The *restricted rewriting process* only rewrites eligible subexpressions.

Thus, we restrict our attention to a term rewriting procedure that rewrites an expression from the bottom up. Termination occurs when there are no more eligible subexpressions. This may occur either because the entire expression is an explicit value or because the remaining eligible subexpressions do not satisfy the left-hand sides of any rewrite rules.

Definition 4.8: A partially ordered set (S, >) is *well-founded* if there is no infinite sequence of elements in S such that

$$S_1 > S_2 > S_3 > ...$$

Our immediate goal is to derive a set of rewrite rules and show that they are convergent with respect to the restricted rewriting process. The standard technique for proving termination of rewrite rules is to concoct a *termination function* that maps expressions into a well-founded set, and show that every possible rule application decreases the value of the termination function under the well-founded ordering. By the definition of a well-founded ordering, no infinite sequence of reductions would then be possible, and termination would indeed be assured. We adopt this technique to show termination of our rules.

We will first present a result that gives sufficient conditions for the termination of rewrite rule systems that model our recursive function definitions. For notational simplicity, it is phrased in terms that assume the measured variable always occurs in the first argument position. This assumption is only for the model, where a consistent permutation of arguments can be performed to bring this about.

Definition 4.9: Let RR be a set of rewrite rules over a set of functions F. Let W be a weighting function that maps function symbols from F into natural numbers. The only constraint on W is that W(f) = 0 for bottom and constructor functions and W(f) > 0 for all other functions. RR has the *structure reducing* property with respect to W if the following conditions are met for each rule L -> R in RR. Let $L = f(e_1,...,e_n)$.

For each nonvariable subexpression $R' = f'(e'_1,...,e'_n)$ of R, either:

1) $W(f) > W(f')$
or 2) $f = f'$ and $e_1 = cons_j(y_1,...,y_m)$
and $e'_1 = y_i$ for some i.

Theorem 4.10: Let RR be a set of rewrite rules having the structure reducing property with respect to some weighting function W. Then RR also has the finite termination property for the restricted rewriting process.

Proof: The proof is rather long so we relegate its details to Appendix B. It makes use of the multiset

ordering [Dershowitz 79] to derive a termination function.

4.6.2 Existence, no function parameters

We will now proceed to demonstrate that function definitions with recursion are necessarily well defined if they satify the semantic rules of the language set down earlier. The derived rewrite rules will first be presented. Next we show that these rules have finite termination, and finally, show that they have unique termination. This will establish the existence and totality of defined functions. Afterwards, we will show how these results can be extended to handle the addition of functional arguments.

Our first task is to model axioms and function definitions by rewrite rules. In our language description, we considered each inductively defined object type to have its own bottom object. For our present purposes, though, we consider them to all share a common bottom object, which we call "nil." This simplifies the axioms. Each type does, however, have a distinct constructor function. The use of a common bottom object poses no problem because type checking ensures that its role as an object of different types will be played correctly. We are then left with just nil, true and false as the only built-in constants, or zero-place functions. For the type token, we can use a representation of integer or sequence of integer. It does not matter which, since token objects cannot be manipulated; they may only be compared for equality, which is readily done with any object type.

Table 4-1 shows the rewrite rules derived from both the axioms and defined functions. Some of these are actually axiom schemes. Axioms for the undefined functions are complete enough to rewrite any eligible expression provided type checking rules are satisfied. Cases that cannot arise because of type mismatches are simply not included. The rules have been parameterized for an arbitrary set of constructor and destructor functions. It is assumed that each constructor and its associated destructors belong to a separate type. The right-hand side of Rule 12 should be interpreted as a conjunction of component equality terms. It was written in this manner to take advantage of existing axioms for the if-function.

Nonrecursive function definitions yield a single rewrite rule, equating function body to function call. Recursive function definitions give rise to two rules based on the object passed in the measured variable position, which for notational convenience is assumed to be the first variable. Clearly, function arguments can be consistently permuted without loss of generality to construct a hypothetical set of rewrite rules. The right-hand sides come from the two expressions appearing in the "object" construction of the recursive function definition. Consider, for example, the "join" function shown on page 37. It is modeled by the two rules:

```
join (nil, S1) = S1
            join (apr(S, e), S1) = apr (join(S, S1), e)
```

The next task is to define a suitable weighting function over the function symbols that will satisfy the conditions of Definition 4.9. This is quite easily done.

Definition 4.11: For each defined function f, let the *definition index* of f, written di(f), be j if f was the jth function to be admitted to the theory. Assume the numbering actually begins at four. Extend di(f) to undefined functions by

```
di(f) = 0     for bottom and constructor functions
        di(f) = 1     for destructor functions
        di(f) = 2     for "if"
        di(f) = 3     for "equal"
```

Thus, di(f) > di(g) for any defined function f and undefined function g.

Theorem 4.12: The rules in Table 4-1 have the finite termination property with respect to the restricted rewriting process.

Proof: We wish to show that the rules have the structure reducing property for weighting function di(f). We must prove, then, that the conditions in Definition 4.9 are satisfied for all rewrite rules. Consider separate cases for each rule L -> R.

```
 1. dest_ij(nil) = nil                    i=1,...,NC; j=1,...,nc(i)
 2. dest_ij(cons_i(x_1,...,x_nc(i)) = x_j  i=1,...,NC; j=1,...,nc(i)
 3. if (true, x, y) = x
 4. if (false, x, y) = y
 5. equal (false, false) = true
 6. equal (true, true) = true
 7. equal (false, true) = false
 8. equal (true, false) = false
 9. equal (nil, nil) = true
10. equal (nil, cons_i(x_1,...,x_nc(i))) = false    i=1,...,NC
11. equal (cons_i(x_1,...,x_nc(i)), nil) = false    i=1,...,NC
12. equal (cons_i(x_1,...,x_nc(i)),cons_i(y_1,...,y_nc(i)))
       =  if equal(x_1, y_1)
            then . . .
                  then if equal(x_nc(i), y_nc(i))
                         then true
                         else false
                       fi
                 . . .
            else false
          fi                                        i=1,...,NC
13. nf_i(x_1,...,x_nnp(i)) = nbody_i                i=1,...,NN
14. rf_i(nil,x_2,...,x_rnp(i)) = rbody_i1           i=1,...,NR
15. rf_i(cons_j(y_1,...,y_nc(j)),x_2,...,x_rnp(i)) = rbody_i2
      i=1,...,NR; j = type of measured variable for function i

Where
    nf_i    = nonrecursive function i
    rf_i    = recursive function i
    nnp(i) = no. of parameters for nonrecursive
               function i
    rnp(i) = no. of parameters for recursive function i
    nc(i)  = no. of components for constructor i
    NC     = no. of constructors
    NN     = no. of nonrecursive functions
    NR     = no. of recursive functions
```

Table 4-1: Derived rewrite rules for functions.

Rule 1. $di(dest_{ij}) = 1 > di(nil) = 0$.

Rules 2-4. These are trivial since the right-hand sides are variables.

Rules 5-11. All cases follow from

$$di(equal) = 3 > di(bottom) = 0.$$

Rule 12. For L we have di(equal) = 3. Consider the subexpressions of R. For true and false subexpressions we have di(bottom) = 0. Similarly, for each if-subexpression di(if) = 2. Each equal-subexpression is a recursive call. But $e_1 = cons_i(x_1,...,x_n)$ and $e_1' = x_j$ for some j. Therefore, the second condition of Definition 4.9 is satisfied.

Rule 13. Since the function nf_i is nonrecursive, each function called in its body must have smaller definition index than nf_i. This is by the language rules for function definitions: each call is to a previously defined function. For each subexpression $g(e_1,...,e_n)$ of R, then, we have

$$di(nf_i) > di(g)$$

Rule 14. This is the "base case" of the recursive function rf_i. By the language rules, the body in this case is not allowed to have any recursive calls. Therefore, we have a situation similar to that of Rule 13:

$$di(rf_i) > di(g)$$

holds by the same argument.

Rule 15. For this case, recursive calls are permitted. Each nonrecursive call is dispensed with as in Rule 14. For each recursive call subexpression, we have $rf_i = g$. By the language rules, if the measured variable position of rf_i has argument $cons(y_1,...,y_m)$, each recursive call must have one of the y_i as the actual argument for the same position. This satisfies the second condition of Definition 4.9.

We have shown for all rewrite rules that the conditions are satisfied, which establishes that the rules are structure reducing. By Theorem 4.10, we conclude that the rules have finite termination. Q.E.D.

Having established finite termination, we must now demonstrate unique termination before we can conclude the existence of functions for our definitions. This, however, is quite easy to accomplish in our case. Generally, unique termination must be determined by analyzing pairs of rewrite rules whose left-hand sides can be unified [Knuth 70]. Two expressions A and B are unifiable if there exists a substitution S such that A/S = B/S. If the left-hand sides of two rewrite rules are unifiable, then it might be possible to apply either rule to a given expression and the potential for nonunique termination would exist.

If, however, no unifications are possible among the left-hand sides in the set of rewrite rules, then unique termination is assured [Musser 80]. This is because a given expression, if it can be rewritten at all, can only be rewritten by the application of a single rule. The order of rule application is therefore irrelevant. All of this is conditioned on the prior determination of finite termination.

Theorem 4.13: The rules in Table 4-1 have the unique termination property.

Proof: By Theorem 4.12, finite termination holds, so it is permissible to conclude unique termination if there are no unifications possible among the set of rewrite rules. By inspecting Table 4-1, it is readily decided that no two left-hand sides are unifiable. Every pair of left-hand sides with the same main operator has at least one argument position with differing function calls. Therefore, no substitution can possibly unify them and we conclude that the rules have the unique termination property. (Alternatively, we could prove directly that nonunifiable rules terminate uniquely under the restricted rewriting process. This would be a simple argument based on a complete induction over the structure of expressions.)

Up to this point, we have shown that the process of rewriting expressions until no longer reducible, using our set of rules, defines a total function from the set of expressions into the set of expressions. How does this help us with our original problem? Originally, our task was to show that a valid function definition actually axiomatizes an existing function. However, for every n-tuple in a function's domain, which is nothing more than an n-tuple of explicit values from D, the rewrite rules can be used to compute the value of the function on that n-tuple.

Theorem 4.14: A function definition without functional arguments satisfying all semantic rules actually defines an existing, total function.

Proof: By Theorems 4.12 and 4.13, the rewriting process defines a total function on expressions. Given an explicit value n-tuple $[d_1,...,d_n]$, the value of an n-place function f on this n-tuple can be computed by rewriting $f(d_1,...,d_n)$ until an irreducible expression is obtained, provided the resulting irreducible term is itself an explicit value.

Suppose the rewriting process were to produce an irreducible expression E that is not an explicit value. Let E' be a subexpression of E that is eligible, meaning it has the form $f(e_1,...,e_m)$, where f is not a bottom or constructor function, but the e_i are explicit values. Such an E' must exist for a finite expression E which is not an explicit value. Through an examination of Table 4-1, it can be seen that one of the rules must apply to E'. Any function call whatsoever having explicit value arguments can be reduced by exactly one of the rules, since type checking ensures that only bottom or constructor functions of the proper types are

supplied as arguments. The rules are complete for explicit value arguments satisfying the type constraints. Therefore, E' is reducible, contradicting our assumption about it. Q.E.D.

4.6.3 Existence, with function parameters

The foregoing results have applied to the case of function definitions without any functional parameters. Now we extend them to handle this additional feature. The basic approach is to keep the same structure of rewrite rules, but to create a new interpretation of the function symbols for defined functions. Rather than view function arguments in the same light as data arguments, we prefer to think of them as parameterizing the function symbol itself. This is entirely reasonable, as we may consider a function definition with function arguments to be a definition schema. When explicit function names are supplied as arguments, a function definition having only data arguments can be derived.

Under this interpretation, a function call written in the language as

$$g(f_1,...,f_m,e_1,...,e_n)$$

where $f_1,...,f_m$ are explicit function names and $e_1,...,e_n$ are expressions, is instead considered to have this conceptual form:

$$g(f_1,...,f_m)\ (e_1,...,e_n)$$

The construction $g(f_1,...,f_m)$ represents a single function symbol, derived from the schematic function g; it is associated with the body of g instantiated by $f_1,...,f_m$. Every possible combination of function arguments gives rise to one or two rewrite rules of the kind previously discussed, depending on whether g is recursive. Thus, another level of schematics is added to the rewrite rule system. With this approach, all that is necessary is to further parameterize the rewrite rules for defined functions, find a new weighting function and use it to prove a new version of Theorem 4.12. All the other results will then apply and we will have extended the proofs to cover the existence of functions with function arguments.

1-12. Same as Table 4-1.

13. $nf_i(f_1,...,f_{nnf(i)})(x_1,...,x_{nnd(i)}) = nbody_i(f_1,...,f_{nnf(i)})$ $\quad i=1,...,NN$

14. $rf_i(f_1,...,f_{rnf(i)})(nil,x_2,...,x_{rnd(i)}) = rbody_{i1}(f_1,...,f_{rnf(i)})$ $\quad i=1,...,NR$

15. $rf_i(f_1,...,f_{rnf(i)})(cons_j(y_1,...,y_{nc(j)}),x_2,...,x_{rnd(i)}) = rbody_{i2}(f_1,...,f_{rnf(i)})$

$i=1,...,NR;$ j = type of measured variable for function i

Where

- nf_i = nonrecursive function i
- rf_i = recursive function i
- nnd(i) = no. of data parameters for nonrecursive function i
- nnf(i) = no. of function parameters for nonrecursive function i
- rnd(i) = no. of data parameters for recursive function i
- rnf(i) = no. of function parameters for recursive function i
- nc(i) = no. of components for constructor i
- NC = no. of constructors
- NN = no. of nonrecursive functions
- NR = no. of recursive functions

Table 4-2: Extended rewrite rules for functional arguments.

The new rewrite rules for defined functions are shown in Table 4-2. They merely formalize what has already been said with words. Each rule is a scheme parameterized by some function arguments $f_1,...,f_m$. There

will be one instance for every combination of explicit function names currently defined in the theory and satisfying the function template constraints. To arrive at a new weighting function, we must come up with a quantity that encodes the information contained in our new style function symbol: $g(f_1,...,f_m)$.

Definition 4.15: Let DI be the maximum definition index, which corresponds to the last function added. Now define the *extended definition index* function, edi(f), as follows.

$$edi(g(f_1,...,f_m)) = h(0) + ... + h(DI)$$

where h(j) is given by:

$$h(j) = 2^j \text{ if any f in } \{g,f_1,...,f_m\} \text{ has di(f) = j}$$
$$= 0 \text{ otherwise}$$

Thus, $edi(g(f_1,...,f_m))$ has a power of two term for each distinct function name among $\{g,f_1,...,f_m\}$. If g has no function arguments, then $edi(g) = 2^{di(g)}$.

We are now in a position to prove a revised version of the termination theorem. We first need to point out a simple but important fact.

Lemma 4.16: A defined function g could never be passed its own name through one of its function arguments $f_1,...,f_m$.

Proof: The truth of this statement is nearly immediate from the definition of function template and the rules for valid function calls. Recall that the function template effectively defines a tree structure. The function template for g would necessarily contain the templates for $f_1,...,f_m$ as proper subtrees. Since template matching is enforced on all function calls, g's header could never match the template of any of its formal function parameters, no matter how indirectly it may have been passed.

Theorem 4.17: The rewrite rules of Table 4-2 have the structure reducing property with respect to the weighting function edi of Definition 4.15.

Proof: Let the left-hand sides of the new rewrite rules be denoted by

$$L = g(f_1,...,f_m)\ (e_1,...,e_n)$$

We must consider every subexpression of the right-hand sides. Assume they have the form

$$R' = g'(f'_1,...,f'_{m'})\ (e'_1,...,e'_{n'})$$

Rules 1-12. These are the rules for the undefined functions. The proofs for these cases are similar to the previous ones. Simply substitute edi(g) wherever di(g) was called for. Since di(g) > di(g') if and only if $2^{di(g)} > 2^{di(g')}$, all of the original cases imply the new ones.

Rule 13. Every subexpression in this case is a nonrecursive call. Consider those members of $\{g',f'_1,...,f'_{m'}\}$ having definition index greater than di(g). We know that such function names could not be explicitly referred to in the body of g because they would have been undefined at the time g was introduced and consequently in violation of the semantic rules. They must have been passed as actual function arguments to g and so are contained in $\{f_1,...,f_m\}$. This means that any term 2^j, for j > di(g), found in $edi(g'(f'_1,...,f'_{m'}))$ must also be in $edi(g(f_1,...,f_m))$.

Now consider the possibility of g being present in $\{g',f'_1,...,f'_{m'}\}$. It could not have appeared explicitly in the body of g because the call is assumed nonrecursive and because a function may not pass its own name as an actual function parameter. But by Lemma 4.16, it could not have been passed through $\{f_1,...,f_m\}$ either. Therefore, g cannot appear in $\{g',f'_1,...,f'_{m'}\}$ at all and the term $2^{di(g)}$ is missing from $edi(g'(f'_1,...,f'_{m'}))$.

Let Z be the sum of 2^j terms due to $\{g',f'_1,...,f'_{m'}\}$ for j > di(g). We then have

$$edi(g(f_1,...,f_m)) = Z + 2^{di(g)} + X$$
$$edi(g'(f'_1,...,f'_{m'})) = Z + Y$$

for some nonnegative integers X and Y. But Y is the sum of distinct 2^j terms for $j < di(g)$, so $Y < 2^{di(g)}$ and it follows that

$$edi(g(f_1,...,f_m)) > edi(g'(f'_1,...,f'_{m'}))$$

Rule 14. Each subexpression in this case must be a nonrecursive call, which has already been dealt with in Rule 13.

Rule 15. Subexpressions here may be either recursive or nonrecursive. Nonrecursive calls are covered as in Rule 13. For recursive calls, recall that the language rules require all function arguments be passed unchanged. This means

$$g(f_1,...,f_m) = g'(f'_1,...,f'_{m'})$$

But now the rule for valid recursive calls may be invoked. Just as in the previous section,

$$e_1 = cons(y_1,...,y_k) \text{ and } e'_1 = y_i \text{ for some i.}$$

Therefore, the second condition of Definition 4.9 is satisfied.

We conclude that the rules of Table 4-2 have the structure reducing property. Q.E.D.

Theorem 4.18: A function definition, possibly containing function arguments, that satisfies all semantic rules of the language defines an existing, total function.

Proof: To show the existence of a function, we proceed basically as before, but with slight modifications. For a function g having m function arguments and n data arguments, we compute the value of g on m + n tuples from its domain. These tuples will have m explicit function names from F and n explicit data values from D. To evaluate $g(f_1,...,f_m,d_1,...,d_n)$, we use the rewrite rule system to compute $g(f_1,...,f_m)(d_1,...,d_n)$. It is required that the computation terminate finitely and uniquely to an explicit data value in D.

Theorem 4.17 gives us termination. The new rewrite rules are still not unifiable so unique termination also holds as before. Similarly, the argument made for Theorem 4.14 concerning the equivalence of explicit values and irreducible terms is still valid. We appeal to these previously proven results and conclude the existence of functions for our definition scheme.

Chapter 5
AUTOMATED ANALYSIS

A major goal of this research has been to develop a methodology that lends itself to mechanized forms of analysis. A long term objective might be to build a special purpose verification system for the kinds of protocol models we have proposed. For this thesis, however, we have contented ourselves with a demonstration of feasibility. This has been done in two ways. First, several programs have been written to perform some analysis and generation of verification conditions. Second, an existing theorem prover has been enlisted to prove the verification conditions. Application of the methods and tools to some example protocols will be described in the next chapter. The present chapter deals with the tools and techniques of mechanical analysis.

5.1 Decision Table Analysis

The decision table technique for specifying protocol behavior imposes a certain structuring discipline on the protocol modeling process. A designer is forced to associate sets of actions with the conditions under which those actions are to be taken. These conditions are expressed in terms of the possible outcomes of a vector of boolean expression evaluations. A single rule identifies a set of such outcomes, any one of which could cause its selection, which in turn would cause its actions to be performed. The full set of rules characterizes all the outcomes for which actions are to be taken.

There are several well-formedness attributes of a decision table that we might like to ascertain as part of the modeling and analysis process. For instance, we might be interested in knowing whether all possible condition outcomes are covered by the rules of a table. If so, we would say the table is complete, meaning some action will be performed for every eventuality. Other useful properties of a table are readily conceived. Analyzing a table for such attributes would be a helpful first step toward the overall analysis of a protocol model. Errors detected at this stage could save much effort in the later verification stages.

The structured nature of a decision table makes the determination of these properties by mechanical analysis an easy task. Historically, methods and algorithms for analyzing decision tables have received a good deal of study [King 69, Metzner 77]. We draw upon some of this previous work as a basis for what follows.

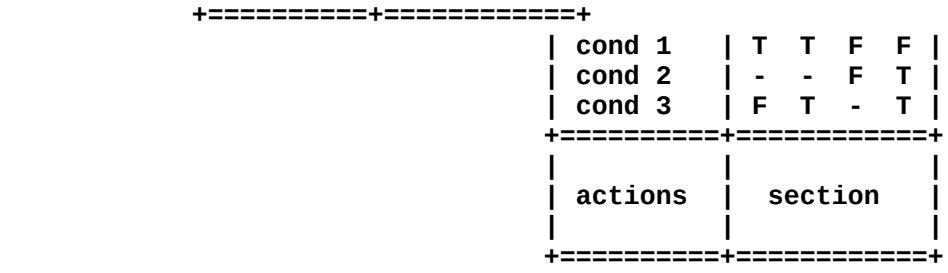


Figure 5-1: Conditions in decision table rules.

Recall the general form of a decision table, partially depicted by example in Fig. 5-1. Analysis of this structure is primarily concerned with the upper half of the table, the part dictating which conditions cause selection of the various rules. The rows are associated with boolean expressions, all of which are evaluated on a common set of variables. Let us refer to these conditions as $C_1,...,C_n$. An evaluation of these conditions is denoted by a boolean vector $[c_1...c_n]$.

Normally, we regard the conditions as being independent, that is, all outcomes (truth value combinations) are possible. It often happens, however, that logical dependencies exist among the conditions. For example, one condition may imply another. Dependencies such as this have the effect of making certain outcomes impossible. When dependencies exist, the analysis can be made more powerful by having the user declare them and taking into account the more restricted set of feasible outcomes that result.

For our purposes, it suffices to accommodate the following four types of logical dependencies.

1. Conditions related by implication. These are pairs of conditions related by C_i -> C_j, C_i -> not C_j, etc.
2. Mutually exclusive conditions. These consist of a set of conditions, at most one of which can be simultaneously true.
3. Exhaustive conditions. Here, at least one condition must be true.
4. Mutually exclusive and exhaustive conditions. Exactly one condition of the set is true.

Given a declaration of relationships among conditions, it is easier to pinpoint potential trouble spots in a table. In addition, the constraints reduce the search space by limiting the number of feasible outcomes. This might enable us to attempt more ambitious forms of analysis, although we prefer to restrict our attention to the more traditional ones. When analyzing a decision table, we wish to answer three basic questions.

1. *Satisfiability.* Are there any rules that can never be selected? In the absence of logical dependencies, all rules are satisfiable because every outcome is feasible. When dependencies exist, though, an ill-formed rule may refer to precisely those outcomes that have been deemed infeasible.
2. *Ambiguity.* Are there any pairs of rules that can be simultaneously eligible for selection? If there are, then a nondeterministic choice is made between them at selection time. If there are no pairwise ambiguities, then the selection process is entirely deterministic. Ambiguity is sometimes referred to as inconsistency.
3. *Completeness.* Do the rules cover all the possible outcomes? If not, then there are situations under which no rule would be selected. In these cases, the default ELSE rule is to take no action whatsoever, meaning that the corresponding event is ignored.

We will shortly sketch a simple algorithm for performing the analysis. It is important to note that the validity of verification does not depend at all on the answer to any of the questions above. The proof methods allow for the possibility of incomplete and ambiguous rules. Analysis is provided mainly as an early feedback path for the designer. Results of the analysis yield information that might indicate the presence of errors, particularly errors of omission. This provides an early warning for the designer before proceeding on to the much more costly verification phase.

Unsatisfiable rules are clearly oversights and should be corrected. Ambiguous or incomplete rules, however, are not necessarily erroneous. In fact, we would expect to see incomplete rules in the normal situation. Often the appropriate response to certain inputs is simply to ignore them. The purpose of the completeness check is to identify those cases under which events are being ignored. The result of an analysis is not simply a yes or no answer; it is a list of relevant conditions constituting the ELSE cases. Thus, it is easy to consult this list and decide whether all the flagged cases are ones that legitimately deserve to be ignored and none are there because of omissions.

The legitimacy of ambiguous rules depends largely on conventions adopted by the designer. Generally speaking, implementation considerations will have more influence than anything else. It may be desirable to

disallow nondeterminism to simplify certain methods of implementation. On the other hand, requiring determinism may result in more verification conditions to prove. An ambiguity may simply represent an overlap on certain outcomes for which the actions of the two different rules turn out to be equivalent. In any event, an analysis tool provides the capability for either enforcing adherence to a convention, or allowing individual judgment on a case by case basis.

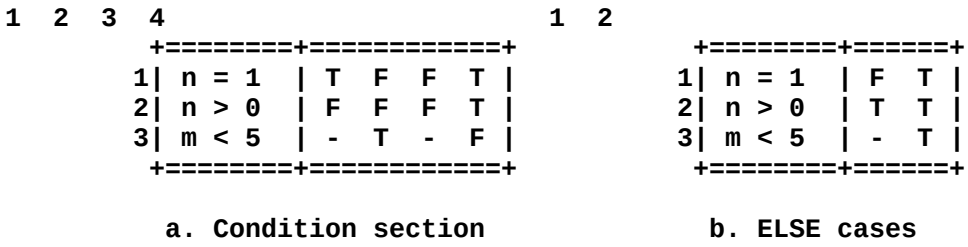


Figure 5-2: Poorly constructed conditions.

Let us consider a small example to illustrate the foregoing concepts. Fig. 5-2(a) shows the condition section of an ill-conceived decision table. Obviously, a logical dependency exists between conditions 1 and 2, namely condition 1 implies condition 2. Hence, rule 1 can never be satisfied. Also, rules 2 and 3 are ambiguous, overlapping on the outcome [F F T]. Finally, the rules are incomplete as well. Part (b) of the figure gives the cases not covered by the rules.

A program to perform the analysis described above has been implemented in Pascal. Since the purpose of this program was to serve as an experimental tool, the algorithm used is straightforward, but not particularly efficient. In fact, it is an exponential algorithm. The basic approach is to perform a brute force enumeration of feasible outcomes for the rules, both individually and collectively. The desired results can then be obtained by comparing these sets. Logical dependency information is taken into account during this procedure. At present, this information is declared by the user and assumed to be true. A realistic system would need to verify that declared dependencies do actually exist.

An outline of the algorithm is given in Fig. 5-3. The algorithm revolves around a recursive function called "gen_feas," which we describe only in brief. This function has two arguments: a boolean condition vector with don't-cares and some logical dependency information. Its result is a set of condition vectors without don't-cares, expanded under the constraints of the logical dependencies. For example,

```
gen_feas([T F T], ld) = {[T F T]}
            gen_feas([T F -], ld) = {[T F F], [T F T]}
```

Any combination not allowed by the dependencies are simply omitted from the result.

In the algorithm, the function gen_feas is called on every rule to obtain the outcomes generated by that rule. The union of these sets is the set of all outcomes accounted for by the rules of the table. The set of all feasible outcomes is found by a call such as gen_feas([---],ld), supplying all don't-cares as the initial condition vector. Comparisons of the various sets yield answers to the questions about satisfiability, ambiguity and completeness.

In the worst case, the gen_feas function visits all nodes of a binary tree. Therefore, it has complexity $O(2^C)$, where C is the number of conditions in a given table. Using a simple algorithm for set intersection, having N^2 complexity, the algorithm of Fig. 5-3 has complexity $O(R^2 2^{2C})$, where R is the number of rules. If this kind of analysis were to be seriously attempted, we obviously would need to find a more efficient algorithm. It should not be a terribly difficult task to discover one. Since the goal of the present research was only to demonstrate feasibility of such a technique, serious algorithm design was hardly justified.

```
procedure analyze (table: decision_table_conditions;
                   ld_info: logical_dependency_info) =
     begin
       var C, account, all_feas, else_cases:
                                 set of condition_vector;
           RC: array [rule_id] of set of condition_vector;

       account := {};
       for i := 1 to nrules(table) do
          RC[i] := gen_feas (c_vect(table, i), ld_info);
          account := account + RC[i];          {union}
          if RC[i] = {} then report "i unsatisfiable" end;
       end;
       for i := 1 to nrules(table) - 1 do
          for j := i + 1 to nrules(table) do
             C := RC[i] * RC[j];               {intersection}
             if C ≠ {}
                then report "i & j ambiguous, have C in common"
             end;
          end;
       end;
       all_feas := gen_feas (don't_cares, ld_info);
       else_cases := all_feas - account;       {set difference}
       if else_cases ≠ {}
          then report "table incomplete, else_cases missing"
       end;
     end;
```

Figure 5-3: Algorithm for decision table analysis.

5.2 Verification Condition Generation

Given a verification condition schema, of the kind developed in Section 3.3, the actual generation of verification conditions is a simple matter. All that is necessary is to supply actual parameters for the schematic VCs and physically emit the resulting formulas. Most of these VCs are associated with decision table rules. VC generation is simply a process of scanning each rule in a table and emitting the VC that the rule gives rise to.

A prototype verification condition generator was implemented as part of the same program containing the decision table analyzer discussed in the previous section. In this way, it was possible to take advantage of some of the results of the analyses. This was primarily limited to the information on ELSE conditions. Recall that an ELSE rule VC has the general form

$$\mathbf{I \ \& \ A \ \& \ not \ (C_{11}\&...\&C_{1n}) \ \& \ ... \ \& \ not \ (C_{m1}\&...\&C_{mn}) \ -> \ I/S}$$

If there are many conditions and many rules, the negated condition terms might be very complex.

During the analysis of a decision table, though, the ELSE conditions that are reported emerge in a somewhat simplified form. This is because of additional information present in the logical dependencies. ELSE condition outcomes are recombined to produce a simplified form with don't-cares introduced wherever possible. Consequently, the ELSE condition terms so generated are much reduced from the general form of the VC schema. It is these terms that are actually used in the final VCs.

The experimental verification condition generator is fairly crude and has no capability to parse expressions of the specification language. It simply manipulates expressions as uninterpreted strings of text. Assertions are represented symbolically by undefined assertion functions. VCs contain calls to these assertion functions with

different substitutions for the variables. In this way, the resulting VCs are still schematic to a certain extent, as the actual assertions are not built into the VCs. If it becomes necessary to change an assertion during the course of verification, as often occurs when we discover that an invariant needs additional terms, then the VCs need not be regenerated.

Before actually using the VCs, they must be translated to a different formalism for use with a mechanical theorem prover. Currently, we are using the Boyer-Moore theorem prover, so the VCs must be translated into formulas of the corresponding theory. Syntactically, such formulas are essentially LISP S-expressions. Some semantic information must also be added since the theory uses untyped data objects. At present, all of this translation is performed by hand. It should not be difficult, though, to construct a mechanical translator.

5.3 Verification Condition Reduction

Verification condition reduction is an attempt to place some human interaction in between the generation and the proof of VCs. If an interactive theorem prover is being used, this interactive capability is already present. The problem is that there is likely to be too much interaction required to complete proofs. If an automatic theorem prover is used, however, a user's problems are not necessarily solved. For nontrivial proofs, he is often faced with the unpleasant choice of either letting the prover grind away endlessly, which can potentially consume a great deal of computational resources, or introducing special purpose lemmas, which is a veiled form of interaction. VC reduction strives to achieve a modest level of interaction by exploiting the special structure inherent in most of the VCs. This interaction takes place after a preliminary step of analyzing an assertion expression to decide how it should be split into cases.

In Section 3.3.1, a schema for the verification conditions of a sequential process was presented. The majority of the VCs resulting from this schema are used to show that the internal invariant is maintained after an event is processed. These VCs all have the general form

$$I \ \& \ PC \rightarrow I/S \tag{3}$$

where I is the invariant assertion, PC represents the path conditions enabling a particular set of actions, and I/S is the invariant instantiated according to the effects of those actions. Most of the work required to carry out proofs is embodied in this type of VC.

Observe that what we have in 3, though, is a logical formula with a special structure. If the invariant assertion I is very simple, then we will be unable to effectively take advantage of that structure. If, however, the invariant is a conjunction of many terms, some of which may themselves be complex, as is typically the case, then a significant potential for exploitation exists. By splitting 3 into subcases, recognizing some as trivially true and generating simpler VCs for the others, we are able to perform a "reduction" of the verification conditions. The actual number of transformed VCs will usually increase; what is reduced is the total amount of work necessary to prove them. In the following, a simple technique for transforming VCs having the form of 3 into a logically sufficient set of VCs will be presented. It is meant to be performed interactively, giving a user the opportunity to remove irrelevant hypotheses from the resulting VCs.

When we say that we will transform a single VC into a set of new ones, what we mean is that the conjunction of transformed VCs is sufficient to imply the original:

$$VC_1 \ \& \ ... \ \& \ VC_n \rightarrow VC$$

Upon obtaining proofs for the $\{VC_i\}$, we could legitimately conclude the truth of the original VC.

Let us begin by rewriting 3 as

$$PC \rightarrow (I \rightarrow I/S) \tag{4}$$

which is a logically equivalent form. Note that in the term I -> I/S, the conclusion matches the hypothesis up to the substitution of variables dictated by S. In particular, the outermost operator of I and I/S will be identical. A good strategy, then, is to split I -> I/S into cases $C_1,...,C_n$ according to the form of I. These cases should satisfy

$$(C_1 \ \& \ ... \ \& \ C_n) \ eqv \ (I \rightarrow I/S)$$

Splitting will be performed if the outermost operator is one of {AND, OR, IMPLIES, IF}.

By generalizing this operation somewhat, we can develop a recursive procedure for case splitting that allows us to remove even nested occurrences of these operators. Suppose we have a formula

$$P \& Q \rightarrow P' \tag{5}$$

where P and P' have the same form, that is, one is a substitution instance of the other. P is called the *primary term* and the terms of Q, which may be a conjunction, are called the *secondary terms*.

Now consider a procedure in which a formula having the structure of 5 is transformed into a set of new formulas, each of which also has the form of 5. The transformation is guided by the outermost operator of P. Splitting may then be done recursively until a set of irreducible formulas is obtained. The resulting formulas will eventually be augmented with the PC terms to generate the transformed VCs:

$$P_i \& Q_i \& PC \rightarrow P_i' \tag{6}$$

The significance of the structure depicted in 6 is that it is schematic. We may generate a case splitting schema that depends only on the particulars of the invariant assertion I. The specific parts of each original VC are the path conditions PC and the variable substitution S. So if we have a schema consisting of several schematic VCs of the form shown in 6, we simply instantiate them with the different PC and S values to generate specific VCs.

As the specific VCs are created, they are handled in one of three ways.

1. If P = P/S, then P = P' and the VC is trivially true and may be dispensed with immediately. This case occurs often because of situations in which S does not change any variable occurring in P.
2. It also happens occasionally that a VC is subsumed by one previously generated. This is likely to happen when many subcases are spawned for terms with few variables. By consulting a list of VCs to be proved, subsumptions can be readily detected and duplicate proof effort can be avoided.
3. The remaining case concerns VCs that actually have to be proved. The user is asked to identify which hypotheses to include in the final version of a VC. They are offered in the order: primary term, secondary terms, and path conditions. It often occurs that a proof is possible using just P -> P' or with only a few additional hypotheses. Deletion or inclusion of hypotheses is the chief means for a user to apply his knowledge of the problem and focus the attention of the theorem prover on relevant information. Further splitting and detection of trivial cases may be performed in conjunction with the hypothesis selection process.

Before actually performing the generation of specific VCs, it is useful to allow a user to add information that may help the reduction process. If a user wishes to add FACT, we add it to the list of hypotheses and generate an additional VC:

$$I \& PC \rightarrow FACT$$

This additional VC ensures that the user's claim is really true, so its use is valid. Where this facility is particularly useful is in cases where the fact is an equality. Substituting one subexpression for another can often generate some additional cases that are trivially reduced to true. The user can help himself considerably by recognizing such situations and pointing them out.

Let us now describe the case splitting process for terms having suitable operators. Assume we have the formula

$$P \& Q \rightarrow P'$$

where either P' = P/S or P = P'/S. This formula is split into several others if P's outermost operator is one of the following.

1. AND. We have the form

$$(A_1 \& \ldots \& A_n) \& Q \rightarrow (A_1' \& \ldots \& A_n')$$

from which n formulas are generated:

$$A_i \& A_1 \& ... \& A_{i-1} \& A_{i+1} \& ... \& A_n \& Q \text{ -> } A_i' \quad i=1,...,n \tag{7}$$

2. OR. Assume we have

$$(A_1 \text{ v } ... \text{ v } A_n) \& Q \text{ -> } (A_1' \text{ v } ... \text{ v } A_n')$$

which is equivalent to a conjunction of n formulas:

$$A_i \& Q \text{ -> } (A_1' \text{ v } ... \text{ v } A_n') \quad i=1,...,n$$

From these we generate

$$A_i \& \sim A_1' \& ... \& \sim A_{i-1}' \& \sim A_{i+1}' \& ... \& \sim A_n' \& Q \text{ -> } A_i' \quad i=1,...,n \tag{8}$$

3. IMPLIES. Given the formula

$$(A \text{ -> } B) \& Q \text{ -> } (A' \text{ -> } B')$$

construct the equivalent form

$$A' \& (A \text{ -> } B) \& Q \text{ -> } B'$$

This may be split into the two cases

```
A' & ~A & Q -> B'
                    A' & B & Q  -> B'
```

which we manipulate to generate:

```
1. A' & ~B' & Q -> A
                  2. B & A' & Q   -> B'                    (9)
```

4. IF. From the formula

$$(\text{if A B C}) \& Q \text{ -> } (\text{if A' B' C'})$$

split into the two cases

```
A' & (if A B C) & Q  -> B'
                ~A' & (if A B C) & Q -> C'
```

which can be further split into four cases:

```
A' & A & B & Q   -> B'
                    A' & ~A & C & Q  -> B'
                    ~A' & A & B & Q  -> C'
                    ~A' & ~A & C & Q -> C'
```

Finally, rearrange these to generate:

```
1. B & A' & A & Q   -> B'
                  2. A' & ~B' & C & Q -> A
                  3. A & ~C' & B & Q  -> A'
                  4. C & ~A' & ~A & Q -> C'                (10)
```

Notice that the generated cases all have the general form of 5. Consequently, each of them will be subject to further splitting.

To illustrate the concept of VC reduction, we present a simple example. Suppose we have the invariant assertion

$$I: x < y \& f(y,z) \& (P(x) \text{ -> } Q(z))$$

on the variables x, y, z. A case splitting schema for this assertion can be generated as follows. Begin with the formula I -> I/S (here Q is implicitly equal to true). Rule 7 can be applied to split the conjunction, yielding

```
1. x < y & f(y,z) & (P(x) -> Q(z)) -> (x < y)/S

      2. f(y,z) & x < y & (P(x) -> Q(z)) -> f(y,z)/S

      3. (P(x) -> Q(z)) & x < y & f(y,z) -> (P(x)/S -> Q(z)/S)
```

The last case can be split further by applying rule 9:

```
3.1 P(x)/S & not Q(z)/S & x < y & f(y,z) -> P(x)

          3.2 Q(z) & P(x)/S & x < y & f(y,z) -> Q(z)/S
```

Now suppose we have two VCs to prove, their path conditions and variable substitutions given by

```
1. PC =  y > 5      S =  (x := x+1)

        2. PC =  y < z      S =  (x := x+1; y := y+1; z := 0)
```

We simply need to apply these to the schema using 6 and check for any trivially true cases. For the first VC, we get:

```
1. x < y & f(y,z) & (P(x) -> Q(z)) & y > 5 -> x+1 < y
       2. f(y,z) & x < y & (P(x) -> Q(z)) & y > 5 -> f(y,z)
       3.1 P(x+1) & not Q(z) & x < y & f(y,z) & y > 5 -> P(x)
       3.2 Q(z) & P(x+1) & x < y & f(y,z) & y > 5 -> Q(z)
```

Note that cases 2 and 3.2 are trivially true and can be forgotten. The other two cases can be subjected to a pruning of hypotheses before being emitted as new VCs. For the second of the original VCs, we have

```
1. x < y & f(y,z) & (P(x) -> Q(z)) & y < z -> x+1 < y+1
       2. f(y,z) & x < y & (P(x) -> Q(z)) & y < z -> f(y+1,0)
       3.1 P(x+1) & not Q(0) & x < y & f(y,z) & y < z -> P(x)
       3.2 Q(z) & P(x+1) & x < y & f(y,z) & y < z -> Q(0)
```

In this case, none are trivially true. Suppose, however, that we had deleted hypotheses from case 3.1 of the previous VC to produce P(x+1) -> P(x). This would then be seen to subsume case 3.1 for the second VC, and it could therefore be dismissed.

A trial implementation of these VC reduction ideas has been performed. It has been implemented as a set of LISP functions that manipulate VCs in their form for input to the Boyer-Moore theorem prover. This form is basically that of LISP S-expressions. Some experience at using this sort of tool will be reported in the next chapter.

5.4 Mechanical Proof

By far the most difficult task of a verification effort is actually proving the verification conditions. When proving verification conditions by hand, a human prover relies heavily on his accumulated knowledge of mathematics and logic to simplify formulas. The proof is typically only carried to a level of detail sufficient to convince another person familiar with the same problem. The proof may be based on reasoning whose level of formality varies, in accordance with prevailing custom. Humans can work through proofs in relatively short periods of time. Humans can also make mistakes.

As verification problems become more complex, the integrity of hand proofs becomes more and more questionable. Verification conditions tend to be large, ugly formulas. They are not what mathematicians would consider elegant theorems. Neither are their proofs elegant. Consequently, mechanical proving should be primarily regarded as a way to achieve confidence in a verification effort. It is a confidence that does not erode as the size of problems increases.

Nevertheless, with current technology, mechanical proofs definitely require more work to carry out. Of

prime importance is the lack of readily available problem domain knowledge that can be brought to bear on a proof attempt when needed. This is one of the most serious impediments to rapid progress [Good 80]. The development of such knowledge takes time, but will eventually pay off. In the meantime, verification will continue to appear inordinately expensive, as the evolution of problem domain theories is lumped in with problem specific proofs. In our work on protocol verification, we have attempted to separate these two activities.

As part of this research, we have experimented with the Boyer-Moore theorem prover [Boyer 79]. This prover uses a quantifier-free logic based on the definition of recursive functions and proofs by induction. Data objects in this theory are untyped, but there is a way to axiomatize inductively constructed objects. The prover has been used extensively on program verification problems of various kinds. It runs as an Interlisp program on DEC 10 and DEC 20 systems. Formulas are expressed in a notation similar to LISP S-expressions. The prover's two most important features for our work are its ability to carry out induction proofs and its ability to search a database of lemmas during a proof. There is no direct interaction with the user during a proof. A user does interact, albeit indirectly, by causing an adequate set of lemmas to be proved beforehand.

If we were to try to prove a set of verification conditions from scratch our chances of success would be very slim indeed. Effective use of the prover requires that we build a theory of our basic concepts that is sufficiently rich to allow proofs to be of manageable size. The strategy employed to achieve this aim is outlined below.

- The basic data types and functions of our specification language are realized by shells and recursive functions in the Boyer-Moore theory. Shells are a form of data typing mechanism in this theory.
- A set of fundamental properties of these functions is postulated and proved. Most of these are properties of the sequence and mapping operators, which are usually proved by induction on the structure of one of the sequence variables. A body of generally useful lemmas is thereby built up.
- Proofs of the verification conditions are attempted without resorting to the use of induction. We would like the prover to rely only on the previously proven lemmas and its simplification heuristics.
- Study of failed proof attempts leads to the conception of additional lemmas which are then proved and added to the knowledge base. This process continues until all of the verification conditions have been proved.

This strategy has been successfully applied in several verification attempts. We will report on those in the next chapter. For the rest of this chapter, we will discuss general techniques for using the theorem prover to solve verification problems that arise from the use of our methods.

5.4.1 Translating expressions

To make use of the theorem prover, it is necessary to map concepts from our specification language into the formal theory used by the prover. This is a fairly easy task and is possible to do by hand. Outwardly, the notations appear quite different, the Boyer-Moore theory having the syntax of S-expressions. Semantically, the main differences have to do with notions of data typing. Another important difference is the use of functional arguments, which the theorem prover does not directly support.

Data objects in the Boyer-Moore theory are not associated with intrinsic data types. Any objects may be passed as arguments to functions. A feature that approaches the conventional notion of data type is called the *shell principle*. This principle allows the introduction of inductively defined objects such as integers and sequences. A shell consists of constructor and bottom functions, and a set of destructor functions. Also included is a recognizer function, which provides a way to test whether an object is a member of a certain shell. A set of axioms that characterize the meaning of the shell functions is automatically added to the current theory when a new shell is introduced.

Several shells are predefined. One is a shell for natural numbers, which we use directly. A shell for lists (sequences) is also available; it follows the notation of the LISP CONS operation. However, in order to have notation similar to that of the specification language, we use our own shell for sequences, which we call "APR."

Summarized below are the equivalent notations for the undefined sequence functions.

```
null            (NULL)
                     S <: e          (APR S E)
                     last(S)         (LST S)
                     nonlast(S)      (NLST S)
```

Other primitive sequence operators are introduced as defined functions. Records are also modeled by shells, each field name being a destructor function.

Recursive and nonrecursive functions may also be defined and added to the theory. Recursive functions are checked for well-definedness by proving that recursive calls decrease some measure function with respect to a well-founded relation. As an example, consider the sequence append operator defined as a recursive function in this theory.

```
(DEFN JOIN
                  (X Y)
                  (IF (SEQP Y)
                      (APR (JOIN X (NLST Y)) (LST Y))
                      X))
```

This function is recursive in its second argument, Y. All the basic operators of the specification language have been realized by functions of this type. It is also the mechanism for introducing user defined functions. Complicated concepts might not be axiomatizable with a single recursive function definition. In those cases, intermediate functions will need to be introduced.

Lemmas can be proved and stored for future use in the database of the user's current theory. There are several types of lemmas, but the primary one of interest is the REWRITE lemma. This is a theorem that is applied during proofs as a conditional rewrite rule. An example of such a lemma is shown below.

```
(PROVE.LEMMA JOIN.NULL
                     (REWRITE)
                     (IMPLIES (PSEQP S)
                              (EQUAL (JOIN (NULL) S) S)))
```

An application of this lemma would rewrite an instance of the left-hand side of the equality to the corresponding instance of the right-hand side. Before this happens, though, the prover must establish that the hypothesis holds under the chosen variable substitution. The vast majority of lemmas written are of this form. Induction is usually required to prove them.

The Boyer-Moore theory does not directly allow functional arguments to functions. Nonetheless, it is possible to add axioms and undefined functions, thereby giving us the means to approximate such a facility. The basic approach is to follow the LISP-like usage of quoting function names and calling explicit APPLY functions. These may be introduced as undefined functions:

```
(DCL APPLY1 (FN X))
                     (DCL APPLY2 (FN X Y))
```

It is then necessary to add an axiom for each function name that is to be passed as a function argument.

```
(ADD.AXIOM APPLY1.DOM
                      (REWRITE)
                      (EQUAL (APPLY1 (QUOTE DOM) X)
                             (DOM X)))
```

In the bodies of defined functions, functional arguments are applied by calling one of these special functions:

```
(DEFN REDUCE
                  (FN I S)
                  (IF (SEQP S)
                      (APPLY2 FN
                              (REDUCE FN I (NLST S))
```

```
                    I)))
```

This is the method that has been used to realize the functional argument feature of the specification language. Because of our use of undefined functions and user supplied axioms, the theorem prover is unable to guarantee the consistency of our theory. However, if we follow the semantic rules of the language, we are assured of consistency by the proofs of Section 4.6.

5.4.2 Developing a theory

In order to solve any significant verification problem, it is essential to develop a deductive theory of the pertinent problem domains. In our case, we have concentrated on concepts useful for modeling transport protocols. A sizeable body of lemmas has been developed, forming the rudiments of a reuseable theory. Most of the lemmas have been expressed as fairly general properties of the basic functions of the specification language. As a result, they may be applied in different contexts and used in different verification efforts.

The evolution of this theory has admittedly been ad hoc. Initially, there was a set of function definitions and axioms used to realize the specification language constructs. From there, a number of "result type" lemmas were proved. Since the Boyer-Moore theory does not directly support typing, when it is necessary to constrain a variable, we must apply a predicate function to the variable to ensure that the constraints are met. For instance, the function (PMAPP M) characterizes what it means for the object M to be a proper mapping. It is then also necessary to know that the result of applying certain functions is an object that satisfies a given constraint, in other words, the results of some functions are typed. This is expressed by means of a lemma, which may be conditional, as in the following example.

```
(PROVE.LEMMA PMAPP.LOWER
                        (REWRITE)
                        (IMPLIES (PMAPP M)
                                 (PMAPP (LOWER M I))))
```

After proving the result-type lemmas, there were a fair number of more or less obvious properties of the basic functions that were easily proved. These are basically simple algebraic properties of the functions. They are usually expressed as unconditional rewrite rules, as illustrated below.

```
(PROVE.LEMMA SIZE.JOIN
                       (REWRITE)
                       (EQUAL (SIZE (JOIN X Y))
                              (PLUS (SIZE X) (SIZE Y))))
```

In addition to algebraic properties, there were also many simple theorems proved stating broad conditions under which various predicate functions are true. The following lemma is an example of this.

```
(PROVE.LEMMA INITIAL.RANGE
                     (REWRITE)
                     (IMPLIES (INITIAL X Y)
                              (INITIAL (RANGE X) (RANGE Y))))
```

More complex lemmas of this sort could easily be discovered. It was not judged particularly fruitful, though, to pursue this line of development very far.

Perhaps the greatest number of lemmas are those introduced as a result of attempts to prove verification conditions and other lemmas. As the need arises, lemmas for reducing various combinations of functions are added to the theory. Although their development is example driven, most of these lemmas are still widely useful. An attempt is made to keep the statement of these theorems as general as possible so that they might be used again some day. Experience has shown that this is a worthwhile goal. Lemmas of this kind are almost invariably expressed as conditional rewrite rules. An example is shown below.

```
(PROVE.LEMMA IN.DOMAIN.LOWER.LESSP
                   (REWRITE)
                   (IMPLIES (AND (LESSP J I)
```

```
            (PMAPP M)
            (NUMBERP I)
            (NUMBERP J))
      (NOT (IN I (DOMAIN (LOWER M J))))))
```

Such theorems are usually proved by induction and the application of previously proven lemmas. Some lemmas are introduced merely as intermediate results to help prove higher level lemmas. Afterwards, they may be turned off so they are no longer applied by the theorem prover.

Occasionally it is necessary to include problem specific lemmas. This occurs when trying to prove complex VCs. Although it may be possible to get a formula to reduce without such lemmas, it might incur an excessive amount of computation time to do so. In these cases, lemmas are added to rewrite complex combinations of functions under specialized conditions. It is not expected that they will be used very much, so they are turned on and off when needed.

To date, a theory containing over 400 lemmas has been developed. All of them have been proved by the theorem prover; *absolutely none* have been assumed. About 200 of them are left activated for automatic search by the theorem prover. A continuing problem stemming from a large body of lemmas such as this is the ever increasing cost of proving theorems. As the prover's knowledge grows, as reflected by a user's database, it becomes increasingly expensive to prove theorems. This is because all pertinent and active rewrite rules are attempted during the simplification of a formula. It is not known how much larger the knowledge base can grow before this effect becomes a limiting factor.

5.4.3 Proving verification conditions

To prove verification conditions, we rely on our body of previously proven facts. The prover is instructed not to resort to induction while proving VCs. Because VCs are typically large formulas with many hypotheses, attempts to prove them by induction would not meet with much success. If an attempt to prove a VC fails, we examine the transcript and conjure up additional lemmas sufficient to complete its proof. Alternatively, it may happen that some of the inactive lemmas can be called up and pressed into service.

The actual construction of formulas supplied to the theorem prover deserves some attention. Case splitting and hypothesis deletion functions are performed as described in Section 5.3. In addition, it often helps to arrange the terms of a formula in a preferred order. Suppose, for example, that we wish to prove the following conjecture.

```
(IMPLIES (AND (ALL (FAPPLY (QUOTE MPAIRP)
                           (FLATTEN PKT.OUT)))
              (PMAPP QUEUE))
         (ALL (FAPPLY (QUOTE MPAIRP)
    (FLATTEN (APR PKT.OUT QUEUE)))))                    (11)
```

Internally, the theorem prover represents formulas in clausal form. A clause is a list of terms assumed to form a disjunction. Thus, 11 is equivalent to:

```
(OR (NOT (ALL (FAPPLY (QUOTE MPAIRP)
                      (FLATTEN PKT.OUT))))
          (NOT (PMAPP QUEUE))
          (ALL (FAPPLY (QUOTE MPAIRP)
       (FLATTEN (APR PKT.OUT QUEUE)))))                 (12)
```

Notice that the last term of 12 has embedded within it a subexpression involving APR, the sequence constructor. It is the reduction of this term that will cause things to happen. The theorem prover, however, will simply go down the list in order, trying to reduce each term. No simplifications are possible for the first two terms and attempts to apply lemmas to these will be fruitless. A useful tactic, therefore, is to move the last term to the top of the clause so it will be reduced first. This is easily done by using the contrapositive of 11:

```
(IMPLIES (NOT (ALL (FAPPLY (QUOTE MPAIRP)
                           (FLATTEN (APR PKT.OUT QUEUE)))))
```

```
(NOT (AND (ALL (FAPPLY (QUOTE MPAIRP)
                       (FLATTEN PKT.OUT)))
          (PMAPP QUEUE))))                                    (13)
```

The effect of this change is that 13 will be converted to a form identical to 12, except for the order of its terms.

The VC reduction program performs this reordering automatically when it emits VCs. It decides when this is appropriate according to whether a term comes from the conclusion of the original VC. Recall that the conclusion contains the instantiated assertion. It is the instantiated terms, with their embedded subexpressions, that cause reductions to take place. It makes sense to try to reduce those terms before any others.

Chapter 6
TRIAL APPLICATIONS

The preceding chapters have detailed a methodology for analyzing protocols and systems of concurrent processes. In this chapter we report on some attempts to apply the methods and tools to a couple of example protocols. Both are data transport protocols, or more precisely, the data transfer component of such protocols.

The first example is a protocol introduced and investigated by Stenning [Stenning 76]. It represents an abstraction of the data transfer functions present in several existing transport layer protocols. Since it was the first example attempted using the methodology, it did not benefit from some of the later improvements. In fact, it was experience with this example that spawned some of these improvements, most notably the idea of a VC reduction phase.

The second example protocol, which we call NanoTCP, is more complex. It is meant to be a model of the essential parts of the data transfer functions in DoD's Transmission Control Protocol (TCP) [Postel 80]. This effort includes the use of later advances to the methods, tools and deductive theory. To the best of our knowledge, mechanical proofs of the protocol features in this example have never been obtained before.

6.1 Stenning Protocol

Our version of this protocol follows the original by Stenning very closely. The only substantive difference is in the handling of timeouts.

6.1.1 Informal description

The purpose of this protocol is to provide a unidirectional data transfer service between a pair of user processes. A similar, though bidirectional, data transfer function would normally be part of a comprehensive transport protocol. Such a transport protocol would include connection management services as well. However, for the purposes of this example we will restrict our attention to the simple data transfer function.

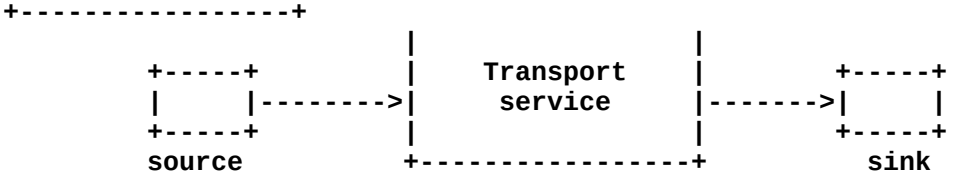


Figure 6-1: Transport service, external view.

We model the simple transport service by a single concurrent process as shown in Fig. 6-1. The process has one input buffer named "source" and one output buffer named "sink." The transport service simply moves messages (in sequence) from the source to the sink.

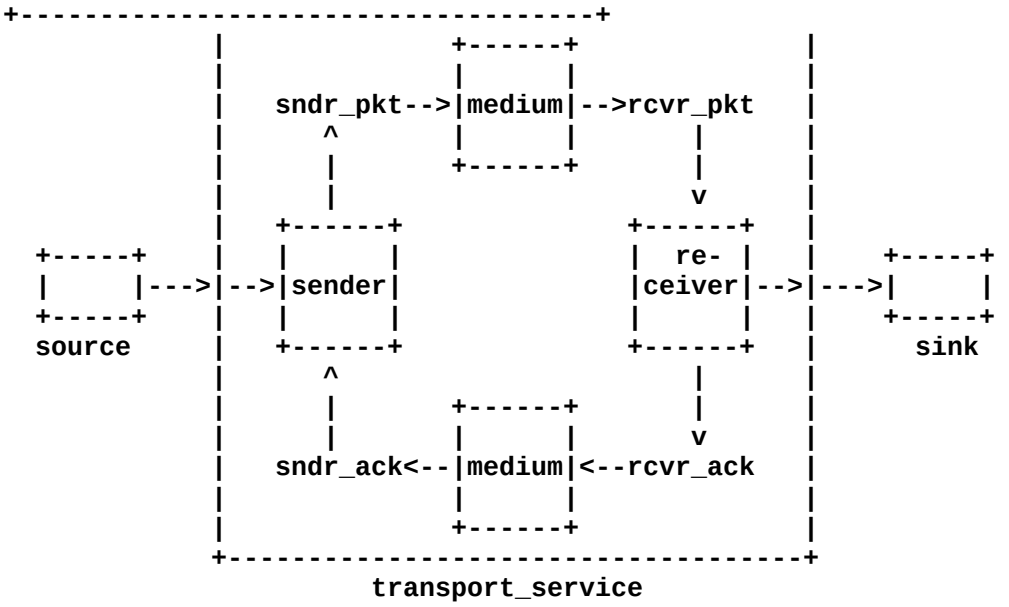


Figure 6-2: Transport service, internal view.

Next we examine the components of this concurrent process. Fig. 6-2 shows that it is made up of a sender, a receiver and two medium processes. The sender and receiver are sequential processes (the transport stations) whereas the transmission media are modeled as (potentially) concurrent processes. For our purposes the medium processes can be considered independent datagram networks, although in practice a single datagram network would be employed. Datagram networks have the property that delivery is unreliable to the extent that packets may be lost, reordered, duplicated or corrupted. It is the responsibility of the transport protocol to overcome these sources of unreliability. We make the usual assumption that packets received with checksum errors are filtered out at some lower level and would appear to be lost by the network. Therefore, the medium processes in our model do not corrupt packets.

There are four internal buffers used to interconnect the subprocesses of the transport service process. The upper path from sender to receiver is for the transmission of packets and the lower path is for returning acknowledgments. A packet contains both a user message and a sequence number. An acknowledgment consists of a sequence number only. Throughout this chapter, sequence numbers are modeled as unbounded natural numbers. During operation, the sender and receiver maintain send and receive windows over the sequence number space. These are used to manage the flow of data.

Very briefly, the processing of events proceeds as follows. When the sender receives a message from the source, it immediately forms a packet out of it, sends the packet toward the receiver and saves it on a retransmission queue. It then updates its internal sequence number variable and starts a timeout if necessary. When a valid acknowledgment arrives, the sender updates its sequence number variables and deletes the portion of the queue which is thereby acknowledged. When a timeout occurs, the entire retransmission queue is sent and a new timeout is started. For simplicity, only a single timeout is set at any given time, rather than one for every outstanding message, as was done in the original protocol.

Upon receipt of an incoming packet, the receiver must distinguish several cases depending on where the packet lies in the sequence space. If the sequence number indicates that it is an old duplicate, then the expected next sequence number is sent back. If the sequence number is not the next expected, but still within the receive window, then it is queued if no copy has already been saved. When the next expected sequence number appears, then as many messages as can legitimately be delivered are sent to the sink. A new value of the next expected sequence number is computed and sent as an acknowledgment, and the queue is updated to delete the messages just delivered.

We will shortly see how this behavior can be specified using our formalism.

6.1.2 Service specification

The transport service is modeled by the concurrent process definition shown in Fig. 6-3. This is the realization of the structure shown in Fig. 6-2.

```
process transport_service (input  source: message;
                                  output sink: message) =
        begin
           buffers (sndr_pkt, rcvr_pkt: packet;
                    sndr_ack, rcvr_ack: natural);
           cobegin
              sender (source, sndr_ack, sndr_pkt);
              receiver (rcvr_pkt, sink, rcvr_ack);
              medium (sndr_pkt, rcvr_pkt);
              medium (rcvr_ack, sndr_ack);
           end
        end
```

Figure 6-3: Transport service process definition.

As far as a formal service specification is concerned, what we would like to be true of this transport service is the usual property about sequenced delivery of messages. More precisely, we require that the output history at the sink is an initial subsequence of the input history at the source. This is expressed quite simply as the external invariant of the transport service process,

sink initial source

where "sink" and "source" are history variable names.

6.1.3 Protocol specification

Before proceeding with the actual protocol specification, let us first characterize the service provided by the medium processes. The medium process may be partially defined by

```
process medium (input  msg_in:  T;
                       output msg_out: T) = pending
```

This definition expresses the fact that we are unconcerned with the internal structure of the medium. All that we care about at this point is that it has an input buffer and an output buffer of some type T.

The medium process needs an external invariant to describe the service of its corresponding protocol layer. Fortunately, this can be provided by a very simple assertion. Since the medium delivers messages with a certain degree of unreliability, the most that can be said about it is that any message appearing at its output must have been received from its input. However, nothing can be said about the ordering of messages or the number of occurrences of a given message that appear at the output. This relationship between histories is expressed by

msg_out follows msg_in

Here "follows" is a relational operator on sequences like initial, except that it is a much weaker relation. The predicate x follows y holds if every element of x appears somewhere in y. If x and y were reduced to sets by removing duplicates, then the follows relation would be equivalent to the subset relation. When applied to sequences, though, follows may hold even if x is larger than y.

Now we will present behavioral specifications for the sender and receiver. Fig. 6-4 contains sequential process definitions for the sender and receiver processes of the Stenning protocol example.

From Fig. 6-4 we see that the sender has a state vector with five components. The interpretation of these variables is as follows.

```
type packet = record (mssg: message;
                                seqno: natural);


           process sender (inputs source: message;
                                   ack_in: natural;
                               output pkt_out: packet) =
           begin
              state vector (unack, next: natural;
                            queue: mapping of packet;
                            timing: boolean;
                            to_time: natural)
                 initially (0, 0, null, false, 0);
              events
                 next - unack < send_window =>
                    on receipt of mess from source
                       handle by source_hdlr;
                 true =>
                    on receipt of ack from ack_in
                       handle by ack_hdlr;
                 timing =>
                    after to_time handle by timeout_hdlr;
              end
           end;


           process receiver (input   pkt_in: packet;
                             outputs sink: message;
                                     ack_out: natural) =
           begin
              state vector (next: natural;
                            queue: mapping of packet)
                 initially (0, null);
              events
                 true => on receipt of pkt from pkt_in
                            handle by pkt_hdlr;
              end
           end;
```

Figure 6-4: Sender and receiver processes.

```
unack   -  sequence number of oldest outstanding message
next    -  next unused sequence number
queue   -  retransmission queue
timing  -  indicates when any messages are outstanding
to_time -  absolute timeout time
```

The quantity (next - unack) is the number of outstanding messages and is used to control whether any new messages are accepted from the source. The send and receive window sizes are given by the constants send_window and rcv_window.

The receiver has two state vector components.

```
next    -  sequence number of next expected message
queue   -  receive queue
```

The receiver is driven entirely by incoming packets; it does not set itself any timeouts. Figures 6-5 and 6-6 show the relationship of the sequence number space and the send and receive windows to the state vector

variables.

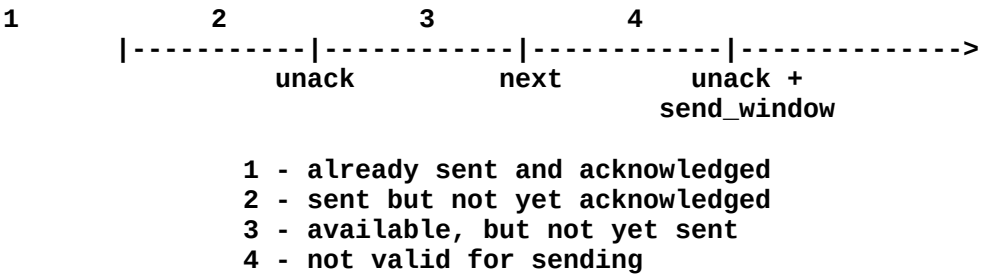


Figure 6-5: Sender's view of sequence number space.

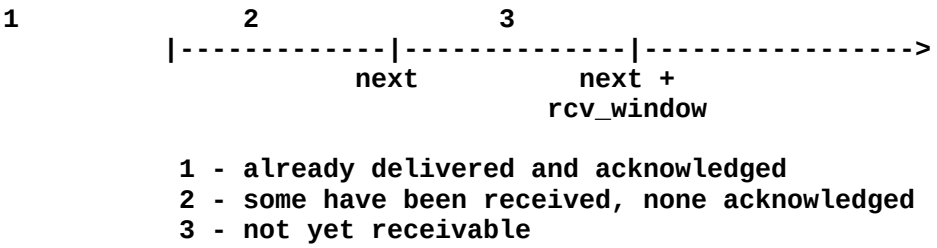


Figure 6-6: Receiver's view of sequence number space.

Event handler specifications for the sender and receiver can be found in Figures 6-7 and 6-8. The expressions in these figures use the mapping operators discussed in Chapter 4. Some of the functions have yet to be described so we take up that task now. A function for extracting a submapping is consec(M). It returns the maximal, initial submapping of M such that all of its domain values are consecutive integers. The next domain value in this sequence (the first missing value) is obtained from the function reach(M).

Event processing is illustrated below with an example from the incoming packet handler of Fig. 6-8. Suppose that the receiver process is currently waiting for message number 6 and has several other messages already in its receive queue.

```
next = 6
queue = [[7,[b,7]], [8,[c,8]], [10,[e,10]]]
domain(queue) = [7, 8, 10]
```

Now suppose that the packet [a,6] arrives through the receiver's pkt_in buffer. This causes rule 4 of the decision table to be selected. Some of the functions that must be evaluated then include:

```
consec(queue) = [[7,[b,7]], [8,[c,8]]]
reach(queue) = 9
upper(queue, 9) = [[10,[e,10]]]
range(consec(queue)) = [[b,7], [c,8]]
apply(".mssg", range(consec(queue))) = [b, c]
```

Thus the response to this incoming packet will be

1. Send [a, b, c] to sink
2. Send [9] to ack_out

and the transition will be

1. Set next to 9
2. Set queue to [[10,[e,10]]]

```
source_hdlr            1 2
               +=====================+=====+
               | timing              | F T |
               +=====================+=====+
               | pkt_out             | A A |
               +---------------------+-----+
               | unack               | - - |
               | next                | B B |
               | queue               | C C |
               | timing              | T - |
               | to_time             | D - |
               +=====================+=====+

               Where
                 A = [[mess, next]]
                 B = next + 1
                 C = queue with ([next] := [mess, next])
                 D = time + delta_t
```

```
  ack_hdlr           1 2
+==================+=====+
| ack > unack      | T T |
| ack = next       | F T |
| timing           | T T |
+==================+=====+
| pkt_out          | - - |
+------------------+-----+
| unack            | A A |
| next             | - - |
| queue            | B C |
| timing           | - F |
| to_time          | - - |
+==================+=====+

Where
  A = ack
  B = upper (queue, ack)
  C = null
```

```
  timeout_hdlr    1
+===============+===+
| ---           |   |
+===============+===+
| pkt_out       | A |
+---------------+---+
| unack         | - |
| next          | - |
| queue         | - |
| timing        | - |
| to_time       | B |
+===============+===+

Where
  A = range (queue)
  B = time + delta_t
```

Figure 6-7: Sender event handlers.

6.1.4 Analysis of specifications

Details of the specification analysis can be found in Appendix C. Both the decision table analysis and VC generation are shown. The results of the decision table analysis are not particularly illuminating, since the tables were so small to begin with. No unsatisfiable or ambiguous rules were detected. Some of the tables are incomplete and their ELSE conditions are displayed. In all cases, they correspond to valid situations in which the corresponding events should be ignored. Therefore, we conclude that the condition portions of the tables capture our intended meaning for the behavioral specification.

The verification conditions are straightforward. The number of VCs for each process is summarized below.

```
sender              9
                        receiver            7
                        transport_service   1
```

```
pkt_hdlr                          1 2 3 4
     +================================+=========+
     | pkt.seqno = next               | F - T T |
     | pkt.seqno > next               | F T - - |
     | pkt.seqno - next < rcv_window  | - T - - |
     | pkt.seqno in domain(queue)     | - F - - |
     | (next+1) in domain(queue)      | - - F T |
     +================================+=========+
     | sink                           | - - B D |
     | ack_out                        | A - C E |
     +--------------------------------+---------+
     | next                           | - - H J |
     | queue                          | - G - K |
     +================================+=========+

     Where
       A = [next]
       B = [pkt.mssg]
       C = [next+1]
       D = pkt.mssg :> apply (".mssg", range (consec (queue)))
       E = [reach(queue)]
       G = queue with ([pkt.seqno] := pkt)
       H = next + 1
       J = reach(queue)
       K = upper(queue, reach(queue))
```

Figure 6-8: Receiver event handler.

Note that sender VC #5 is trivially true. This is because it corresponds to the ELSE rule of the source event handler. Since this table is complete, there are no ELSE conditions. In the VC, this is indicated by the third hypothesis being false.

6.1.5 Process invariants

Next we present the invariants for the sender and receiver processes. The external invariant of the sender process is given by

```
consistent (".seqno", "=", pkt_out)
       & source = apply (".mssg",
                         range (latest (".seqno", pkt_out)))
```

Here a couple of new functions are encountered that need to be explained. The predicate "consistent" is used in this case to express the fact that any two packets in the pkt_out history with the same sequence number field also have the same message field. This fact is needed in the proof of the receiver since it allows the receiver to treat all packets with the same sequence number interchangeably. More generally, if f is a unary function of objects of type T, g is a binary predicate on T, and S is a sequence of T, then consistent(f, g, S) means

for all x, y in S, f(x) = f(y) -> g(x, y)

The function "latest" is used to create a mapping object from a sequence. Let S be a sequence of type T and f be a unary, integer valued function on T. A call on latest(f, S) first applies f to each element e of S to create the domain-range pairs [f(e), e] and then creates a mapping out of these by retaining the rightmost (latest) pair of those with duplicate domain values. For example,

```
pkt_out = [[a,0], [b,1], [a,0], [c,2]]
               latest (".seqno", pkt_out) =
                   [[0,[a,0]], [1,[b,1]], [2,[c,2]]]
```

In this way, we can create a value which is an ordered packet stream with all the duplicate entries removed. By extracting the message fields we recover the original sequence of messages received from the source.

The external invariant for the receiver is somewhat more complicated.

```
consistent (".seqno", "=", pkt_in)
       -> if 0 in domain (latest (".seqno", pkt_in))
            then         sink
                 initial apply (".mssg",
                                range(consec(latest(".seqno",
                                                    pkt_in))))
            else sink = null
          fi
```

The implication reflects the fact that the receiver can only deliver the correct messages if the incoming packets have the consistency property. Its conclusion gives the relation between input and output histories, as was done for the sender's invariant. In this case, the additional complexity results from the fact that an incoming packet may have arrived at a time when its sequence number fell beyond the receive window. The receiver would have ignored it but it still would have appeared in the input history. Hence the messages actually delivered to the sink may lag those that could have been delivered based on purely logical considerations. Observe that in all of these invariants there is no mention of the acknowledgment path from receiver to sender. Due to the unboundedness of sequence numbers in our model of this protocol, sequence numbers are not being reused and hence the proof is independent of the acknowledgment handling.

```
consistent (".seqno", "=", pkt_out)
        & range(queue) follows pkt_out
        & source = apply (".mssg",
                          range (latest (".seqno", pkt_out)))
        & if next = 0
            then   ptk_out = null
                 & queue = null
            else   pkt_out ne null
                 &   last (latest (".seqno", pkt_out)).dom
                   = next - 1
                 &   highest (apply (".seqno", pkt_out))
                   = next - 1
                 & (   queue ne null
                    -> last(queue).dom = next - 1)
          fi
```

Figure 6-9: Sender's internal invariant.

The internal invariants for the sender and receiver are shown in Figures 6-9 and 6-10. Because of their complexity, we make no attempt to annotate the invariants. These assertions are merely typical of what is necessary to make an invariant strong enough to carry an induction (the induction is implicit in the sequential process VCs). Precise relations must be stated between histories and state vector variables.

6.1.6 Proofs of VCs

Given the VCs displayed in Appendix C, we constructed equivalent formulas in the notation accepted by the Boyer-Moore theorem prover. In the process some simple optimization was performed. This included deleting hypotheses and discarding subsumed VCs. Note that these activities would normally be done as part of the VC reduction phase. However, at the time we did this work on the Stenning protocol, the methods and tools for VC reduction had not yet been developed.

Through these tactics, two of the sender's VCs were totally eliminated. This was achieved by taking into account the actual assertion used for the invariant. When this is done, it happens that VC sender#4 is subsumed

```
consistent (".seqno", "=", pkt_in)
      -> (          queue
            follows upper (latest (".seqno", pkt_in), next + 1)
          & next le reach (latest (".seqno", pkt_in))
          & if 0 in domain (latest (".seqno", pkt_in))
              then   next > 0
                   &   sink
                     = apply (".mssg",
                              range (lower (latest (".seqno",
                                                    pkt_in),
                                            next - 1)))
              else   next = 0
                   & sink = null
            fi )
```

Figure 6-10: Receiver's internal invariant.

by sender#3. These two VCs differ only in the hypothesis for the variable "timing," and the substitutions to the variables "timing" and "to_time." Since neither variable actually appears in the invariant assertion, if the hypothesis in question is deleted, both formulas will be identical after expanding the assertion function. Also, VC sender#8 turns out to be trivially true. This is because the only variable substituted for, ack_in, does not appear in the invariant assertion.

Finally, some simplification of the concurrency VC was performed. This involved equating the histories across internal buffers. The mechanics of this procedure are described in Section 7.3.

After performing these transformations, a revised set of VCs is obtained. These are the actual formulas submitted to the theorem prover. They are expressed in terms of several assertion functions, which have been given definitions as shown earlier. The proofs were carried out with respect to an earlier version of the deductive theory. At the time, some 300 lemmas were in the database, with about 200 of them activated for automatic search by the prover.

A summary of the proof execution times is given in Table 6-1. These numbers reflect lumped sums of computation time usage. The time to actually perform a proof is added in with the time taken for I/O and garbage collection. A grand total of 27 minutes was required to do the full proof. Observe that most of that time was consumed by the receiver's VCs. This is consistent with our intuitive understanding of this and similar protocols. It is the receiver that has the more complex processing because it must straighten out the disrupting effects of the unreliable datagram network.

The actual proof transcripts are too voluminous to reproduce in full. Instead, a few representative proofs have been selected and displayed in Appendix C. Complete transcripts can be found in a separate report [DiVito 82a].

6.2 NanoTCP Protocol

TCP is a transport protocol of considerable importance, having been adopted as a standard for use by the Department of Defense. It is also being offered by an ever increasing number of commercial vendors. While the complexity of this protocol precludes our being able to analyze it in its entirety, we are nonetheless able to handle a significant part of it. Our model encompasses the data transfer component of this protocol, with slight modifications. No attempt is made to model connection management features. Perhaps this work can be extended at some point in the future to include those features as well.

VC	Sec	Min
Transport_service	10.0	0.17
Sender #1	2.6	0.04
#2	16.8	0.28
#3	20.7	0.35
#4	23.5	0.39
#5	27.8	0.46
#6	104.4	1.74
Total sender	195.8	3.26
Receiver #1	4.3	0.07
#2	41.7	0.70
#3	184.7	3.08
#4	128.0	2.13
#5	185.5	3.09
#6	404.4	6.74
#7	447.6	7.46
Total receiver	1396.2	23.27
Grand total	1602.0	26.70

Table 6-1: Computation time for Stenning proofs.

6.2.1 Informal description

Like the Stenning protocol, the NanoTCP protocol transfers messages from one user process to another. A message consists of a sequence of octets (bytes). Messages delivered to the receiving user, however, are not necessarily identical to those offered at the originating end. It is the *stream* of octets at the output that must be the same as the stream of octets at the input. In the course of delivering messages, the protocol modules may rearrange the message boundaries, but the ordered sequence of bytes remains intact.

In addition, the receiving user is given the ability to control the flow of messages. This is accomplished by letting it explicitly issue credits for the number of octets it is prepared to accept. At any given time, the receiving user may offer a credit value N, which has the meaning that N additional octets are authorized for delivery. In the same spirit, the sending user is given receipts to indicate when octets have been successfully delivered. A receipt consists of a number N, meaning that N additional bytes in the stream have been absorbed. The protocol service is depicted in Fig. 6-11.

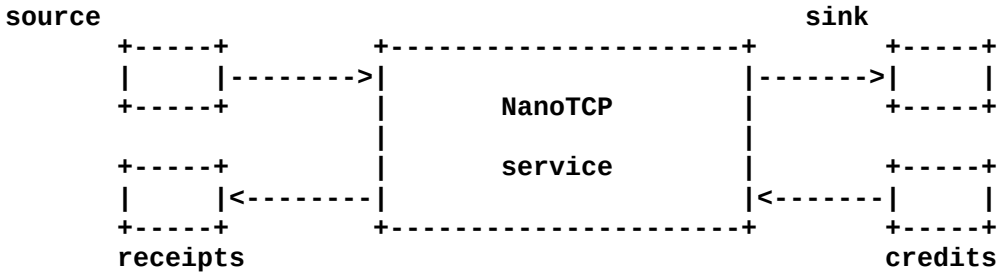


Figure 6-11: NanoTCP protocol service.

Internally, the protocol is realized by a sender module and a receiver module that communicate through a lower level datagram protocol service. The datagram service is modeled in the same way that it was for the

Stenning protocol, including its separation into two distinct processes. The overall structure is also similar: packets are transmitted in one direction and acknowledgments are returned in the opposite direction. Fig. 6-12 shows this internal structure.

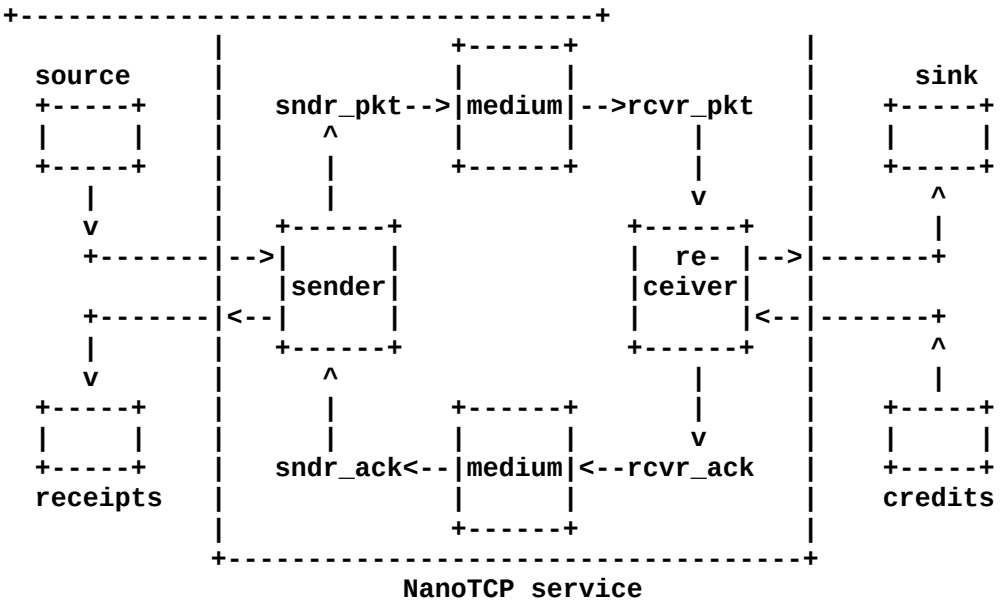


Figure 6-12: Internal structure of NanoTCP.

In the figure, packets flow along the top path from sender to receiver, and acknowledgments flow along the bottom path from receiver to sender. In the actual protocol, a packet contains a sequence number and a sequence of octets. The sequence number corresponds to the first octet and subsequent numbers are assumed to be associated with the remaining octets. In our protocol model, we find it more convenient to use a mapping representation for packets. A packet will have an actual sequence number for each data octet. As long as these numbers are consecutive integers, the two representations are conceptually equivalent. An acknowledgment consists of two numbers: an acknowledging sequence number and a value giving the right hand edge of the receive window. As before, all numbers are being modeled by unbounded integers.

The protocol achieves delivery of octets using a conventional positive acknowledgment plus retransmission scheme, as does the Stenning protocol. Both sender and receiver maintain windows to demarcate the portion of sequence number space that is currently being used. The receiver's window is controlled by the credits it has received and by the acceptance of valid packets. The sender's window is controlled by acknowledgments it gets from the receiver.

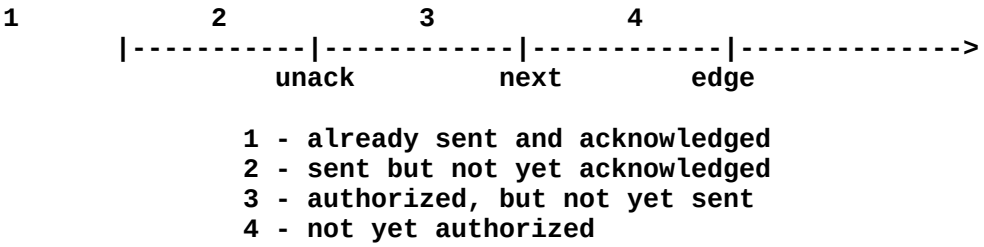


Figure 6-13: Send window.

The sender maintains a retransmission queue of outstanding octets, several sequence number variables and information for timeouts. Its view of the sequence number space is shown in Fig. 6-13. The sender must process three kinds of events.

1. When a new message arrives from the source, its first octet is assigned sequence number "next." A packet is formed and transmitted, and next is updated by the length of this message. The new octets are appended to the retransmission queue. If next is ever advanced past edge, no further messages are accepted from the source until the receiver opens up the window.
2. The handling of an arriving acknowledgment depends on where its acknowledgment number lies with respect to unack. If unack < ack, then some bytes are being acknowledged and unack is updated to reflect that fact. Those bytes are removed from the queue and a receipt is sent to the user. In all cases, edge is updated if the new value exceeds the old.
3. When a timeout occurs, all of the outstanding octets are retransmitted in a single packet. This constitutes repackaging and the receiver is able to handle it. A new timeout is set.

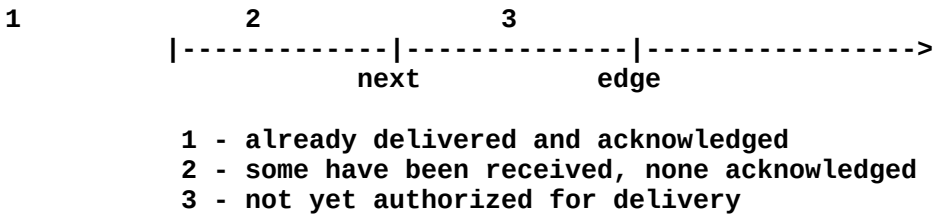


Figure 6-14: Receive window.

The receiver maintains a receive queue and two variables to manage its window. Fig. 6-14 depicts the receiver's view of the sequence number space. There are two kinds of events for the receiver to handle.

a. When a credit is received from the user, the receiver updates its edge variable by the new increment and reports this to the sender by transmitting an acknowledgment with the current value of next and the new value of edge.
b. When an incoming packet arrives, the receiver distinguishes three cases according to where the octets lie in the sequence number space. Refer to Fig. 6-15 for a pictorial interpretation.
    1. The packet contains only old data. Simply send an acknowledgment with the current values of edge and next.
    2. The packet contains new data, but not the next octet expected. Put the portion that falls within the window into the receive queue, possibly overlaying octets already present.
    3. The packet contains new data and the octet for next is present. Disregard any octets in the packet that are outside of the window. Take the remaining good bytes and those already queued and form a message of all bytes that can legitimately be delivered to the sink. Delete these bytes from the queue and update next. Send an acknowledgment to the sender.

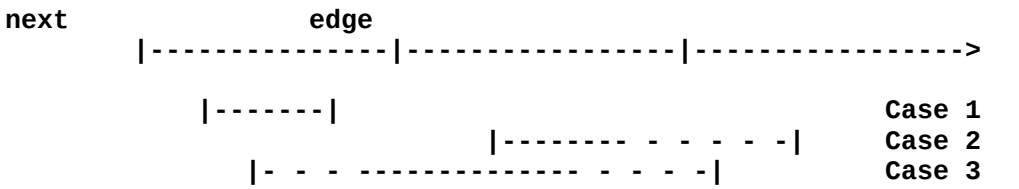


Figure 6-15: Possibilities for incoming packets.

As can be seen, the receiver must be prepared to deal with packets that may overlap in any way. Consequently, messages delivered to the sink will have different boundaries (and may differ in number) from those taken from the source.

6.2.2 Service specification

The NanoTCP service is modeled by the concurrent process definition in Fig. 6-16.

```
process NanoTCP (input  source: message;
                        output receipts: natural;
                        input  credits: natural;
                        output sink: message) =
        begin
           buffers (sndr_pkt, rcvr_pkt: packet;
                    sndr_ack, rcvr_ack: acknowledgment);
           cobegin
              sender (source, sndr_ack, receipts, sndr_pkt);
              receiver (credits, rcvr_pkt, sink, rcvr_ack);
              medium (sndr_pkt, rcvr_pkt);
              medium (rcvr_ack, sndr_ack);
           end
        end
```

Figure 6-16: NanoTCP process definition.

We will give this protocol a formal service specification similar to that of the Stenning protocol. The initial subsequence property is what we want, but it must be applied to the byte streams rather than the message streams. There will be no meaningful correlation between the message sequences. Hence, we give the following external invariant to the NanoTCP process:

flatten(sink) initial flatten(source)

The function "flatten" takes an object having type

sequence of sequence of T

and produces a result having type sequence of T. It flattens out its argument by concatenating its elements, which are themselves sequences. In this way, the assertion relates byte streams rather than message streams.

There certainly are other interesting properties about this protocol that one might like to establish. For instance, it would be nice to show that the receipts never exceed the credits, or that the receipts never exceed the number of octets offered for delivery. Such statements could easily be formulated as separate assertions or appended to the one above. For this example, though, we have limited ourselves to the more important property of data delivery integrity.

6.2.3 Protocol specification

The medium processes in this example are identical to those for the Stenning protocol example. We assume the same service specification for these processes as before.

The behavioral specification for the NanoTCP modules is presented in Figures 6-17, 6-18 and 6-19. The sequence number variables have the interpretation ascribed to them in Figures 6-13 and 6-14. The other variables are analogous to their counterparts in the previous example.

Several new functions have been used in these event handlers. The function enmap(S,k) encodes a sequence of objects into a mapping, as illustrated below.

$$\text{enmap}([e_1,...,e_n],k) = [[k,e_1],[k+1,e_2],...,[k+n-1,e_n]]$$

It is used to attach sequence numbers to the octets of a message obtained from the source. The function union(M1,M2) merges two mappings together into a single mapping. If there are any duplicate pairs, the ones from M2 are used in the result. This function has been used to merge octets and their sequence numbers with

```
type message = sequence of octet;

        type packet = mapping of octet;

        type acknowledgment = record (ackno, edge: natural);


        process sender (inputs  source: message;
                                ack_in: acknowledgment;
                       outputs receipts: natural;
                                pkt_out: packet) =
        begin
           state vector (unack, next, edge: natural;
                         queue: mapping of octet;
                         timing: boolean;
                         to_time: natural)
              initially (0, 0, 0, null, false, 0);
           events
             next le edge =>
                 on receipt of mess from source
                    handle by source_hdlr;
              true =>
                 on receipt of ack from ack_in
                    handle by ack_hdlr;
              timing =>
                 after to_time handle by timeout_hdlr;
           end
        end;


        process receiver (inputs  credits: natural;
                                  pkt_in: packet;
                         outputs sink: message;
                                  ack_out: acknowledgment) =
        begin
           state vector (next, edge: natural;
                         queue: mapping of octet)
              initially (0, 0, null);
           events
              true => on receipt of cred from credits
                         handle by cred_hdlr;
              true => on receipt of pkt from pkt_in
                         handle by pkt_hdlr;
           end
        end;
```

Figure 6-17: NanoTCP sender and receiver modules.

the contents of the send or receive queue, with duplicates being automatically eliminated in the process.

```
source_hdlr        1 2
    +==================+=====+
    | timing           | F T |
    | mess = null      | F F |
    +==================+=====+
    | receipts         | - - |
    | pkt_out          | A A |
    +------------------+-----+
    | unack            | - - |
    | next             | B B |
    | edge             | - - |
    | queue            | C C |
    | timing           | T - |
    | to_time          | D - |
    +==================+=====+

    Where
      A = [enmap(mess, next)]
      B = next + size(mess)
      C = queue union enmap(mess,next)
      D = time + delta_t
```

```
  timeout_hdlr     1
+================+===+
| ---            | - |
+================+===+
| receipts       | - |
| pkt_out        | A |
+----------------+---+
| unack          | - |
| next           | - |
| edge           | - |
| queue          | - |
| timing         | - |
| to_time        | B |
+================+===+

  Where
    A = [queue]
    B = time + delta_t
```

```
  ack_hdlr                      1 2 3 4
+============================+=========+
| unack < ack.ackno          | - F T T |
| next = ack.ackno           | - - F T |
| timing                     | F T T T |
+============================+=========+
| receipts                   | - - B B |
| pkt_out                    | - - - - |
+----------------------------+---------+
| unack                      | - - C C |
| next                       | - - - - |
| edge                       | A A A A |
| queue                      | - - D E |
| timing                     | - - - F |
| to_time                    | - - - - |
+============================+=========+

Where
  A = max (edge, ack.edge)
  B = [ack.ackno - unack]
  C = ack.ackno
  D = upper (queue, ack.ackno)
  E = null
```

Figure 6-18: Sender event handlers.

6.2.4 Analysis of specifications

As before, we present the actual details of specification analysis in Appendix D. The results of the decision table analysis are similar to those of the Stenning protocol example. No unsatisfiable or ambiguous rules were detected. ELSE conditions for the incomplete tables are displayed. Once again, we conclude from examining these conditions that no errors of omission were committed.

```
cred_hdlr          1
                    +==================+===+
                    | ---              | - |
                    +==================+===+
                    | sink             | - |
                    | ack_out          | A |
                    +------------------+---+
                    | next             | - |
                    | edge             | B |
                    | queue            | - |
                    +==================+===+

                    Where
                      A = [[next, edge+cred]]
                      B = edge + cred


                  pkt_hdlr                          1 2 3
                +================================+=======+
                | last(pkt).dom < next           | T - F |
                | next < first(pkt).dom          | - T F |
                | pkt = null                     | F F F |
                | next le edge                   | - - T |
                +================================+=======+
                | sink                           | - - C |
                | ack_out                        | A - D |
                +--------------------------------+-------+
                | next                           | - - E |
                | edge                           | - - - |
                | queue                          | - B G |
                +================================+=======+

                Where
                  A = [[next, edge]]
                  B = queue union lower (pkt, edge)
                  C = [range (consec (M))]
                  D = [[reach(M), edge]]
                  E = reach(M)
                  G = upper (queue, reach(M) + 1)
                  M = queue union upper(lower(pkt,edge),next)
```

Figure 6-19: Receiver event handlers.

The counts of generated verification conditions are:

```
sender          11
                          receiver        8
                          NanoTCP         1
```

The VCs themselves are displayed in the appendix.

6.2.5 Process invariants

The external invariant for the sender process is as follows.

```
consistent (".dom", "=", flatten(pkt_out))
     & all (apply (contig, pkt_out))
     &   flatten(source)
     = range (reduce (with2, null, flatten(pkt_out)))                    (14)
```

This assertion requires a bit of explanation. The function "consistent" is used as before, only this time it is keying off the domain values of the sequence "flatten(pkt_out)." Recall that the history variable pkt_out represents a sequence of packets, each packet being a mapping of octets. Applying flatten to this sequence produces a sequence of domain-range pairs, which is unordered and may contain duplicates. The predicate, however, ensures that those duplicate pairs are consistent with respect to their domain values.

The function contig(M) is a predicate meaning that the mapping M has domain values that are consecutive integers. As a result, the second term of 14 states that every element of the pkt_out history is a contiguous mapping. Both of the first two terms of the assertion state attributes of the outgoing packet stream that the receiver will need as preconditions for its invariant.

In Chapter 4, page 40, we presented a definition of the "reduce" function. We now use this function to compute a mapping object from a sequence of domain-range pairs. Recall that the "with" operator is a three place function, returning a mapping object as result:

M with ([i] := e)

We would now like to use a derived with-function, which is binary, so it can be supplied as an argument to reduce. It can be defined as follows.

```
function with2 (M: mapping of ?T;
                           P: pair (?T)): mapping of T =
                M with ([P.dom] := P.rng)
```

Using this function in reduce(with2,null,S) gives us a way to accumulate a mapping value by successively adding in the pairs of S, in left to right order. The use of reduce in this way performs an operation similar to the "latest" function described in Section 6.1.5. Therefore, the third term of 14 states that the byte stream taken from the source is equal to the range values of the "reduced" and "flattened" outgoing packet stream.

The receiver's external invariant is shown below.

```
consistent (".dom", "=", flatten(pkt_in))
        & all (apply (contig, pkt_in))
     -> if 0 in domain (reduce (with2, null, flatten(pkt_in)))
          then         flatten(sink)
               initial range (consec (reduce (with2, null,
                                    flatten(pkt_in))))
          else sink = null
        fi
```

Given that the consistency and contiguity requirements of the incoming packet stream are satisfied, the receiver maintains a relation between the incoming packet stream and the messages delivered to the sink. This relation is that the byte stream sent to the sink is an initial subsequence of the valid octets from the incoming packet stream. We cannot strengthen this relation to equality because of the possibility that the sender may transmit octets that are slightly beyond the right-hand edge of the receive window. With the invariant presented, incoming packets may fall anywhere in the sequence number space.

The sender and receiver internal invariants are shown in Figures 6-20 and 6-21. They are very similar to their counterparts for the Stenning protocol example.

```
all (apply (contig, pkt_out))
         & contig(queue)
         & consistent (".dom", "=", flatten(pkt_out))
         & queue follows flatten(pkt_out)
         &   flatten(source)
           = range (reduce (with2, null, flatten(pkt_out)))
         & highest (domain (flatten(pkt_out))) = next - 1
         & if next = 0
             then   flatten(ptk_out) = null
                  & queue = null
             else   flatten(pkt_out) ne null
                  &   last (reduce (with2, null,
                                    flatten(pkt_out))).dom
                    = next - 1
                  & (   queue ne null
                     -> last(queue).dom = next - 1)
           fi
```

Figure 6-20: Sender's internal invariant.

```
( consistent (".dom", "=", flatten(pkt_in))
        & all (apply (contig, pkt_in)) )
     -> (          queue
           follows upper (reduce (with2, null,
                                  flatten(pkt_in)), next + 1)
         & next le reach (reduce (with2, null,
                                  flatten(pkt_in)))
         & if 0 in domain (flatten(pkt_in))
             then   next > 0
                  &   flatten(sink)
                    = range (lower (reduce (with2, null,
                                            flatten(pkt_in)),
                                    next - 1))
             else   next = 0
                  & sink = null
           fi )
```

Figure 6-21: Receiver's internal invariant.

6.2.6 VC reduction

Unlike the previous example, this trial application benefited from the use of the VC reduction tools and techniques discussed in Chapter 5. We can say without reservation that the addition of this phase to the methodology was very helpful. Earlier attempts to prove the VCs in their original form were stymied by the excessive amounts of computation time required to obtain their proofs. The complexity of the assertions and the growing size of the lemma knowledge base made it increasingly difficult to prove the VCs, most notably those of the receiver process.

VC reduction helped to boil down some of this complexity, yielding a new set of VCs that were more numerous, but simpler. Most significant was the effect of hypothesis deletion. The majority of the simplified VCs required only a few of the potentially available hypotheses. Those that retained most of their hypotheses were aided by lemmas activated only for the duration of individual proofs. This meant these lemmas did not slow down or interfere with other proofs.

The results of the VC reduction process for the NanoTCP example are shown in Table 6-2. The

computation time consumed by the reduction program is somewhat over two minutes for each of the sender and receiver. From this table, it is not actually possible to tell how much of a reduction in proof time was achieved. In the future, we hope to repeat the proof of the Stenning protocol example so that a comparison can be made. It should also be kept in mind that the use of the VC reduction tools does require some effort on the part of the user, so any decrease in proof time must be traded off against the efforts of the user. What we believe, however, is that some form of human intervention is necessary to obtain any proofs at all for difficult problems. The key to success will be finding the right degree of interaction that is both tolerable to a user and sufficient to break up hard proofs into manageable pieces.

	Sender	Receiver
Original VCs	9	6
Total derived VCs	113	89
trivially true	64	41
subsumed	7	20
proved	5	5
generated	37	23

Table 6-2: VC reduction results for NanoTCP.

6.2.7 Proofs of VCs

Upon completion of the VC reduction phase, the derived VCs were submitted to the theorem prover. A few representative proofs are shown in Appendix D. Complete transcripts are presented in a separate report [DiVito 82b]. A summary of the execution times of these proofs is displayed in Table 6-3. These results are aggregated times expressed in terms of the original VC numbers. Derived VCs were grouped according to the original VCs from which they came. The values in the table represent sums of times for these groups.

As can be seen, the total computation time required is on the order of 29 minutes. This is only two minutes more than the proofs of the Stenning protocol example. This gives some indication of the effectiveness of the VC reduction technique. Preliminary proof attempts for this example showed proof times running significantly higher than similar Stenning VC proofs. Indeed, this what we would expect, as some of the NanoTCP proofs are substantially more difficult. The fact that the two proofs consumed comparable resources implies that VC reduction has provided a useful contribution. Obviously, more experimentation is needed before we can claim anything stronger than this.

Observe from the table that nearly half of the proof effort went into proving VC Receiver#7. This VC corresponds to rule 3 of the receiver's pkt_in handler. Although we have encoded it into a single, innocent looking rule, this case contains some very complex processing. It is responsible for extracting the good bytes from the incoming packet, delivering them to the sink and updating the state variables. Because of potentially arbitrary overlaps with the receive window, computing these values is no easy task. It is not at all surprising that proofs for this rule were difficult.

Finally, we would like to comment on the viability of developing and using a reuseable deductive theory. At the time we verified the Stenning protocol example, we had approximately 300 lemmas in our theorem prover knowledge base. In order to verify the NanoTCP example, some 100 new lemmas were added. Had we not had the first 300 lemmas to start with, we would have had to re-derive them, or at least a substantial subset of them (about half of the lemmas used in this proof are from the original set). The prior existence of these lemmas meant we had a significant platform upon which to build our proof. While it could be argued that these two protocols are very similar, and indeed they are, this is not an uncommon situation. There is a great deal of uniformity among different protocols and we should strive to exploit it.

VC	Sec	Min
NanoTCP	101.2	1.69
Sender #1	4.3	0.07
#2	37.8	0.63
#3	127.2	2.12
#4	0.0	0.00
#5	1.1	0.02
#6	0.0	0.00
#7	0.0	0.00
#8	12.1	0.20
#9	0.0	0.00
#10	0.0	0.00
#11	50.4	0.84
Total sender	232.9	3.88
Receiver #1	11.0	0.18
#2	313.8	5.23
#3	0.0	0.00
#4	0.0	0.00
#5	39.2	0.65
#6	72.0	1.20
#7	835.6	13.93
#8	130.9	2.18
Total receiver	1402.4	23.37
Grand total	1736.5	28.94

Table 6-3: Computation time for NanoTCP proofs.

Chapter 7
GENERALIZATION OF MODEL

In Chapters 2 and 3, we presented some basic methods for modeling and verifying protocols and abstract processes. These techniques take advantage of a fairly restricted model of concurrent processing. If these restrictions are tolerable for the problems we wish to solve, then we gain because modeling is facilitated and the power of analysis is enhanced.

Unfortunately, real-life problems are not always that simple. In particular, the demands of connection management features for protocols are such that the basic specification methods we have developed will normally be inadequate. It will, of course, depend on the properties we intend to prove, but those having to do with proper synchronization of virtual connections will be trouble enough. More generally, the difficulty arises from processes whose operation is highly state dependent, where the current value of state depends on the relative order of message receptions. This is precisely the situation in a transport protocol module; the state may be changed either by user commands or by packets received from the other transport station.

The present chapter proposes some extensions to the basic methods that should enable the modeling and verification of connection management features. More importantly, it allows the specification of connection management to be integrated with the specification of data transfer functions. This gives us the capability to verify correct operation of the various services in their proper context, as well as the capability to explore their interaction. This last point is especially important. It is one thing to study the behavior of certain functions in isolation, but it is quite another to study and ascertain their interrelationships. Nowhere are subtle errors more likely to hide than at the interface between major components of a protocol or other concurrent system. In any event, we believe the extensions give us the desired capabilities, at least in principle. The methods have been developed without sacrificing the all-important properties of specification and proof modularity. No real experimentation, however, has yet been performed with these techniques, so it remains to be seen how well they will work in practice.

Basic features for connection management have to do with commands sent from a user process to a protocol module and status messages sent from a protocol module to a user process. Commands would typically be used to open or close a connection, or listen for an incoming connection request. Status messages would be sent to indicate when changes in the state of a connection have been completed. A transport station’s behavior with respect to connection management is usually characterized by means of a finite state machine. The major states control the kind of processing performed for various events.

7.1 Embedded Finite State Machine

As just mentioned, transport stations commonly follow an embedded finite state machine (FSM) as part of their normal operation. An FSM state variable is maintained among the local state variables. The current state is usually consulted on every event to help determine what the appropriate response is. An FSM state variable can easily be represented by a Pascal-like enumeration type variable. Such a variable can only take on one of several distinct values, the constants of which are denoted by user declared identifiers. In our specification language, the type "token" can be used for this purpose.

Recall the decision table technique for specifying event handling. Given an FSM variable of the kind described above, we could construct the condition part of a decision table with rows as shown in Fig. 7-1. This usage is a bit clumsy, though, due primarily to the need for several similar rules when we wish to specify the same processing for more than one state. It would be helpful to have a built-in method for handling FSM related processing.

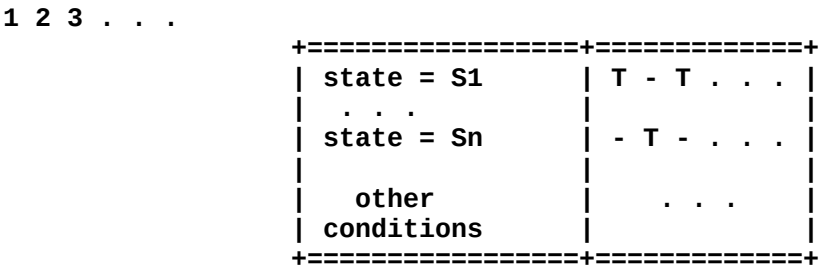


Figure 7-1: Realizing FSM related conditions.

An extension to accommodate a built-in FSM variable is illustrated in Fig. 7-2. The presence of an embedded FSM is signaled when a user declares a state vector component named "state" of type token. Decision tables are augmented by a special condition section placed above the regular conditions. The special state section is used to indicate which states enable a rule for selection. For example, rule 1 in the figure is eligible for selection under the condition

(state = S1 or state = S3) and <cond 1>

This method provides a notationally concise way of specifying complex, state-oriented enabling conditions. When the FSM variable is to be updated, we abbreviate the state names by numbers, as shown in the first transition row of the table.

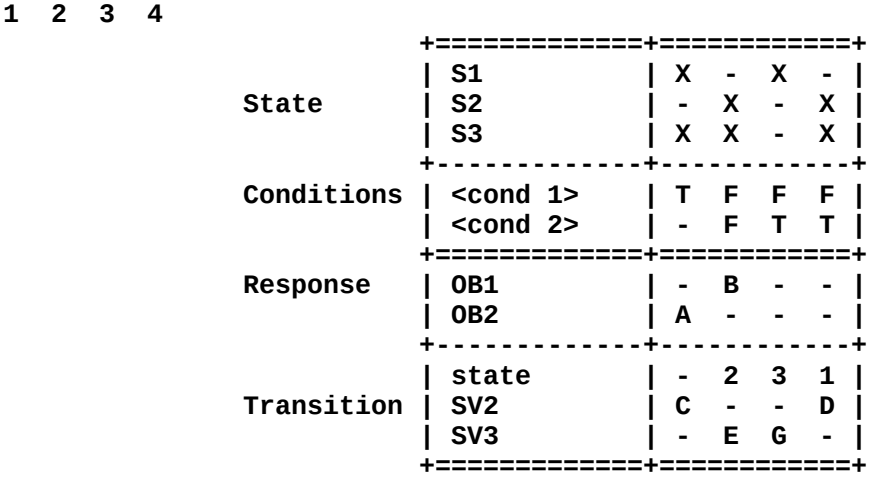


Figure 7-2: Use of built-in FSM.

The extension we have described is obviously not a necessity. Its motivation is practical rather than theoretical. In addition to the notational benefits, it provides a way to increase the power of the decision table analysis discussed in Chapter 5. By exploiting the special relationships existing within the state rows, states being mutually exclusive and exhaustive, the size of the search space is kept small. The prototype analysis program does in fact handle this extended feature in the manner described.

7.2 Extended Histories

An important limitation of the formal specification methods discussed so far is that it is impossible to write an assertion that is sensitive to the relative order of message sends and receives. Consider, for example, the problem of specifying connection management behavior for a transport station. Imagine that the transport station is in its idle state. If it receives a user command to open a connection, it goes into state "open_sent," say. Alternatively, if it receives an incoming packet with an open request, it goes into state "open_received." At any point in the future, then, the current state depends on which of these events happens first, if both should occur within a short interval of time. If the only information available to describe behavior is the message histories, then it is not possible to characterize the current state as a function of these histories. We need something more.

One possibility for extending the model is to provide a single, time-merged history that has the relative ordering information implicit in the message sequence ordering. Such a history would consist of a sequence of pairs $[m_i, b_i]$, where m_i is the ith message to be received (or sent) by a process and b_i is the buffer through which it traveled. This form of history is often called a *trace* [Misra 81], and it has been used extensively in CSP based modules of computation. A problem with this method for use with our specification language is the need to merge together message sequences of dissimilar types.

Another approach, the one we actually propose to use, is based on the use of timestamp information. Histories are augmented to include timestamps, with the resulting auxiliary variables being termed *extended histories*. Gypsy makes use of timestamps and extended histories [Good 79], but for a different purpose. In Gypsy they serve to disambiguate multiple message streams flowing into or out of a single buffer.

The timestamps we will use are logical timestamps and bear no relationship to real time. Each sequential process is assumed to have an additional auxiliary variable, EC, called an *event counter*. This variable does just what its name implies: it counts the number of receive and timeout events that occur within a single process. Each message in an extended history will be associated with an event count value; this value represents the "time" at which a message was sent or received. An event count variable will actually be incremented by two for each event. This allows us to distinguish a receive event from the response that follows it. Thus, if a receive event occurs at time EC, all of its response messages are sent at time EC+1. If messages are sent to different buffers in response to a single event, then it is not possible to discern from the extended histories which was actually sent first; they all appear to happen at the same "time." Event count variables are strictly local and not visible to higher levels of the process hierarchy.

Now we need a representation for extended histories. Gypsy, for instance, uses a sequence of record objects, each record containing a message field and a timestamp field:

sequence of record (message: T; timestamp: integer)

We propose instead to use a representation based on the mapping type. If we use the domain values of a mapping to represent timestamps, we can use the range values to represent messages. Actually, the range values must be sequences of messages because a response is defined to send sequences of messages. Thus, we arrive at the following representation for extended histories:

mapping of sequence of T

The motivation for this choice is the resulting simplicity of proof rules and VCs, as well as the possibility of taking advantage of the theory developed for mapping objects. To obtain the unstamped version of a history, we simply use existing functions to extract the relevant objects:

flatten(range(H))

To express the basic axiom about buffers we need to use

flatten(range(B_in)) initial flatten(range(B_out))

because there is no relationship between the timestamps themselves.

Extended histories give us a method, in principle, to write assertions that capture relative order information. The timestamps of a sequential process are not directly comparable, though, with those of another. Hence, to

describe the relative ordering of messages at opposite ends of a concurrent process, we will need to use information found in the contents of messages as well. This is as it should be, since this is what it takes to physically achieve synchronization anyway. Timestamps only provide information that is available to an observer at a single location; they are not globally meaningful. What gives us some concern is the potentially increased difficulty of doing proofs. Extended histories have introduced more complexity and further experience is needed before we can say whether it will be worth it.

7.3 Process Waiting Behavior

In a sequential process definition, it is possible to attach an enabling condition to each event processing statement. For receive events, this feature is interpreted to mean that if a process is idle, then it is waiting to receive from all those buffers associated with true enabling conditions. Those buffers associated with false enabling conditions are not being waited upon and any messages arriving to these buffers will not be received. These messages will be queued in the buffer for possible reception during later events.

An important consequence of this situation is that if we know a buffer is being waited upon, then we know the buffer must be empty. The significance of this fact is that it allows us to equate the input and output histories of a buffer. Recall that in the proof rule for concurrent processes, we use a term

B_in initial B_out (15)

for every local buffer B. If it is known that a process is waiting on B, then this relationship can be strengthened to B_in = B_out.

For this reason, it is desirable to include information about the waiting behavior of a process in its external invariant. If we had such information, it could be used in the manner described above to simplify concurrency proofs. Therefore, we propose adding a set of boolean auxiliary variables to serve as indicators of process waiting behavior. For each sequential process P, there is an auxiliary variable W_j associated with its input buffer j. W_j is true if and only if P is waiting on input buffer j. We assume that these variables are updated when event processing has completed and the process begins to wait for its next event.

The waiting variables are visible to the next higher level process. Consequently, they may be passed up through the process hierarchy in the same way that history variables are. They may take part in the proof of a concurrent process through the use of the extended buffer axiom

```
if W
                    then B_in = B_out
                    else B_in initial B_out
                fi
```

instead of 15. Normally, the corresponding process must include the waiting indicator in its external invariant for this conditional expression to be of any use. If a process uses the constant true for an enabling condition, then it always waits on the associated buffer and equality of the histories may always be assumed in proofs of its parent process. If extended histories are used, then a more elaborate buffer axiom is needed:

```
if W
      then flatten(range(B_in)) = flatten(range(B_out))
      else flatten(range(B_in)) initial flatten(range(B_out))
   fi
```

Specifying the waiting behavior of processes is not directly relevant to connection management in protocols. Nevertheless, it is an additional technique available for expressing assertions. In general, it will allow us to claim stronger relationships between histories than we might otherwise be able to state. This is potentially useful for any application of the methods.

Chapter 8
CONCLUSION

8.1 Summary of Results

We have presented the development of a methodology for the formal specification and deductive verification of communications protocols and distributed system models. In this methodology, protocol systems are modeled by concurrent processes that communicate via message passing. Techniques are provided for specifying the behavior of protocol modules using a state transition paradigm. Methods for verifying safety properties of processes have been devised. To facilitate the expression of problem domain concepts, a special purpose specification language has also been designed.

The methodology is intended for use with partially automated analysis tools. Most important among these are mechanical theorem provers. Trial applications of the methods have been carried out for a pair of example protocols, with fully mechanical proofs being obtained. Extensions to the basic methods were proposed for specifying more complex kinds of behavior.

We feel that the major contributions of this research are the following:

- A model of concurrent processing based on that of Gypsy was devised. Included is a new technique for specifying the behavior of sequential processes that is largely nonprocedural. The decision table formalism offers concise and effective expression of complex process behaviors.
- Modular verification methods were developed for our process model. These are variations of the Gypsy methods, with suitable modifications being made so they would complement the features of sequential processes. A verification schema was derived, which when properly instantiated, yields the VCs necessary to prove a specific process.
- A new language for expressing both assertions and behavioral specifications was designed. This language emphasizes generic features and a small, fixed set of data structures. The data types and primitive operations have been chosen to facilitate the modeling of protocol processing and the specification of formal properties.
- Techniques for performing certain simple automated tasks to aid analysis were developed. Prototype implementations of simple tools were constructed. A strategy for applying a particular automatic theorem prover, the Boyer-Moore prover, to our verification problems was devised.
- A deductive theory of protocol concepts based on the primitive operations of our specification language has been evolving. A sizeable body of lemmas was postulated and proved using the theorem prover.
- Trial applications of the methodology were carried out on two example protocols. One was the Stenning protocol and the other was a variant of the data transfer portion of TCP. Specifications were written and all the assertions necessary to verify a data delivery property were introduced. Complete proofs were carried out using the theorem prover. The proofs depend only on previously proven lemmas; nothing has been assumed.
- Some extensions to the basic methods for handling connection management and other state

oriented processing were proposed.

8.2 Related Work

Other work related to ours can be found in the area of program verification as well as the area of protocol analysis. There is a considerable body of literature in each of these areas. We briefly compare our research with that of others', concentrating primarily on methods for verifying concurrent processes.

The original work on Gypsy [Good 78, Good 79] is most closely related to ours. Our model of concurrency and modular verification techniques are taken from Gypsy. Gypsy is primarily a methodology for verifying concurrent programs and was not intended for modeling protocols per se. It can be used in this way, however, by using a procedural form of specification. Doing so would make it possible to take advantage of the automated tools provided in the Gypsy verification system.

Some work by Hoare and Chao Chen has been directed at verifying CSP models of processes [ChaoChen 81, Hoare 81]. They have developed two calculi for performing correctness proofs. Their notions of history variables and modular proof are very similar to those of Gypsy and our own. Some properties of the HDLC protocol have been proved using their methods.

Misra and Chandy [Misra 81] have developed verification methods for a model of concurrent processes based on CSP. Their method features hierarchical process construction and modular verification similar to Gypsy. Their auxiliary variables are known as "traces," and are somewhat different from our histories. Proof rules for concurrency have induction built into them so they can handle proofs of network behavior that is circular in nature.

There have been many models and techniques proposed for protocol specification and analysis. Those less closely related to ours include methods based on finite state machines [Danthine 80], "duologues" [Rudin 78], "colloquies" [LeMoli 73], and Petri nets [Postel 76]. As most of this work focuses on algorithmic analysis methods, we will not discuss them any further. Instead, we consider those based on programming languages or similar models in more detail, since these make use of deductive verification techniques.

Stenning performed some of the early work on protocol verification [Stenning 76]. He first analyzed the protocol that we used as our first example, the approach being based on a slightly extended form of Pascal. He applied traditional Hoare-style proof methods without explicitly modeling concurrency. Assertions were expressed as global invariants and there was no attempt to modularize the proof.

Bochmann attempted to construct mixed models comprised of both finite state machine components and programming language components [Bochmann 77]. In this way, it should be possible to verify safety properties using assertion techniques and some liveness properties using state exploration techniques. Tradeoffs along these lines were explored.

Kroghdal used Simula to model service primitives for a link-level protocol [Kroghdal 78]. He then proved that certain properties hold if the primitives are used in the prescribed way. This programming language approach to modeling protocol behavior used "protocol skeletons" as a basic formalism.

Brand and Joyner applied the techniques of symbolic execution to analyze low level protocols [Brand 78]. They used automated tools that were originally developed for use with microcode verification. Because of this, only very simple data types could be used.

More recently, Hailpern and Owicki applied temporal logic to protocol verification [Hailpern 80, Hailpern 81]. Their underlying process model and proofs of safety properties are very similar to ours. They use procedural statements to specify protocol behavior. Temporal logic enables them to verify both safety and liveness properties. They also have analyzed the Stenning protocol.

Schwartz and Melliar-Smith have also used temporal logic in their protocol verification work [Schwartz 81].

Their work differs from Hailpern's in that they do not have history variables. Instead, they specify behavior by means of causal axioms expressed directly in terms of program variables.

A consortium of workers at ISI, led by S. L. Gerhart and C. A. Sunshine, developed a state transition formalism for use with the AFFIRM verification system [Thompson 81]. Our concept of state transition system bears some resemblance to theirs. They use algebraic specification techniques, however, to write axioms giving the effects of transitions. AFFIRM can then be used to prove properties of a specification or prove that one specification is consistent with another. Schwabe has used this work as a starting point for his analyses of connection management protocols [Schwabe 81a, Schwabe 81b]. He has also designed a specification language for writing modular specifications of state transition systems.

Lam and Shankar have developed an analysis technique based on "protocol projections" [Lam 82]. By concentrating on individual protocol functions, they are able to reduce the analysis of a complex protocol to the analysis of a few simpler "image protocols." They give sufficient conditions under which the proof of an image protocol implies the proof of its projection in the context of the original protocol.

8.3 Directions for Future Research

There is no shortage of future problems for us to solve. Several possibilities for continuing and extending our research immediately come to mind.

- Analysis of connection management features needs to be explored in earnest. We have given only vague suggestions of how it might be done. A great deal of experimentation is needed to bring these ideas to life.
- Investigation of liveness properties would be the next major addition to work toward. A desirable goal would be to incorporate temporal logic methods into our methodology [Pnueli 77, Lamport 80]. Much of this could be adapted from existing work [Hailpern 80]. One concern, however, is the lack of theorem proving support for such logical reasoning systems.
- We have not yet considered verification of implementations. Given our style of behavioral specification, we should be able to devise a method for showing that a concrete implementation of a process is consistent with an abstract specification of it.
- Further work on formulation of concepts and proofs of lemmas will always be needed. If we ever hope to routinely analyze complex protocols or systems, then we must put in a lot of leg-work to make this a reality.
- More experience with trial applications is obviously called for. We need to attempt a larger variety of protocols and other applications to expand our practical working knowledge.

Appendix A
Definitions of Primitive Functions

The following is a list of primitive function definitions, including most functions appearing in this thesis. Not included are the integer operators and logical connectives, as these are well understood and would be built into any theorem prover we chose to use. The definitions follow the syntax and semantics described in Chapter 4. Most are concerned with defining generic operations on sequences and mappings. The order of presentation is one that reflects a bottom up introduction of definitions. Note that a few functions are provided only as intermediate steps in the axiomatization of complex concepts.

```
function apl (e : ?T; S : sequence of ?T) : sequence of T =
    object S is
      null   : [e];
      V <: x : apl(e, V) <: x
    end;

function first (S : sequence of ?T) : T =
    object S is
      null   : default(T);
      V <: e : if V = null then e else first(V) fi
    end;

function nonfirst (S : sequence of ?T) : sequence of T =
    object S is
      null   : null;
      V <: e : if V = null then null else nonfirst(V) <: e fi
    end;

function join (X, Y : sequence of ?T) : sequence of T =
    object Y is
      null   : X;
      S <: e : join(X, S) <: e
    end;

function size (S : sequence of ?T) : natural =
    object S is
      null   : 0;
      V <: e : size(V) + 1
    end;

function in (e : ?T; S: sequence of ?T) : boolean =
    object S is
      null   : false;
      V <: x : e = x or in(e, V)
    end;

function follows (X, Y : sequence of ?T) : boolean =
    object X is
      null   : true;
      S <: e : in(e, Y) & follows(S, Y)
    end;

function subseq (X, Y : sequence of ?T) : boolean =
    object Y is
      null   : X = null;
```

```
      S <: e :    subseq(X, S)
                or if X = null
                     then true
                     else   last(X) = e
                          & subseq(nonlast(X), S)
                   fi
    end;

function initial (X, Y : sequence of ?T) : boolean =
    object Y is
      null   : X = null;
      S <: e : X = Y or initial(X, S)
    end;

function with (M : mapping of ?T;
               i : natural; e : ?T) : mapping of T =
    object M is
      null   : [[i, e]];
      S <: p : if i < p.dom
                 then with(S, i, e) <: p
                 else if i = p.dom
                        then S <: [i, e]
                        else M <: [i, e]
                      fi
               fi
    end;

function without (M : mapping of ?T;
                  i : natural) : mapping of T =
    object M is
      null   : null;
      S <: p : if i = p.dom
                 then S
                 else without(S, i) <: p
               fi
    end;

function elt (M : mapping of ?T; i : natural) : T =
    object M is
      null   : default(T);
      S <: p : if i = p.dom then p.rng else elt(S, i) fi
    end;

function domain (M : mapping of ?T) : sequence of natural =
    object M is
      null   : null;
      S <: p : domain(S) <: p.dom
    end;

function range (M : mapping of ?T) : sequence of T =
    object M is
      null   : null;
      S <: p : range(S) <: p.rng
    end;

function lower (M : mapping of ?T;
                i : natural) : mapping of T =
```

```
    object M is
      null   : null;
      S <: p : if i < p.dom then lower(S, i) else M fi
    end;

function upper (M : mapping of ?T;
                i : natural) : mapping of T =
    object M is
      null   : null;
      S <: p : if p.dom < i
                 then null
                 else upper(S, i) <: p
               fi
    end;

function enmap (S : sequence of ?T;
                n : natural) : mapping of T =
    object S is
      null   : null;
      V <: e : enmap(V, n) <: [n + size(V), e]
    end;

function all (S : sequence of boolean) : boolean =
    object S is
      null   : true;
      V <: b : all(V) & b
    end;

function some (S : sequence of boolean) : boolean =
    object S is
      null   : false;
      V <: b : some(V) or b
    end;

function tally (S : sequence of boolean) : natural =
    object S is
      null   : 0;
      V <: b : if b then tally(V) + 1 else tally(V) fi
    end;

function highest (S : sequence of natural) : natural =
    object S is
      null   : 0;
      V <: n : max (highest(V), n)
    end;

function lowest (S : sequence of natural) : natural =
    object S is
      null   : 0;
      V <: n : if V = null then n else min (lowest(V), n) fi
    end;

function reverse (S : sequence of ?T) : sequence of T =
    object S is
      null   : null;
      V <: e : apl(e, reverse(V))
    end;
```

```
function flatten (S : sequence of sequence of ?T) :
                    sequence of T =
    object S is
      null   : null;
      V <: E : join (flatten(V), E)
    end;

function select (S : sequence of ?T;
                 B : sequence of boolean) : sequence of T =
    object B is
      null   : null;
      V <: p : if p
                 then select (nonlast(S), V) <: last(S)
                 else select (nonlast(S), V)
               fi
    end;

function unique (S : sequence of ?T) : boolean =
    object S is
      null   : true;
      V <: e : not in(e, V) & unique(V)
    end;

function disjoint (X, Y : sequence of ?T) : boolean =
    object Y is
      null   : true;
      S <: e : not in(e, X) & disjoint(X, S)
    end;

function lags (X, Y : sequence of ?T; n : natural) : boolean =
    object Y is
      null   : X = null;
      S <: e : (X = Y) or (n > 0 & lags(X, S, n-1))
    end;

function increasing (S : sequence of integer) : boolean =
    object S is
      null   : true;
      V <: n : if V = null
                 then true
                 else last(V) < n & increasing(V)
               fi
    end;

function ordered (S : sequence of integer) : boolean =
    object S is
      null   : true;
      V <: n : if V = null
                 then true
                 else last(V) le n & ordered(V)
               fi
    end;

function intseq (i, j : natural) : sequence of natural =
    object j is
      0     : if i = 0 then [0] else null fi
      n + 1 : if j < i then null else intseq(i, n) <: j fi
```

```
    end;

function condense (S : sequence of ?T) : sequence of T =
    object S is
      null   : null;
      V <: e : if in(e, V)
                 then condense(V)
                 else condense(V) <: e
               fi
    end;

function insert (S : sequence of integer;
                 n : integer) : sequence of integer =
    object S is
      null   : [n];
      V <: i : if n < i
                 then insert(V, n) <: i
                 else S <: n
               fi
    end;

function sort (S : sequence of integer) : sequence of integer =
    object S is
      null   : null;
      V <: n : insert (sort(V), n)
    end;

function apply (f : fun(?T1 : ?T2);
                S : sequence of ?T1) : sequence of T2 =
    object S is
      null   :  null;
      V <: e :  apply(f, V) <: f(e)
    end;

function couple (f : fun(?T1, ?T2 : ?T3);
                 S : sequence of ?T1;
                 T : sequence of ?T2) : sequence of T3 =
    object T is
      null   :  null;
      V <: e :  couple(f, nonlast(S), V) <: f(last(S), e)
    end;

function adjoin (f : fun(?T1, ?T1 : ?T2);
                 S : sequence of ?T1) : sequence of T2 =
    object S is
      null   :  null;
      V <: e :  if V = null
                  then null
                  else adjoin(f, V) <: f(last(V), e)
                fi
    end;

function extend (f : fun(sequence of ?T1, ?T1 : ?T2);
                 S : sequence of ?T1) : sequence of T2 =
   object S is
     null   :  null;
     V <: e :  extend(f, V) <: f(V, e)
```

```
   end;

function reduce (f : fun(?T1, ?T2 : ?T1);
                 I : ?T1; S : sequence of ?T2) : T1 =
    object S is
      null   :  I;
      V <: e :  f(reduce(f, I, V), e)
    end;

function scan (f : fun(?T1, ?T2 : ?T1);
               I : ?T1; S : sequence of ?T2) : sequence of T1 =
    object S is
      null   :  null;
      V <: e :  scan(f, I, V) <: reduce(f, I, S)
    end;

function is_consec (S : sequence of integer) : boolean =
    object S is
      null   : true;
      V <: n : if V = null
                 then true
                 else last(V) + 1 = n & is_consec(V)
               fi
    end;

function consec (M : mapping of ?T) : mapping of T =
    object M is
      null   : null;
      S <: p : if is_consec(domain(M)) then M else consec(S) fi
    end;

function consistent_elt (f : fun(?T1 : ?T2);
                         g : fun(?T1, ?T1 : boolean);
                         S : sequence of ?T1;
                         e: ?T1) : boolean =
    object S is
      null   : true;
      V <: x : if f(x) = f(e)
                 then   g(x, e) & g(e, x)
                      & consistent_elt (f, g, V, e)
                 else consistent_elt (f, g, V, e)
               fi
    end;

function consistent (f : fun(?T1 : ?T2);
                     g : fun(?T1, ?T1 : boolean);
                     S : sequence of ?T1) : boolean =
    object S is
      null   : true;
      V <: e :   consistent_elt (f, g, S, e)
               & consistent (f, g, V)
   end;

function latest (f : fun(?T : natural);
                 S : sequence of ?T) : mapping of T =
    object S is
      null   : null;
```

```
      V <: e : with (latest(f, V), f(e), e)
    end;

function reach (M : mapping of ?T) : natural =
    object M is
      null   : 0;
      S <: p : if is_consec(domain(M))
                 then p.dom + 1
                 else reach(S)
               fi
    end;

function with2 (M : mapping of ?T;
                p : pair(?T) ) : mapping of T =
    with (M, p.dom, p.rng);

function contig (M : mapping of ?T) : boolean =
    is_consec (domain(M));

function union (X, Y : mapping of ?T) : mapping of T =
    reduce (with2, X, Y);
```

Appendix B
Proof of Theorem 4.10

We give a detailed proof of Theorem 4.10, which is a general result used to establish termination of a restricted class of rewrite rule systems. The proof relies heavily on a well-founded ordering investigated by Dershowitz and Manna, known as the multiset ordering [Dershowitz 79]. We state its definition before going into the proof. We also introduce another ordering and show that it is well-founded.

Definition B.1: Let MS(S) denote the set of all finite multisets (bags) over elements of some set S. Assume (S, >) to be a partially ordered set. There is a *multiset ordering* >> on MS(S) induced by the ordering > on S, defined as follows.

M >> M' iff for some multisets X, Y in MS(S),
X is nonempty and a submultiset of M,
M' = (M - X) u Y
and for all y in Y, there exists an x in X
such that x > y.

We refer to X as the removal multiset and Y as the replacement multiset.

In words, this definition says that a multiset M is decreased by removing at least one element from M and replacing it (them) with a finite number (possibly zero) of new elements, each of which is smaller than one of the elements removed. An example using natural numbers is

$$\{1,3,3,4\} >> \{1,3,4,1,2,2\}$$

where X = {3} was removed and Y = {1,2,2} replaced it. Dershowitz and Manna proved that the multiset ordering (MS(S), >>) is well-founded if and only if (S, >) is.

We will also have use for a well-founded ordering on finite sequences. This ordering can be justified by appealing to the multiset ordering.

Definition B.2: Let Q(U) denote the set of all finite sequences over elements of some set U. Assume (U, >) to be a partially ordered set. There is a *sequence ordering* >> on Q(U) induced by the ordering > on U, defined as follows.

S >> S' iff for some sequences A, B, X, Y in Q(U),
X is nonnull, S = A @ X @ B, S' = A @ Y @ B,
and for every element y_i in Y either
1) $x_1 > y_i$
or 2) $x_1 = y_i$
and length(X) > length(Y) - i + 1.

The gist of this definition is that a sequence S is decreased by removing a contiguous subsequence X from S and replacing it by a sequence Y, each of whose elements is either less than x_1 or equal to x_1, but closer to the end of Y than x_1 is to the end of X. Note that Y may actually be longer than X. For example,

$$[4,7,3,1,6] >> [4,5,6,2,7,4,6]$$

because X = [7,3,1] is replaced by Y = [5,6,2,7,4]. All elements of Y are less than 7, except for one occurrence of 7, which is second from the end of Y, whereas 7 is third from the end of X.

Theorem B.3: Let (U, >) be a partially ordered set. If (U, >) is well-founded, then so is the sequence ordering (Q(U), >>) induced by it.

Proof: The method of proof is to introduce a function h that maps sequences into a known well-founded set. The function h is chosen so that if S >> S' under the sequence ordering, h(S) >> h(S') under the known well-ordering. Thus,

$$S_1 >> S_2 >> S_3 >> \ldots$$

could never be true, because if it were,

$$h(S_1) >> h(S_2) >> h(S_3) >> ...$$

would be as well, yielding a contradiction.

Actually, it is convenient to define h as having three sequence arguments, the concatenation of which is understood to form the sequence of interest. The functionality of h is

$$h: U \times U \times U \rightarrow MS(U \times N)$$

where the range of h is the set of multisets over ordered pairs from U and the natural numbers. These pairs will be compared lexicographically. The definition of h is

$$h(X, Y, Z) = \{[x_1,0],...,[x_l,0], [y_1,m],...,[y_i,m-i+1],...[y_m,1], [z_1,0],...,[z_n,0]\}$$

where X, Y, Z are assumed to have lengths l, m, n.

Now suppose S >> S' under the sequence ordering. Thus,

$$S = A \mathbin{@} X \mathbin{@} B \text{ and } S' = A \mathbin{@} Y \mathbin{@} B$$

for some sequences A, B, X, Y. Consider the images of S and S' under h.

$$h(A, X, B) = \{[a_1,0],...,[a_p,0],[x_1,m],..., [x_m,1],[b_1,0],...,[b_q,0]\}$$

$$h(A, Y, B) = \{[a_1,0],...,[a_p,0],[y_1,n],..., [y_n,1],[b_1,0],...,[b_q,0]\}$$

We need to show

$$h(A, X, B) >> h(A, Y, B)$$

under the multiset ordering. The components contributed by A and B are identical in both multisets, so the removal multiset will consist of the X pairs and the replacement multiset will consist of the Y pairs. Those elements y_i of Y that satisfy $x_1 > y_i$ result in

$$[x_1,m] > [y_i,n-i+1].$$

Those elements y_i satisfying the second condition of the sequence ordering have $x_1 = y_i$ and $m > n-i+1$. So this case also results in

$$[x_1,m] > [y_i,n-i+1].$$

Therefore, every element in the replacement multiset is less than some element of the removal multiset, from which we may conclude

$$h(A, X, B) >> h(A, Y, B).$$

Q.E.D.

Clearly, the sequence ordering could have been defined in a more general way. Each element y_j could be matched up with some element x_i, not necessarily the first. The more specialized version, though, will satisfy our immediate needs.

We are now in a position to prove our main result. Recall the definition of the structure reducing property, stated as Definition 4.9 on page 43.

Theorem B.4: (Theorem 4.10) Let RR be a set of rewrite rules having the structure reducing property with respect to some weighting function W. Then RR also has the finite termination property for the restricted rewriting process.

Proof: We will use the weighting function W to derive a termination function TF and then show that

TF is decreased by the application of any rule in RR. The termination function will map expressions into multisets, leading to the use of the multiset ordering. Elements of the multiset will be sequences, to be compared under the sequence ordering. Finally, elements of these sequences will be ordered pairs, compared under the lexicographic ordering. TF is thus defined over:

$$TF: E \rightarrow MS(Q(N \times N))$$

By the well-foundedness of (N, >), each ordering in the chain of orderings above will also be well-founded.

Consider a tree representation for expressions. Each leaf denotes a variable or nullary function. Each nonleaf denotes a function application in which its children are the arguments to the function. Every node is the root of a subtree representing a subexpression. Now suppose $V_1,...,V_n$ are vertices in the path from the root to the jth leaf of some tree T, V_1 being the root and V_n being the leaf. Let P(T,j) denote the path from T's root to its jth leaf, that is, the sequence of vertices $[V_1,...,V_n]$. Let fn(V) denote the main operator symbol of the function call represented by vertex V.

Next define a function that maps paths in the tree representation for expressions into sequences of ordered pairs.

$$\mathbf{SP([V_1,...,V_n]) = [P_1,...,P_n]}$$

$$\mathbf{\text{where } P_i = [W(fn(V_i)),1] \text{ if } V_{i+1} \text{ is in the measured variable position of } fn(V_i)}$$
$$\mathbf{= [W(fn(V_i)),0] \text{ otherwise}}$$

P_n is defined to be $[W(fn(V_n)),0]$. Also, $P_i = [0,0]$ if V_i represents a bottom or constructor function.

Finally, for a tree T, TF(T) is the multiset of all such sequences for the root to leaf paths in T:

$$TF(T) = \{SP(P(T,1)),...,SP(P(T,k))\}$$

where T is assumed to have k leaves.

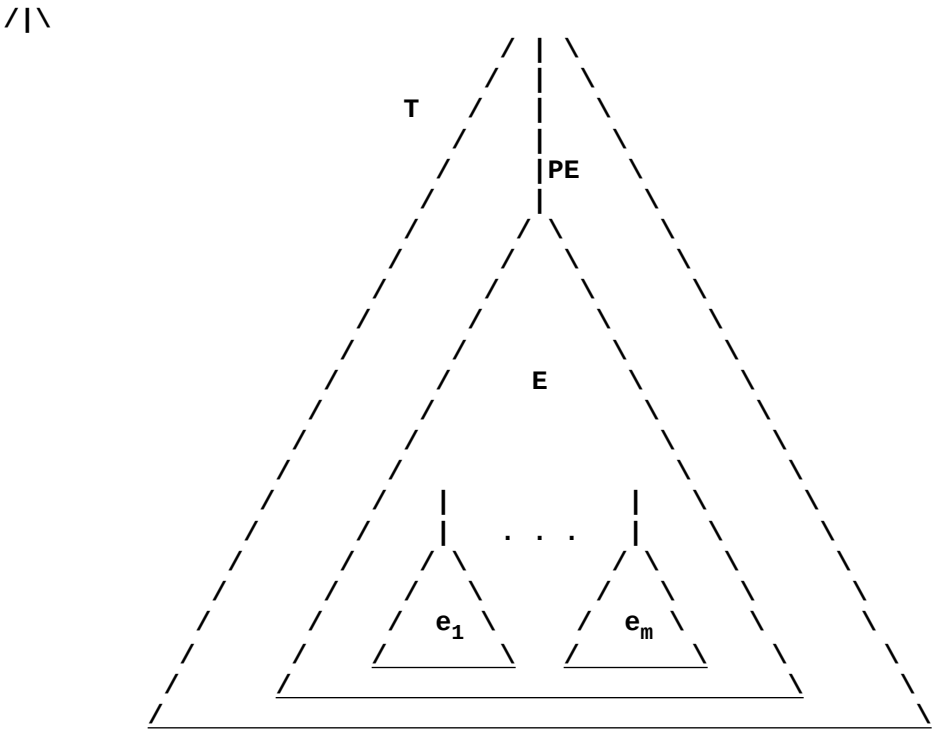
Now consider the effect of rewriting E, an eligible subexpression of T, by applying rule L -> R to E. (Where there is no confusion, we will use names like T to refer to both an expression and the tree that represents it.) The effect is to produce a new expression T' in which everything is identical to T except that subexpression E has been replaced by E'. This situation is depicted in Fig. B-1. In this figure, subtrees are represented by triangles. The expressions $e_1,...,e_m$ are the expressions substituted for variables in L, the left-hand side of the rewrite rule. Those same expressions are substituted for variables in the right-hand side, which are identified as $e'_1,...,e'_m$, in the figure. It is the case that each e'_j is equal to some e_i. In addition, because E is an eligible expression, each e_i is necessarily an explicit value.

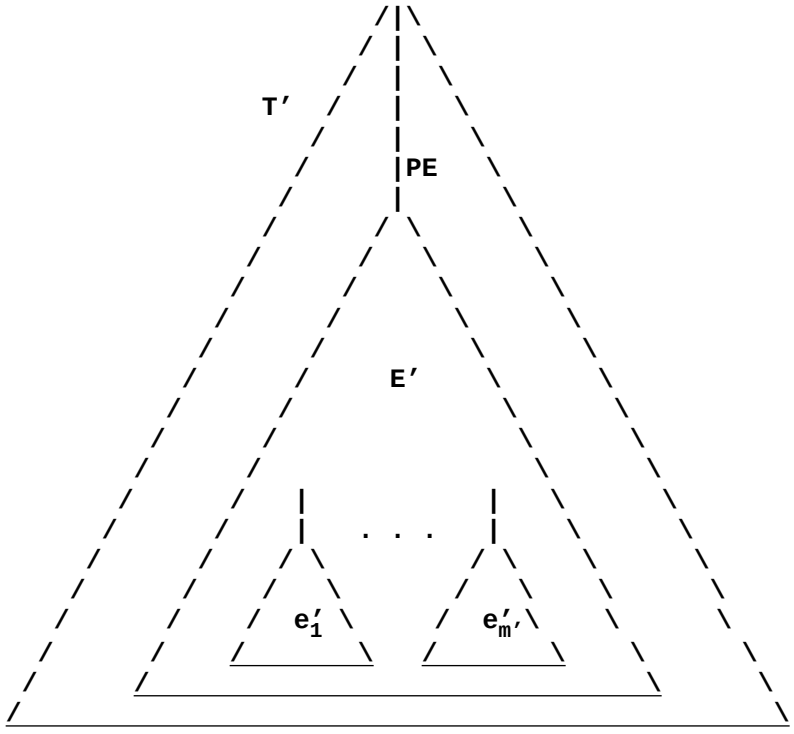
To ensure termination, we must show that TF(T) >> TF(T'). This in turn requires that we exhibit multisets X and Y satisfying the conditions of the multiset ordering. Since only E is being rewritten, all other portions of T will be identical to the corresponding parts of T'. In particular, those paths of T not going through E will match the paths of T' not going through E'. Also, the subpath PE, going from the root of T to the parent node of E, will match the corresponding subpath in T'. Our removal multiset X, then, comes from the paths that go through E. Similarly, the replacement multiset Y is based on the paths through E'. So if we let Z be the multiset contributed by paths not going through E, we have

$$TF(T) = Z \cup X \qquad TF(T') = Z \cup Y$$
$$X = \{SP(PE \mathbin{@} P(E,1)),...,SP(PE \mathbin{@} P(E,q))\}$$
$$Y = \{SP(PE \mathbin{@} P(E',1)),...,SP(PE \mathbin{@} P(E',r))\} \qquad (1)$$

where q and r are the number of leaves in E and E'. X is nonempty because L would have no function calls otherwise. By construction, X is a submultiset of TF(T) and TF(T') = (TF(T) - X) u Y. So all that remains is to show that each element y of Y in 1 is less than some element x of X under the ordering for multiset elements.

Figure B-1: Rewriting E, a subexpression of T.




T′
PE
E′
. . .
e′1
e′m′

These elements are sequences of ordered pairs, corresponding to the root to leaf paths. This means we have to invoke the definition of sequence ordering. Let us divide a path through E or E' into three subpaths:

1. The common subpath PE.
2. The nodes corresponding to function calls introduced in L or R.
3. The subpath going through e_i or e_j', the expression substituted for some variable of L or R.

Any one of these subpaths may be empty, but not all simultaneously.

For each sequence y of Y in 1, we must find a sequence x of X such that x >> y under the sequence ordering. Let P_x and P_y be the paths through E and E' giving rise to sequences x and y. Let V be the root node of E. We consider two cases according to the nodes found in P_y.

Case 1. Suppose P_y does not go through any nodes representing recursive calls in R. In this case, each node V' in subpath 2 of P_y corresponds to a nonrecursive call in R. Consequently, W(fn(V)) > W(fn(V')) for each such V', since the rules of RR are assumed to have the structure reducing property. Choose for P_x any path through E. It then follows that

$$[W(fn(V)),m] > [W(fn(V')),n] \tag{2}$$

for nodes V' in R no matter what the values of m and n are. To show x >> y, we assign as sequence A (in the definition of sequence ordering) the value SP(PE). B is chosen to be null. X is the concatenation resulting from subpaths 2 and 3 of x. Similarly, Y is taken from the concatenation of subpaths 2 and 3 of y. By 2 and the fact that all elements of any subpath 3 are [0,0] (all explicit values), the first condition of the sequence ordering is satisfied for every element in Y. Therefore, x >> y.

Case 2. Suppose P_y goes through one or more nodes representing recursive calls in R. For each node in subpath 2 of P_y corresponding to a nonrecursive call, we conclude that 2 holds as in Case 1. Consider those nodes V' representing recursive calls in R. For these nodes we have fn(V) = fn(V') and consequently W(fn(V)) = W(fn(V')). The second component of the ordered pair [W(fn(V')),m] will be 1 if the successor of V' represents the measured argument of the recursive call; it will be 0 for a non-measured argument. Because the measured argument of a recursive call in R is a variable, only the last recursive call along subpath 2 of P_y can have m = 1.

Case 2.1. Consider the case where P_y passes through none of the measured arguments of the recursive calls in R. In this case, choose P_x to be any path going through fn(V)'s measured argument position. Thus, for each recursive call V' in P_y we have

$$[W(fn(V)),1] > [W(fn(V')),0] \tag{3}$$

and by a construction similar to that of Case 1, we conclude x >> y.

Case 2.2. Alternatively, P_y goes through the measured argument position of its last recursive call V'. Consequently, we have for this node:

$$[W(fn(V)),1] = [W(fn(V')),1]$$

We must pick, therefore, P_x to be a path going through e_i in L, where e_i is the measured argument in R' and cons(e_1,...,e_n) is the measured argument in L. For x and y we have

x = PE @ [[W(fn(V)),1],[0,0]] @ SP(path through e_i)
y = PE @ [D_1,...,D_k,[W(fn(V)),1]] @ SP(path through e_i)

where the D_j represent pairs associated with the nodes above V'. By the second condition of the sequence ordering, we conclude x >> y because each member of D_1,...,D_k satisfies either 2 or 3.

Having shown with the multiset and sequence orderings that TF(T) >> TF(T'), we conclude that the rewrite rule set RR cannot give rise to infinite rewriting. Q.E.D.

Appendix C
Analysis Results: Stenning Protocol

We present a partial listing of the mechanical analysis performed on our Stenning protocol model. Included are the decision table analysis, VC generation and some of the VC proofs. All of this is machine generated output, which we have edited slightly in order to fit more information per page.

C.1 Decision Table Analysis

```
Decision table analysis for process "sender"


Table for event class "source"

No logical dependencies are in effect.


All rules are satisfiable.


No ambiguities have been detected.


Out of a total of    2 possible enabling combinations,    2 are
accounted for by the rules of the table.  Therefore the table
is complete and has no ELSE rule.
```

```
Table for event class "ack_in"

No logical dependencies are in effect.


All rules are satisfiable.


No ambiguities have been detected.


Out of a total of    8 possible enabling combinations,    2 are
accounted for by the rules of the table.  Therefore the table
is incomplete and has the following ELSE rules.

    ELSE conditions
                                    1 2
  +================================+=====+
 1| ack > unack                    | F T |
 2| ack = next                     | - - |
 3| timing                         | - F |
  +================================+=====+
```

Table for event class "timeout"

No logical dependencies are in effect.

All rules are satisfiable.

No ambiguities have been detected.

Out of a total of 1 possible enabling combinations, 1 are accounted for by the rules of the table. Therefore the table is complete and has no ELSE rule.

Decision table analysis for process "receiver"

Table for event class "pkt_in"

The following logical dependencies are in effect:

Conditions 1 and 2 are mutually exclusive.
Condition 1 implies condition 3.

All rules are satisfiable.

No ambiguities have been detected.

Out of a total of 20 possible enabling combinations, 14 are accounted for by the rules of the table. Therefore the table is incomplete and has the following ELSE rules.

ELSE conditions

		1	2
1	pkt.seqno = next	-	-
2	pkt.seqno > next	T	T
3	pkt.seqno - next < rcv_window	F	T
4	pkt.seqno in domain(queue)	-	T
5	(next+1) in domain(queue)	-	-

C.2 Verification Condition Generation

```
Verification conditions for process "sender"

VC 1.  Initialization.

    sender_int (null, null, null, 0, 0, null, false, 0)

_____________________________________________________________________

VC 2.  External invariant.

H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
 -->
C1.  sender_ext (source, ack_in, pkt_out)

_____________________________________________________________________

VC  3.  Event processing.  Event class "source", rule  1.

H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
H2.  next - unack < send_window
H3.  not (timing)
 -->
C1.  sender_int (source <: mess, ack_in,
                 pkt_out @ [[mess, next]],
                 unack, next + 1,
                 queue with ([next] := [mess, next]),
                 true, time + delta_t)

_____________________________________________________________________

VC  4.  Event processing.  Event class "source", rule  2.

H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
H2.  next - unack < send_window
H3.  timing
 -->
C1.  sender_int (source <: mess, ack_in,
                 pkt_out @ [[mess, next]],
                 unack, next + 1,
                 queue with ([next] := [mess, next]),
                 timing, to_time)

_____________________________________________________________________

VC  5.  Event processing.  Event class "source", ELSE rule.
```

```
H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
H2.  next - unack < send_window
H3.  false
 -->
C1.  sender_int (source <: mess, ack_in, pkt_out, unack,
                 next, queue, timing, to_time)
```

```
VC  6.  Event processing.  Event class "ack_in", rule  1.

H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
H2.  true
H3.  ack > unack
H4.  not (ack = next)
H5.  timing
 -->
C1.  sender_int (source, ack_in <: ack, pkt_out, ack,
                 next, upper (queue, ack), timing, to_time)
```

```
VC  7.  Event processing.  Event class "ack_in", rule  2.

H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
H2.  true
H3.  ack > unack
H4.  ack = next
H5.  timing
 -->
C1.  sender_int (source, ack_in <: ack, pkt_out, ack,
                 next, null, false, to_time)
```

```
VC  8.  Event processing.  Event class "ack_in", ELSE rule.

H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
H2.  true
H3.     not (ack > unack)
     or     ack > unack
        and not (timing)
 -->
C1.  sender_int (source, ack_in <: ack, pkt_out, unack,
                 next, queue, timing, to_time)
```

```
VC  9.  Event processing.  Event class "timeout", rule  1.
```

```
H1.  sender_int (source, ack_in, pkt_out, unack, next, queue,
                 timing, to_time)
H2.  timing
 -->
C1.  sender_int (source, ack_in, pkt_out @ range (queue),
                 unack, next, queue, timing, time + delta_t)
```

```
Verification conditions for process "receiver"

VC 1.  Initialization.

    receiver_int (null, null, null, 0, null)
```

```
VC 2.  External invariant.

H1.  receiver_int (pkt_in, sink, ack_out, next, queue)
 -->
C1.  receiver_ext (pkt_in, sink, ack_out)
```

```
VC  3.  Event processing.  Event class "pkt_in", rule  1.

H1.  receiver_int (pkt_in, sink, ack_out, next, queue)
H2.  true
H3.  not (pkt.seqno = next)
H4.  not (pkt.seqno > next)
 -->
C1.  receiver_int (pkt_in <: pkt, sink, ack_out @ [next],
                   next, queue)
```

```
VC  4.  Event processing.  Event class "pkt_in", rule  2.

H1.  receiver_int (pkt_in, sink, ack_out, next, queue)
H2.  true
H3.  pkt.seqno > next
H4.  pkt.seqno - next < rcv_window
H5.  not (pkt.seqno in domain(queue))
 -->
C1.  receiver_int (pkt_in <: pkt, sink, ack_out,
                   next, queue with ([pkt.seqno] := pkt))
```

```
VC  5.  Event processing.  Event class "pkt_in", rule  3.

H1.  receiver_int (pkt_in, sink, ack_out, next, queue)
H2.  true
H3.  pkt.seqno = next
H4.  not ((next+1) in domain(queue))
 -->
C1.  receiver_int (pkt_in <: pkt, sink @ [pkt.mssg],
                   ack_out @ [next+1], next + 1, queue)
```

```
VC  6.  Event processing.  Event class "pkt_in", rule  4.

H1.  receiver_int (pkt_in, sink, ack_out, next, queue)
H2.  true
H3.  pkt.seqno = next
H4.  (next+1) in domain(queue)
 -->
C1.  receiver_int (
       pkt_in <: pkt,
       sink @ pkt.mssg :>
            apply (".mssg", range (consec (queue))),
       ack_out @ [reach(queue)],
       reach(queue),
       upper(queue, reach(queue)))
```

```
VC  7.  Event processing.  Event class "pkt_in", ELSE rule.

H1.  receiver_int (pkt_in, sink, ack_out, next, queue)
H2.  true
H3.         pkt.seqno > next
          and not (pkt.seqno - next < rcv_window)
     or     pkt.seqno > next
          and pkt.seqno - next < rcv_window
          and pkt.seqno in domain(queue)
 -->
C1.  receiver_int (pkt_in <: pkt, sink, ack_out, next, queue)
```

```
Verification condition for process "transport"


VC 1.  Concurrency VC.


H1.  sender_ext (source, sndr.ack_in, sndr.pkt_out)
H2.  sndr.ack_in initial am.p_out
H3.  receiver_ext (rcvr.pkt_in, sink, rcvr.ack_out)
H4.  rcvr.pkt_in initial pm.p_out
```

```
H5.  medium_ext (pm.p_in, pm.p_out)
H6.  pm.p_in initial sndr.pkt_out
H7.  medium_ext (am.p_in, am.p_out)
H8.  am.p_in initial  rcvr.ack_out
 -->
C1.  transport_ext (source, sink)
```

C.3 Verification Condition Proofs

Due to space limitations, we include only a few representative proofs. The first is the concurrency VC proof and the others are proofs of typical sequential VCs. The translation from expressions in our language to the notation of the Boyer-Moore theory was performed by hand.

```
Proof of VC  'TRANSPORT#1'

     (IMPLIES (AND (SENDER.EXT SOURCE ACK.IN PKT.OUT)
                   (RECEIVER.EXT PKT.IN SINK ACK.OUT)
                   (FOLLOWS PKT.IN PKT.OUT)
                   (FOLLOWS ACK.IN ACK.OUT))
              (INITIAL SINK SOURCE))

This formula can be simplified, using the abbreviations
SENDER.EXT, AND, and IMPLIES, to:

      (IMPLIES
        (AND (CONSISTENT (QUOTE SEQNO)
                         (QUOTE EQUAL)
                         PKT.OUT)
             (EQUAL SOURCE
                    (FAPPLY (QUOTE MSSG)
                            (RANGE (LATEST (QUOTE SEQNO)
                                           PKT.OUT))))
             (RECEIVER.EXT PKT.IN SINK ACK.OUT)
             (FOLLOWS PKT.IN PKT.OUT)
             (FOLLOWS ACK.IN ACK.OUT))
        (INITIAL SINK SOURCE)),

which we simplify, applying CONSISTENT.FOLLOWS, INITIAL.FAPPLY,
INITIAL.CONSEC.FOLLOWS, NUMBERP.SEQNO,
FOLLOWS.LATEST.CONSISTENT,
PMAPP.LATEST, INITIAL.RANGE, and INITIAL.TRANS, and opening up
RECEIVER.EXT and INITIAL, to:

      T.

Q.E.D.

13069 conses
9.963 seconds
0.0 seconds, garbage collection time


------------------------------------------------------------
```

```
Proof of VC  'SENDER#2'

     (IMPLIES (SENDER.INT SOURCE ACK.IN PKT.OUT STATE UNACK
                          NEXT QUEUE TOT)
              (SENDER.EXT SOURCE ACK.IN PKT.OUT))

This conjecture can be simplified, using the abbreviations
SENDER.INT and IMPLIES, to:

      (IMPLIES
       (AND
        (PMAPP QUEUE)
        (NUMBERP NEXT)
        (FOLLOWS (RANGE QUEUE) PKT.OUT)
        (CONSISTENT (QUOTE SEQNO)
                    (QUOTE EQUAL)
                    PKT.OUT)
        (EQUAL SOURCE
               (FAPPLY (QUOTE MSSG)
                       (RANGE (LATEST (QUOTE SEQNO) PKT.OUT))))
        (IF
         (EQUAL NEXT 0)
         (IF (EQUAL PKT.OUT (QUOTE (1QUOTE NULL)))
             (EQUAL QUEUE (QUOTE (1QUOTE NULL)))
             F)
         (IF (SEQP PKT.OUT)
             (IF (EQUAL (DOM (LST (LATEST (QUOTE SEQNO)
                                          PKT.OUT)))
                        (SUB1 NEXT))
                 (IF (EQUAL (HIGHEST (FAPPLY (QUOTE SEQNO)
                                              PKT.OUT))
                            (SUB1 NEXT))
                     (IF (SEQP QUEUE)
                         (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT))
                         T)
                     F)
                 F)
             F)))
       (SENDER.EXT SOURCE ACK.IN PKT.OUT)),

which simplifies, opening up SENDER.EXT, LATEST, RANGE, SEQP,
FAPPLY, EQUAL, and CONSISTENT, to:

      T.

Q.E.D.

15990 conses
13.576 seconds
3.261 seconds, garbage collection time


------------------------------------------------------------



Proof of VC  'SENDER#3'
```

```
     (IMPLIES (AND (SENDER.INT SOURCE ACK.IN PKT.OUT STATE
                               UNACK NEXT QUEUE TOT)
                   (NUMBERP ACK)
                   (LESSP UNACK ACK))
              (SENDER.INT SOURCE
                          (APR ACK.IN ACK)
                          PKT.OUT STATE UNACK NEXT
                          (UPPER QUEUE ACK)
                          TOT))
```

This conjecture can be simplified, using the abbreviations SENDER.INT, AND, and IMPLIES, to:

```
      (IMPLIES
       (AND
        (PMAPP QUEUE)
        (NUMBERP NEXT)
        (FOLLOWS (RANGE QUEUE) PKT.OUT)
        (CONSISTENT (QUOTE SEQNO)
                    (QUOTE EQUAL)
                    PKT.OUT)
        (EQUAL SOURCE
               (FAPPLY (QUOTE MSSG)
                       (RANGE (LATEST (QUOTE SEQNO)
                                      PKT.OUT))))
        (IF
         (EQUAL NEXT 0)
         (IF (EQUAL PKT.OUT (QUOTE (1QUOTE NULL)))
             (EQUAL QUEUE (QUOTE (1QUOTE NULL)))
             F)
         (IF (SEQP PKT.OUT)
             (IF (EQUAL (DOM (LST (LATEST (QUOTE SEQNO)
                                          PKT.OUT)))
                        (SUB1 NEXT))
                 (IF (EQUAL (HIGHEST (FAPPLY (QUOTE SEQNO)
                                              PKT.OUT))
                            (SUB1 NEXT))
                     (IF (SEQP QUEUE)
                         (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT))
                         T)
                     F)
                 F)
             F))
        (NUMBERP ACK)
        (LESSP UNACK ACK))
       (SENDER.INT SOURCE
                   (APR ACK.IN ACK)
                   PKT.OUT STATE UNACK NEXT
                   (UPPER QUEUE ACK)
                   TOT)),
```

which simplifies, rewriting with the lemmas LST.UPPER, FOLLOWS.RANGE.UPPER, and PMAPP.UPPER, and expanding the functions UPPER, FOLLOWS, SEQP, RANGE, PMAPP, SENDER.INT, LATEST, FAPPLY, EQUAL, and CONSISTENT, to:

```
      T.

Q.E.D.

24249 conses
20.704 seconds
0.0 seconds, garbage collection time


---------------------------------------------------------------


Proof of VC  'RECEIVER#2'

     (IMPLIES (RECEIVER.INT PKT.IN SINK ACK.OUT NEXT QUEUE)
              (RECEIVER.EXT PKT.IN SINK ACK.OUT))

This formula can be simplified, using the abbreviations
RECEIVER.EXT, RECEIVER.INT, and IMPLIES, to:

      (IMPLIES
       (AND
        (PMAPP QUEUE)
        (NUMBERP NEXT)
        (IF
         (CONSISTENT (QUOTE SEQNO)
                     (QUOTE EQUAL)
                     PKT.IN)
         (IF
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (IF
           (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                  NEXT)
           F
           (IF
            (IN 0
                (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
            (IF
             (LESSP 0 NEXT)
             (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO)
                                               PKT.IN)
                                       (SUB1 NEXT)))))
             F)
            (IF (EQUAL NEXT 0)
                (EQUAL SINK (QUOTE (1QUOTE NULL)))
                F)))
          F)
         T)
        (CONSISTENT (QUOTE SEQNO)
                    (QUOTE EQUAL)
                    PKT.IN))
       (IF
```

```
        (IN 0
            (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
        (INITIAL SINK
            (FAPPLY (QUOTE MSSG)
                    (RANGE (CONSEC (LATEST (QUOTE SEQNO)
                                           PKT.IN)))))
        (EQUAL SINK (QUOTE (1QUOTE NULL)))))),
```

which simplifies, expanding the functions EQUAL and LESSP, to:

```
      (IMPLIES
       (AND
          (PMAPP QUEUE)
          (NUMBERP NEXT)
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                      NEXT))
          (IN 0
              (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL NEXT 0))
          (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO)
                                               PKT.IN)
                                       (SUB1 NEXT)))))
          (CONSISTENT (QUOTE SEQNO)
                      (QUOTE EQUAL)
                      PKT.IN))
       (INITIAL SINK
          (FAPPLY (QUOTE MSSG)
                  (RANGE (CONSEC (LATEST (QUOTE SEQNO)
                                         PKT.IN)))))),
```

which again simplifies, using linear arithmetic and applying INITIAL.RANGE, PMAPP.LATEST, INITIAL.LOWER.CONSEC, and INITIAL.FAPPLY, to:

```
      T.
```

Q.E.D.

```
42883 conses
35.007 seconds
6.695 seconds, garbage collection time
```

Proof of VC ’RECEIVER#5’

```
     (IMPLIES (AND (RECEIVER.INT PKT.IN SINK ACK.OUT NEXT
                                 QUEUE)
                   (LESSP NEXT (SEQNO PKT))
```

```
                (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
                (PKTP PKT))
           (RECEIVER.INT (APR PKT.IN PKT)
                         SINK ACK.OUT NEXT
                         (WITHE QUEUE (SEQNO PKT) PKT)))
```

This formula can be simplified, using the abbreviations NOT, RECEIVER.INT, AND, and IMPLIES, to:

```
      (IMPLIES
       (AND
        (PMAPP QUEUE)
        (NUMBERP NEXT)
        (IF
         (CONSISTENT (QUOTE SEQNO)
                     (QUOTE EQUAL)
                     PKT.IN)
         (IF
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (IF
           (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                  NEXT)
           F
           (IF
            (IN 0
                (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
            (IF
             (LESSP 0 NEXT)
             (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO)
                                               PKT.IN)
                                       (SUB1 NEXT)))))
             F)
            (IF (EQUAL NEXT 0)
                (EQUAL SINK (QUOTE (1QUOTE NULL)))
                F)))
           F)
          T)
         (LESSP NEXT (SEQNO PKT))
         (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
         (PKTP PKT))
        (RECEIVER.INT (APR PKT.IN PKT)
                      SINK ACK.OUT NEXT
                      (WITHE QUEUE (SEQNO PKT) PKT))),
```

which we simplify, applying LST.APR, NLST.APR, APPLY2.EQUAL, APPLY1.SEQNO, PMAPP.WITHE, IN.DOMAIN.WITHE.IF, SUB1.ADD1, UPPER.WITHE.IF, PMAPP.LATEST, and LOWER.WITHE.IF, and opening up EQUAL, LESSP, CONSISTENT, CONSISTENT2, RECEIVER.INT, and LATEST, to the following nine new goals:

```
Case 9. (IMPLIES
              (AND (PMAPP QUEUE)
                   (NUMBERP NEXT)
```

```
                    (FOLLOWS QUEUE
                             (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                                    (ADD1 NEXT)))
                    (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                                NEXT))
                    (NOT (IN 0
                             (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN))))
                    (EQUAL NEXT 0)
                    (EQUAL SINK (QUOTE (1QUOTE NULL)))
                    (NOT (EQUAL (SEQNO PKT) 0))
                    (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
                    (PKTP PKT)
                    (CONSISTENT2 (QUOTE SEQNO)
                                 (QUOTE EQUAL)
                                 PKT.IN PKT)
                    (CONSISTENT (QUOTE SEQNO)
                                (QUOTE EQUAL)
                                PKT.IN)
                    (LESSP (SUB1 (SEQNO PKT)) NEXT))
               (FOLLOWS (WITHE QUEUE (SEQNO PKT) PKT)
                        (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                               (ADD1 NEXT)))),
```

which we again simplify, using linear arithmetic, to:

```
      T.
```

```
Case 8. (IMPLIES
               (AND (PMAPP QUEUE)
                    (NUMBERP NEXT)
                    (FOLLOWS QUEUE
                             (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                                    (ADD1 NEXT)))
                    (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                                NEXT))
                    (NOT (IN 0
                             (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN))))
                    (EQUAL NEXT 0)
                    (EQUAL SINK (QUOTE (1QUOTE NULL)))
                    (NOT (EQUAL (SEQNO PKT) 0))
                    (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
                    (PKTP PKT)
                    (CONSISTENT2 (QUOTE SEQNO)
                                 (QUOTE EQUAL)
                                 PKT.IN PKT)
                    (CONSISTENT (QUOTE SEQNO)
                                (QUOTE EQUAL)
                                PKT.IN)
                    (NOT (LESSP (SUB1 (SEQNO PKT)) NEXT)))
               (FOLLOWS (WITHE QUEUE (SEQNO PKT) PKT)
                        (WITHE (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                                      (ADD1 NEXT))
                               (SEQNO PKT)
                               PKT))),
```

which again simplifies, applying the lemmas RNG.MPAIR, DOM.MPAIR, IN.WITHE.MPAIRP, IN.DOMAIN.UPPER, FOLLOWS.WITHE.IF, PMAPP.WITHE,

```
  PMAPP.LATEST, PMAPP.UPPER, and FOLLOWS.WITHE.IN.MPAIR, and
  expanding the functions NUMBERP, EQUAL, and LESSP, to:

        T.

Case 7. (IMPLIES
         (AND
          (PMAPP QUEUE)
          (NUMBERP NEXT)
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                      NEXT))
          (IN 0
              (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL NEXT 0))
          (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                       (SUB1 NEXT)))))
          (LESSP NEXT (SEQNO PKT))
          (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
          (PKTP PKT)
          (CONSISTENT2 (QUOTE SEQNO)
                       (QUOTE EQUAL)
                       PKT.IN PKT)
          (CONSISTENT (QUOTE SEQNO)
                      (QUOTE EQUAL)
                      PKT.IN)
          (EQUAL (SEQNO PKT) (SUB1 NEXT))
          (NOT (IN (MPAIR (SEQNO PKT) PKT)
                   (LATEST (QUOTE SEQNO) PKT.IN)))
          (EQUAL (SUB1 NEXT) 0))
         (EQUAL SINK
                (FAPPLY (QUOTE MSSG)
                        (RANGE (APR (QUOTE (1QUOTE NULL))
                                    (MPAIR 0 PKT))))))),

  which again simplifies, using linear arithmetic, to:

        T.

Case 6. (IMPLIES
         (AND
          (PMAPP QUEUE)
          (NUMBERP NEXT)
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                      NEXT))
          (IN 0
              (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL NEXT 0))
          (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
```

```
                    (RANGE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                  (SUB1 NEXT)))))
          (LESSP NEXT (SEQNO PKT))
          (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
          (PKTP PKT)
          (CONSISTENT2 (QUOTE SEQNO)
                       (QUOTE EQUAL)
                       PKT.IN PKT)
          (CONSISTENT (QUOTE SEQNO)
                      (QUOTE EQUAL)
                      PKT.IN)
          (EQUAL (SEQNO PKT) (SUB1 NEXT))
          (NOT (IN (MPAIR (SEQNO PKT) PKT)
                   (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL (SUB1 NEXT) 0)))
         (EQUAL SINK
            (FAPPLY (QUOTE MSSG)
                    (RANGE (APR (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                       (SUB1 (SUB1 NEXT)))
                                (MPAIR (SEQNO PKT) PKT)))))).
```

However this simplifies again, using linear arithmetic, to:

T.

```
Case 5. (IMPLIES
          (AND
           (PMAPP QUEUE)
           (NUMBERP NEXT)
           (FOLLOWS QUEUE
                    (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                           (ADD1 NEXT)))
           (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                       NEXT))
           (IN 0
               (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
           (NOT (EQUAL NEXT 0))
           (EQUAL SINK
                  (FAPPLY (QUOTE MSSG)
                          (RANGE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                        (SUB1 NEXT)))))
           (LESSP NEXT (SEQNO PKT))
           (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
           (PKTP PKT)
           (CONSISTENT2 (QUOTE SEQNO)
                        (QUOTE EQUAL)
                        PKT.IN PKT)
           (CONSISTENT (QUOTE SEQNO)
                       (QUOTE EQUAL)
                       PKT.IN)
           (NOT (EQUAL (SEQNO PKT) (SUB1 NEXT)))
           (NOT (LESSP (SUB1 NEXT) (SEQNO PKT))))
          (EQUAL SINK
           (FAPPLY (QUOTE MSSG)
                   (RANGE (WITHE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                        (SUB1 NEXT))
                                 (SEQNO PKT)
```

```
                                PKT))))),
```

which again simplifies, using linear arithmetic, to:

```
       T.

Case 4. (IMPLIES
         (AND
          (PMAPP QUEUE)
          (NUMBERP NEXT)
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                      NEXT))
          (IN 0
              (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL NEXT 0))
          (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                       (SUB1 NEXT)))))
          (LESSP NEXT (SEQNO PKT))
          (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
          (PKTP PKT)
          (CONSISTENT2 (QUOTE SEQNO)
                       (QUOTE EQUAL)
                       PKT.IN PKT)
          (CONSISTENT (QUOTE SEQNO)
                      (QUOTE EQUAL)
                      PKT.IN))
         (NOT (LESSP (REACH (WITHE (LATEST (QUOTE SEQNO) PKT.IN)
                                   (SEQNO PKT)
                                   PKT))
                     NEXT))).
```

But this simplifies again, using linear arithmetic and rewriting with LESSP.REACH.WITHE.2 and PMAPP.LATEST, to:

```
       T.

Case 3. (IMPLIES
         (AND
          (PMAPP QUEUE)
          (NUMBERP NEXT)
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                      NEXT))
          (IN 0
              (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL NEXT 0))
          (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                       (SUB1 NEXT)))))
```

```
          (LESSP NEXT (SEQNO PKT))
          (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
          (PKTP PKT)
          (CONSISTENT2 (QUOTE SEQNO)
                       (QUOTE EQUAL)
                       PKT.IN PKT)
          (CONSISTENT (QUOTE SEQNO)
                      (QUOTE EQUAL)
                      PKT.IN)
          (EQUAL (SEQNO PKT) 0))
         (FOLLOWS (WITHE QUEUE (SEQNO PKT) PKT)
                  (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                         (ADD1 NEXT)))).
```

This again simplifies, using linear arithmetic, to:

```
       T.

Case 2. (IMPLIES
         (AND
          (PMAPP QUEUE)
          (NUMBERP NEXT)
          (FOLLOWS QUEUE
                   (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                          (ADD1 NEXT)))
          (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                      NEXT))
          (IN 0
              (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL NEXT 0))
          (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                       (SUB1 NEXT)))))
          (LESSP NEXT (SEQNO PKT))
          (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
          (PKTP PKT)
          (CONSISTENT2 (QUOTE SEQNO)
                       (QUOTE EQUAL)
                       PKT.IN PKT)
          (CONSISTENT (QUOTE SEQNO)
                      (QUOTE EQUAL)
                      PKT.IN)
          (LESSP (SUB1 (SEQNO PKT)) NEXT))
         (FOLLOWS (WITHE QUEUE (SEQNO PKT) PKT)
                  (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                         (ADD1 NEXT)))),
```

which again simplifies, using linear arithmetic, to:

```
       T.

Case 1. (IMPLIES
         (AND
          (PMAPP QUEUE)
          (NUMBERP NEXT)
          (FOLLOWS QUEUE
```

```
                (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                       (ADD1 NEXT)))
          (NOT (LESSP (REACH (LATEST (QUOTE SEQNO) PKT.IN))
                      NEXT))
          (IN 0
              (DOMAIN (LATEST (QUOTE SEQNO) PKT.IN)))
          (NOT (EQUAL NEXT 0))
          (EQUAL SINK
                 (FAPPLY (QUOTE MSSG)
                         (RANGE (LOWER (LATEST (QUOTE SEQNO) PKT.IN)
                                       (SUB1 NEXT)))))
          (LESSP NEXT (SEQNO PKT))
          (NOT (IN (SEQNO PKT) (DOMAIN QUEUE)))
          (PKTP PKT)
          (CONSISTENT2 (QUOTE SEQNO)
                       (QUOTE EQUAL)
                       PKT.IN PKT)
          (CONSISTENT (QUOTE SEQNO)
                      (QUOTE EQUAL)
                      PKT.IN)
          (NOT (EQUAL (SEQNO PKT) 0))
          (NOT (LESSP (SUB1 (SEQNO PKT)) NEXT)))
         (FOLLOWS (WITHE QUEUE (SEQNO PKT) PKT)
                  (WITHE (UPPER (LATEST (QUOTE SEQNO) PKT.IN)
                                (ADD1 NEXT))
                         (SEQNO PKT)
                         PKT))),
```

which again simplifies, rewriting with RNG.MPAIR, DOM.MPAIR, IN.WITHE.MPAIRP, SUB1.ADD1, IN.DOMAIN.UPPER, FOLLOWS.WITHE.IF, PMAPP.WITHE, PMAPP.LATEST, PMAPP.UPPER, and FOLLOWS.WITHE.IN.MPAIR, and expanding LESSP, to:

```
      T.
```

Q.E.D.

```
176501 conses
161.821 seconds
23.681 seconds, garbage collection time
```

Appendix D
Analysis Results: NanoTCP Protocol

In this appendix, we partially show the results of analyzing the NanoTCP protocol. This information is analogous to that of Appendix C. In addition we present part of the log from the VC reduction phase.

D.1 Decision Table Analysis

```
Decision table analysis for process "sender"


Table for event class "source"

No logical dependencies are in effect.


All rules are satisfiable.


No ambiguities have been detected.


Out of a total of    4 possible enabling combinations,    2 are
accounted for by the rules of the table.  Therefore the table
is incomplete and has the following ELSE rules.

    ELSE conditions
                                    1
  +================================+===+
 1| timing                         | - |
 2| mess = null                    | T |
  +================================+===+
```

```
Table for event class "ack_in"

No logical dependencies are in effect.


All rules are satisfiable.


No ambiguities have been detected.


Out of a total of    8 possible enabling combinations,    8 are
accounted for by the rules of the table.  Therefore the table
is complete and has no ELSE rule.
```

Table for event class "timeout"

No logical dependencies are in effect.

All rules are satisfiable.

No ambiguities have been detected.

Out of a total of 1 possible enabling combinations, 1 are accounted for by the rules of the table. Therefore the table is complete and has no ELSE rule.

Decision table analysis for process "receiver"

Table for event class "credits"

No logical dependencies are in effect.

All rules are satisfiable.

No ambiguities have been detected.

Out of a total of 1 possible enabling combinations, 1 are accounted for by the rules of the table. Therefore the table is complete and has no ELSE rule.

Table for event class "pkt_in"

The following logical dependencies are in effect:

Conditions 1 and 2 are mutually exclusive.
Condition 2 implies not condition 3.

All rules are satisfiable.

No ambiguities have been detected.

Out of a total of 10 possible enabling combinations, 5 are accounted for by the rules of the table. Therefore the table is incomplete and has the following ELSE rules.

```
   ELSE conditions
                                    1 2
  +================================+=====+
 1| last(pkt).dom < next           | - F |
 2| next < first(pkt).dom          | - F |
 3| pkt = null                     | T F |
 4| next le edge                   | - F |
  +================================+=====+
```

D.2 Verification Condition Generation

```
Verification conditions for process "sender"


VC 1.  Initialization.

    sender_int (null, null, null, null,
                0, 0, 0, null, false, 0)

_____________________________________________________________________


VC 2.  External invariant.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
 -->
C1.  sender_ext (source, ack_in, receipts, pkt_out)

_____________________________________________________________________


VC  3.  Event processing.  Event class "source", rule  1.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  next le edge
H3.  not (timing)
H4.  not (mess = null)
 -->
C1.  sender_int (source <: mess, ack_in, receipts,
                 pkt_out @ [enmap (mess, next)],
                 unack, next + size(mess), edge,
                 queue union enmap (mess, next),
                 true, time + delta_t)


_____________________________________________________________________


VC  4.  Event processing.  Event class "source", rule  2.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
```

```
                edge, queue, timing, to_time)
H2.  next le edge
H3.  timing
H4.  not (mess = null)
 -->
C1.  sender_int (source <: mess, ack_in, receipts,
                 pkt_out @ [enmap (mess, next)],
                 unack, next + size(mess), edge,
                 queue union enmap (mess, next),
                 timing, to_time)
```

```
VC  5.  Event processing.  Event class "source", ELSE rule.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  next le edge
H3.  mess = null
 -->
C1.  sender_int (source <: mess, ack_in, receipts, pkt_out,
                 unack, next, edge, queue, timing, to_time)
```

```
VC  6.  Event processing.  Event class "ack_in", rule  1.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  true
H3.  not (timing)
 -->
C1.  sender_int (source, ack_in <: ack, receipts, pkt_out,
                 unack, next, max (edge, ack.edge),
                 queue, timing, to_time)
```

```
VC  7.  Event processing.  Event class "ack_in", rule  2.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  true
H3.  not (unack < ack.ackno)
H4.  timing
 -->
C1.  sender_int (source, ack_in <: ack, receipts, pkt_out,
                 unack, next, max (edge, ack.edge),
                 queue, timing, to_time)
```

```
VC  8.  Event processing.  Event class "ack_in", rule  3.
```

```
H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  true
H3.  unack < ack.ackno
H4.  not (next = ack.ackno)
H5.  timing
 -->
C1.  sender_int (source, ack_in <: ack,
                 receipts @ [ack.ackno - unack],
                 pkt_out, ack.ackno, next,
                 max (edge, ack.edge), upper (queue, ack.ackno),
                 timing, to_time)
```

```
VC  9.  Event processing.  Event class "ack_in", rule  4.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  true
H3.  unack < ack.ackno
H4.  next = ack.ackno
H5.  timing
 -->
C1.  sender_int (source, ack_in <: ack,
                 receipts @ [ack.ackno - unack],
                 pkt_out, ack.ackno, next, max (edge, ack.edge),
                 null, false, to_time)
```

```
VC 10.  Event processing.  Event class "ack_in", ELSE rule.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  true
H3.  false
 -->
C1.  sender_int (source, ack_in <: ack, receipts, pkt_out,
                 unack, next, edge, queue, timing, to_time)
```

```
VC 11.  Event processing.  Event class "timeout", rule  1.

H1.  sender_int (source, ack_in, receipts, pkt_out, unack, next,
                 edge, queue, timing, to_time)
H2.  timing
 -->
C1.  sender_int (source, ack_in, receipts, pkt_out @ [queue],
                 unack, next, edge, queue, timing, time + delta_t)
```

```
Verification conditions for process "receiver"


VC 1.  Initialization.

    receiver_int (null, null, null, null, 0, 0, null)

_____________________________________________________________________


VC 2.  External invariant.

H1.  receiver_int (credits, pkt_in, sink, ack_out, next, edge, queue)
 -->
C1.  receiver_ext (credits, pkt_in, sink, ack_out)

_____________________________________________________________________


VC  3.  Event processing.  Event class "credits", rule  1.

H1.  receiver_int (credits, pkt_in, sink, ack_out, next, edge, queue)
H2.  true
 -->
C1.  receiver_int (credits <: cred, pkt_in, sink,
                   ack_out @ [[next, edge+cred]], next,
                   edge + cred, queue)

_____________________________________________________________________


VC  4.  Event processing.  Event class "credits", ELSE rule.

H1.  receiver_int (credits, pkt_in, sink, ack_out, next, edge, queue)
H2.  true
H3.  false
 -->
C1.  receiver_int (credits <: cred, pkt_in, sink, ack_out,
                   next, edge, queue)

_____________________________________________________________________


VC  5.  Event processing.  Event class "pkt_in", rule  1.

H1.  receiver_int (credits, pkt_in, sink, ack_out, next, edge, queue)
H2.  true
H3.  last(pkt).dom < next
H4.  not (pkt = null)
 -->
C1.  receiver_int (credits, pkt_in <: pkt, sink,
                   ack_out @ [[next, edge]], next, edge, queue)

_____________________________________________________________________


VC  6.  Event processing.  Event class "pkt_in", rule  2.
```

```
H1.  receiver_int (credits, pkt_in, sink, ack_out, next, edge, queue)
H2.  true
H3.  next < first(pkt).dom
H4.  not (pkt = null)
 -->
C1.  receiver_int (credits, pkt_in <: pkt, sink, ack_out, next,
                   edge, [queue union lower (pkt, edge)])
```

```
VC  7.  Event processing.  Event class "pkt_in", rule  3.

H1.  receiver_int (credits, pkt_in, sink, ack_out, next, edge, queue)
H2.  true
H3.  not (last(pkt).dom < next)
H4.  not (next < first(pkt).dom)
H5.  not (pkt = null)
H6.  next le edge
 -->
C1.  receiver_int (credits, pkt_in <: pkt,
                   sink @ [range (consec (M))],
                   ack_out @ [[reach(M), edge]],
                   reach(M), edge, upper (queue, reach(M) + 1))
```

```
VC  8.  Event processing.  Event class "pkt_in", ELSE rule.

H1.  receiver_int (credits, pkt_in, sink, ack_out, next, edge, queue)
H2.  true
H3.     pkt = null
     or    (not (last(pkt).dom < next)
        and not (next < first(pkt).dom)
        and not (pkt = null)
        and not (next le edge) )
 -->
C1.  receiver_int (credits, pkt_in <: pkt, sink, ack_out,
                   next, edge, queue)
```

```
Verification condition for process "NanoTCP"


VC 1.  Concurrency VC.


H1.  sender_ext (source, sndr.ack_in, receipts, sndr.pkt_out)
H2.  sndr.ack_in initial am.m_out
H3.  receiver_ext (credits, rcvr.pkt_in, sink, rcvr.ack_out)
H4.  rcvr.pkt_in initial pm.m_out
H5.  medium_ext (pm.m_in, pm.m_out)
```

```
H6.  pm.m_in initial sndr.pkt_out
H7.  medium_ext (am.m_in, am.m_out)
H8.  am.m_in initial rcvr.ack_out
 -->
C1.  NanoTCP_ext (source, credits, sink, receipts)
```

D.3 Verification Condition Reduction

We show the log that resulted from running the VC reduction program against one of the receiver's VCs. The effect is to produce several new VCs, after determining that some cases are trivially true and others are subsumed by previously generated VCs.

```
Path conditions for VC 'RECEIVER#6':

   (AND (LESSP NEXT (DOM (FIRST PKT)))
        (SEQP PKT)
        (PMAPP PKT))

Variable substitution:

   CREDITS      :=  CREDITS
   PKT.IN       :=  (APR PKT.IN PKT)
   SINK         :=  SINK
   ACK.OUT      :=  ACK.OUT
   NEXT         :=  NEXT
   EDGE         :=  EDGE
   QUEUE        :=  (REDUCE (QUOTE WITH)
                            QUEUE
                            (LOWER PKT EDGE))

Instantiated assertion:

   (AND
     (PMAPP (REDUCE (QUOTE WITH)
                    QUEUE
                    (LOWER PKT EDGE)))
     (NUMBERP NEXT)
     (NUMBERP EDGE)
     (IMPLIES
       (AND (CONSISTENT (QUOTE DOM)
                        (QUOTE EQUAL)
                        (FLATTEN (APR PKT.IN PKT)))
            (ALL (FAPPLY (QUOTE CONTIG)
                         (APR PKT.IN PKT))))
       (AND
         (FOLLOWS (REDUCE (QUOTE WITH)
                          QUEUE
                          (LOWER PKT EDGE))
                  (UPPER (REDUCE (QUOTE WITH)
                                 (NULL)
                                 (FLATTEN (APR PKT.IN PKT)))
                         (ADD1 NEXT)))
         (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                    (NULL)
```

```
                                    (FLATTEN (APR PKT.IN PKT))))
                    NEXT))
          (ALL (FAPPLY (QUOTE MPAIRP)
                       (FLATTEN (APR PKT.IN PKT))))
          (IF
            (IN 0 (DOMAIN (FLATTEN (APR PKT.IN PKT))))
            (AND (LESSP 0 NEXT)
                 (EQUAL (FLATTEN SINK)
                        (RANGE (LOWER (REDUCE (QUOTE WITH)
                                              (NULL)
                                              (FLATTEN (APR PKT.IN
                                                            PKT)))
                                       (SUB1 NEXT)))))
            (AND (EQUAL NEXT 0)
                 (EQUAL SINK (NULL)))))))

--------------------

VC 'RECEIVER#5.0.1' subsumes VC 'RECEIVER#6.0.1':

   (EQUAL (FLATTEN (APR PKT.IN PKT))
          (JOIN (FLATTEN PKT.IN)
                PKT))

Equality substitution performed.

--------------------

Generating new VC 'RECEIVER#6.1':

   (IMPLIES (PMAPP QUEUE)
            (PMAPP (REDUCE (QUOTE WITH)
                           QUEUE
                           (LOWER PKT EDGE))))


VC not proved, being written.

--------------------

VC 'RECEIVER#6.2' is trivially true:

   (IMPLIES (NUMBERP NEXT)
            (NUMBERP NEXT))

--------------------

VC 'RECEIVER#6.3' is trivially true:

   (IMPLIES (NUMBERP EDGE)
            (NUMBERP EDGE))

--------------------

VC 'RECEIVER#5.0.2.1' subsumes VC 'RECEIVER#6.4.1.1':

   (IMPLIES
```

```
     (AND
       (ALL (FAPPLY (QUOTE CONTIG)
                    (APR PKT.IN PKT)))
       (OR
         (NOT (FOLLOWS (REDUCE (QUOTE WITH)
                               QUEUE
                               (LOWER PKT EDGE))
                       (UPPER (REDUCE (QUOTE WITH)
                                      (NULL)
                                      (JOIN (FLATTEN PKT.IN)
                                            PKT))
                              (ADD1 NEXT))))
         (LESSP (REACH (REDUCE (QUOTE WITH)
                               (NULL)
                               (JOIN (FLATTEN PKT.IN)
                                     PKT)))
                NEXT)
         (NOT (ALL (FAPPLY (QUOTE MPAIRP)
                           (JOIN (FLATTEN PKT.IN)
                                 PKT))))
         (IF
           (IN 0 (DOMAIN (JOIN (FLATTEN PKT.IN)
                               PKT)))
           (OR
             (NOT (LESSP 0 NEXT))
             (NOT
               (EQUAL (FLATTEN SINK)
                      (RANGE (LOWER (REDUCE (QUOTE WITH)
                                            (NULL)
                                            (JOIN (FLATTEN PKT.IN)
                                                  PKT))
                                    (SUB1 NEXT))))))
           (OR (NOT (EQUAL NEXT 0))
               (NOT (EQUAL SINK (NULL))))))
       (PMAPP QUEUE)
       (NUMBERP NEXT)
       (NUMBERP EDGE)
       (LESSP NEXT (DOM (FIRST PKT)))
       (SEQP PKT)
       (PMAPP PKT)
       (CONSISTENT (QUOTE DOM)
                   (QUOTE EQUAL)
                   (JOIN (FLATTEN PKT.IN)
                         PKT)))
     (CONSISTENT (QUOTE DOM)
                 (QUOTE EQUAL)
                 (FLATTEN PKT.IN)))

--------------------

VC 'RECEIVER#5.0.2.2' subsumes VC 'RECEIVER#6.4.1.2':

   (IMPLIES
     (AND
       (CONSISTENT (QUOTE DOM)
                   (QUOTE EQUAL)
                   (JOIN (FLATTEN PKT.IN)
```

```
                    PKT))
      (OR
        (NOT (FOLLOWS (REDUCE (QUOTE WITH)
                              QUEUE
                              (LOWER PKT EDGE))
                      (UPPER (REDUCE (QUOTE WITH)
                                     (NULL)
                                     (JOIN (FLATTEN PKT.IN)
                                           PKT))
                             (ADD1 NEXT))))
        (LESSP (REACH (REDUCE (QUOTE WITH)
                              (NULL)
                              (JOIN (FLATTEN PKT.IN)
                                    PKT)))
               NEXT)
        (NOT (ALL (FAPPLY (QUOTE MPAIRP)
                          (JOIN (FLATTEN PKT.IN)
                                PKT))))
        (IF
          (IN 0 (DOMAIN (JOIN (FLATTEN PKT.IN)
                              PKT)))
          (OR
            (NOT (LESSP 0 NEXT))
            (NOT
              (EQUAL (FLATTEN SINK)
                     (RANGE (LOWER (REDUCE (QUOTE WITH)
                                           (NULL)
                                           (JOIN (FLATTEN PKT.IN)
                                                 PKT))
                                   (SUB1 NEXT))))))
          (OR (NOT (EQUAL NEXT 0))
              (NOT (EQUAL SINK (NULL))))))
        (PMAPP QUEUE)
        (NUMBERP NEXT)
        (NUMBERP EDGE)
        (LESSP NEXT (DOM (FIRST PKT)))
        (SEQP PKT)
        (PMAPP PKT)
        (ALL (FAPPLY (QUOTE CONTIG)
                     (APR PKT.IN PKT))))
      (ALL (FAPPLY (QUOTE CONTIG)
                   PKT.IN)))

--------------------

Generating new VC 'RECEIVER#6.4.2.1':

   (IMPLIES (AND (FOLLOWS QUEUE (UPPER (REDUCE (QUOTE WITH)
                                               (NULL)
                                               (FLATTEN PKT.IN))
                                       (ADD1 NEXT)))
                 (ALL (FAPPLY (QUOTE MPAIRP)
                              (FLATTEN PKT.IN)))
                 (CONSISTENT (QUOTE DOM)
                             (QUOTE EQUAL)
                             (JOIN (FLATTEN PKT.IN)
                                   PKT))
```

```
                (PMAPP QUEUE)
                (NUMBERP NEXT)
                (LESSP NEXT (DOM (FIRST PKT)))
                (PMAPP PKT))
           (FOLLOWS (REDUCE (QUOTE WITH)
                            QUEUE
                            (LOWER PKT EDGE))
                    (UPPER (REDUCE (QUOTE WITH)
                                   (NULL)
                                   (JOIN (FLATTEN PKT.IN)
                                         PKT))
                           (ADD1 NEXT))))


VC not proved, being written.

To be proved with lemmas (FOLLOWS.REDUCE.WITH.UPPER)


--------------------

Generating new VC 'RECEIVER#6.4.2.2':

   (IMPLIES (AND (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                            (NULL)
                                            (FLATTEN PKT.IN)))
                             NEXT))
                 (NUMBERP NEXT)
                 (LESSP NEXT (DOM (FIRST PKT)))
                 (PMAPP PKT))
            (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                       (NULL)
                                       (JOIN (FLATTEN PKT.IN)
                                             PKT)))
                        NEXT)))


VC not proved, being written.

--------------------

VC 'RECEIVER#5.4.2.3' subsumes VC 'RECEIVER#6.4.2.3':

   (IMPLIES
     (AND
       (FOLLOWS QUEUE (UPPER (REDUCE (QUOTE WITH)
                                     (NULL)
                                     (FLATTEN PKT.IN))
                             (ADD1 NEXT)))
       (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                  (NULL)
                                  (FLATTEN PKT.IN)))
                   NEXT))
       (IF (IN 0 (DOMAIN (FLATTEN PKT.IN)))
           (AND (LESSP 0 NEXT)
                (EQUAL (FLATTEN SINK)
                       (RANGE (LOWER (REDUCE (QUOTE WITH)
```

```
                                              (NULL)
                                              (FLATTEN PKT.IN))
                                        (SUB1 NEXT)))))
           (AND (EQUAL NEXT 0)
                (EQUAL SINK (NULL))))
      (CONSISTENT (QUOTE DOM)
                  (QUOTE EQUAL)
                  (JOIN (FLATTEN PKT.IN)
                        PKT))
      (ALL (FAPPLY (QUOTE CONTIG)
                   (APR PKT.IN PKT)))
      (PMAPP QUEUE)
      (NUMBERP NEXT)
      (NUMBERP EDGE)
      (LESSP NEXT (DOM (FIRST PKT)))
      (SEQP PKT)
      (PMAPP PKT)
      (ALL (FAPPLY (QUOTE MPAIRP)
                   (FLATTEN PKT.IN))))
    (ALL (FAPPLY (QUOTE MPAIRP)
                 (JOIN (FLATTEN PKT.IN)
                       PKT))))

--------------------

VC 'RECEIVER#6.4.2.4.1.1' is trivially true:

   (IMPLIES (LESSP 0 NEXT)
            (LESSP 0 NEXT))

--------------------

Using equality hypothesis:
   (EQUAL (FLATTEN SINK)
          (RANGE (LOWER (REDUCE (QUOTE WITH)
                                (NULL)
                                (FLATTEN PKT.IN))
                        (SUB1 NEXT))))

Generating new VC 'RECEIVER#6.4.2.4.1.2':

   (IMPLIES (AND (LESSP 0 NEXT)
                 (NUMBERP NEXT)
                 (LESSP NEXT (DOM (FIRST PKT)))
                 (PMAPP PKT))
            (EQUAL (RANGE (LOWER (REDUCE (QUOTE WITH)
                                         (NULL)
                                         (FLATTEN PKT.IN))
                                 (SUB1 NEXT)))
                   (RANGE (LOWER (REDUCE (QUOTE WITH)
                                         (NULL)
                                         (JOIN (FLATTEN PKT.IN)
                                               PKT))
                                 (SUB1 NEXT)))))


VC not proved, being written.
```

```
--------------------

Generating new VC 'RECEIVER#6.4.2.4.2':

   (IMPLIES (AND (IN 0 (DOMAIN (JOIN (FLATTEN PKT.IN)
                                     PKT)))
                 (EQUAL NEXT 0)
                 (LESSP NEXT (DOM (FIRST PKT)))
                 (PMAPP PKT))
            (IN 0 (DOMAIN (FLATTEN PKT.IN))))


VC not proved, being written.

--------------------

VC 'RECEIVER#5.4.2.4.3' subsumes VC 'RECEIVER#6.4.2.4.3':

   (IMPLIES (AND (OR (NOT (EQUAL NEXT 0))
                     (NOT (EQUAL SINK (NULL))))
                 (LESSP 0 NEXT)
                 (EQUAL (FLATTEN SINK)
                        (RANGE (LOWER (REDUCE (QUOTE WITH)
                                              (NULL)
                                              (FLATTEN PKT.IN))
                                      (SUB1 NEXT))))
                 (FOLLOWS QUEUE (UPPER (REDUCE (QUOTE WITH)
                                               (NULL)
                                               (FLATTEN PKT.IN))
                                       (ADD1 NEXT)))
                 (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                            (NULL)
                                            (FLATTEN PKT.IN)))
                             NEXT))
                 (ALL (FAPPLY (QUOTE MPAIRP)
                              (FLATTEN PKT.IN)))
                 (CONSISTENT (QUOTE DOM)
                             (QUOTE EQUAL)
                             (JOIN (FLATTEN PKT.IN)
                                   PKT))
                 (ALL (FAPPLY (QUOTE CONTIG)
                              (APR PKT.IN PKT)))
                 (PMAPP QUEUE)
                 (NUMBERP NEXT)
                 (NUMBERP EDGE)
                 (LESSP NEXT (DOM (FIRST PKT)))
                 (SEQP PKT)
                 (PMAPP PKT)
                 (IN 0 (DOMAIN (FLATTEN PKT.IN))))
            (IN 0 (DOMAIN (JOIN (FLATTEN PKT.IN)
                                PKT))))

--------------------

VC 'RECEIVER#6.4.2.4.4.1' is trivially true:

   (IMPLIES (EQUAL NEXT 0)
```

```
            (EQUAL NEXT 0))

--------------------

VC 'RECEIVER#6.4.2.4.4.2' is trivially true:

   (IMPLIES (EQUAL SINK (NULL))
            (EQUAL SINK (NULL)))
```

D.4 Verification Condition Proofs

```
Proof of VC  'NANOTCP#1'

     (IMPLIES (AND (SENDER.EXT SOURCE ACK.IN RECEIPTS PKT.OUT)
                   (RECEIVER.EXT CREDITS PKT.IN SINK ACK.OUT)
                   (FOLLOWS PKT.IN PKT.OUT)
                   (ALL (FAPPLY (QUOTE PMAPP) PKT.OUT)))
              (INITIAL (FLATTEN SINK)
                       (FLATTEN SOURCE)))

This formula can be simplified, using the abbreviations SENDER.EXT,
AND, and IMPLIES, to:

      (IMPLIES (AND (CONSISTENT (QUOTE DOM)
                                (QUOTE EQUAL)
                                (FLATTEN PKT.OUT))
                    (ALL (FAPPLY (QUOTE CONTIG) PKT.OUT))
                    (EQUAL (FLATTEN SOURCE)
                           (RANGE (REDUCE (QUOTE WITH)
                                          (QUOTE (1QUOTE NULL))
                                          (FLATTEN PKT.OUT))))
                    (RECEIVER.EXT CREDITS PKT.IN SINK ACK.OUT)
                    (FOLLOWS PKT.IN PKT.OUT)
                    (ALL (FAPPLY (QUOTE PMAPP) PKT.OUT)))
               (INITIAL (FLATTEN SINK)
                        (FLATTEN SOURCE))),

which we simplify, applying IN.DOMAIN.REDUCE.WITH,
ALL.FAPPLY.MPAIRP.FLATTEN, ALL.FAPPLY.FOLLOWS, and
CONSISTENT.FLATTEN.FOLLOWS, and opening up IN, DOMAIN, PMAPP,
RECEIVER.EXT, FLATTEN, SEQP, and INITIAL, to:

      (IMPLIES
          (AND (CONSISTENT (QUOTE DOM)
                           (QUOTE EQUAL)
                           (FLATTEN PKT.OUT))
               (ALL (FAPPLY (QUOTE CONTIG) PKT.OUT))
               (EQUAL (FLATTEN SOURCE)
                      (RANGE (REDUCE (QUOTE WITH)
                                     (QUOTE (1QUOTE NULL))
                                     (FLATTEN PKT.OUT))))
               (IN 0 (DOMAIN (FLATTEN PKT.IN)))
               (INITIAL (FLATTEN SINK)
```

```
                    (RANGE (CONSEC (REDUCE (QUOTE WITH)
                                           (QUOTE (1QUOTE NULL))
                                           (FLATTEN PKT.IN)))))
               (FOLLOWS PKT.IN PKT.OUT)
               (ALL (FAPPLY (QUOTE PMAPP) PKT.OUT)))
          (INITIAL (FLATTEN SINK)
                   (FLATTEN SOURCE))).

We use the above equality hypothesis by substituting:
      (RANGE (REDUCE (QUOTE WITH)
                     (QUOTE (1QUOTE NULL))
                     (FLATTEN PKT.OUT)))
for (FLATTEN SOURCE) and keeping the equality hypothesis.  The
result is:

      (IMPLIES
          (AND (CONSISTENT (QUOTE DOM)
                           (QUOTE EQUAL)
                           (FLATTEN PKT.OUT))
               (ALL (FAPPLY (QUOTE CONTIG) PKT.OUT))
               (EQUAL (FLATTEN SOURCE)
                      (RANGE (REDUCE (QUOTE WITH)
                                     (QUOTE (1QUOTE NULL))
                                     (FLATTEN PKT.OUT))))
               (IN 0 (DOMAIN (FLATTEN PKT.IN)))
               (INITIAL (FLATTEN SINK)
                        (RANGE (CONSEC (REDUCE (QUOTE WITH)
                                               (QUOTE (1QUOTE NULL))
                                               (FLATTEN PKT.IN)))))
               (FOLLOWS PKT.IN PKT.OUT)
               (ALL (FAPPLY (QUOTE PMAPP) PKT.OUT)))
          (INITIAL (FLATTEN SINK)
                   (RANGE (REDUCE (QUOTE WITH)
                                  (QUOTE (1QUOTE NULL))
                                  (FLATTEN PKT.OUT))))),

which further simplifies, applying the lemmas INITIAL.RANGE,
PMAPP.REDUCE.WITH, IN.DOMAIN.REDUCE.WITH, ALL.FAPPLY.FOLLOWS,
FOLLOWS.REDUCE.WITH, FOLLOWS.FLATTEN, ALL.FAPPLY.MPAIRP.FLATTEN,
INITIAL.CONSEC.FOLLOWS, and INITIAL.TRANS, and expanding the
function PMAPP, to:

      T.

Q.E.D.

73672 conses
81.063 seconds
20.092 seconds, garbage collection time

----------------------------------------------------------------------



Proof of VC  'SENDER#3.5.2'

     (OR (CONTIG (JOIN QUEUE (ENMAP MESS NEXT)))
```

```
         (NOT (CONTIG QUEUE))
         (NOT (PMAPP QUEUE))
         (NOT (NUMBERP NEXT))
         (EQUAL NEXT 0)
         (IF (SEQP QUEUE)
             (NOT (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT)))
             F)
         (NOT (SEQP MESS)))
```

This formula can be simplified, using the abbreviations NOT, OR, DOMAIN.JOIN, and CONTIG, to:

```
      (IMPLIES
           (AND (NOT (CONSECP (JOIN (DOMAIN QUEUE)
                                    (DOMAIN (ENMAP MESS NEXT)))))
                (CONSECP (DOMAIN QUEUE))
                (PMAPP QUEUE)
                (NUMBERP NEXT)
                (NOT (EQUAL NEXT 0))
                (NOT (IF (SEQP QUEUE)
                         (NOT (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT)))
                         F)))
           (NOT (SEQP MESS))).
```

This simplifies, expanding the definition of NOT, to two new conjectures:

```
Case 2. (IMPLIES
              (AND (NOT (CONSECP (JOIN (DOMAIN QUEUE)
                                       (DOMAIN (ENMAP MESS NEXT)))))
                   (CONSECP (DOMAIN QUEUE))
                   (PMAPP QUEUE)
                   (NUMBERP NEXT)
                   (NOT (EQUAL NEXT 0))
                   (NOT (SEQP QUEUE)))
              (NOT (SEQP MESS))),
```

which again simplifies, applying PISEQP.DOMAIN.2, JOIN.NULL.PISEQP, and CONSECP.DOMAIN.ENMAP, and opening up the function DOMAIN, to:

```
       T.

Case 1. (IMPLIES
              (AND (NOT (CONSECP (JOIN (DOMAIN QUEUE)
                                       (DOMAIN (ENMAP MESS NEXT)))))
                   (CONSECP (DOMAIN QUEUE))
                   (PMAPP QUEUE)
                   (NUMBERP NEXT)
                   (NOT (EQUAL NEXT 0))
                   (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT)))
              (NOT (SEQP MESS))),
```

which we again simplify, using linear arithmetic and applying CONSECP.DOMAIN.ENMAP, DOM.MPAIR, FIRST.ENMAP, and CONSECP.JOIN.DOMAIN, to:

```
        T.

Q.E.D.

5857 conses
6.779 seconds
0.0 seconds, garbage collection time

----------------------------------------------------------------------



Proof of VC  'SENDER#11.9.2'

     (OR (EQUAL (HIGHEST (DOMAIN (FLATTEN (APR PKT.OUT QUEUE))))
                (SUB1 NEXT))
         (NOT (EQUAL (HIGHEST (DOMAIN (FLATTEN PKT.OUT)))
                     (SUB1 NEXT)))
         (NOT (PMAPP QUEUE))
         (NOT (NUMBERP NEXT))
         (EQUAL NEXT 0)
         (IF (SEQP QUEUE)
             (NOT (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT)))
             F))

  This formula can be simplified, using the abbreviations NOT and
OR, to:

      (IMPLIES
       (AND
        (NOT (EQUAL (HIGHEST (DOMAIN (FLATTEN (APR PKT.OUT QUEUE))))
                    (SUB1 NEXT)))
        (EQUAL (HIGHEST (DOMAIN (FLATTEN PKT.OUT)))
               (SUB1 NEXT))
        (PMAPP QUEUE)
        (NUMBERP NEXT)
        (NOT (EQUAL NEXT 0)))
       (IF (SEQP QUEUE)
           (NOT (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT)))
           F)).

This simplifies, applying LST.APR, NLST.APR, DOMAIN.JOIN,
PISEQP.DOMAIN.2, HIGHEST.JOIN, and HIGHEST.DOMAIN, and expanding
the functions FLATTEN, MAX, and NOT, to:

      (IMPLIES (AND (LESSP (HIGHEST (DOMAIN (FLATTEN PKT.OUT)))
                           (DOM (LST QUEUE)))
                    (NOT (EQUAL (DOM (LST QUEUE)) (SUB1 NEXT)))
                    (EQUAL (HIGHEST (DOMAIN (FLATTEN PKT.OUT)))
                           (SUB1 NEXT))
                    (PMAPP QUEUE)
                    (NUMBERP NEXT)
                    (NOT (EQUAL NEXT 0)))
               (SEQP QUEUE)),

which again simplifies, applying LST.NSEQP, and opening up the
functions DOM, EQUAL, and LESSP, to:
```

```
      T.

Q.E.D.

7238 conses
8.073 seconds
3.328 seconds, garbage collection time

----------------------------------------------------------------



Proof of VC  'RECEIVER#7.4.2.2.1'

     (OR (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                    (NULL)
                                    (JOIN (FLATTEN PKT.IN) PKT)))
                     (REACH M)))
         (LESSP (REACH (REDUCE (QUOTE WITH)
                               (NULL)
                               (FLATTEN PKT.IN)))
                NEXT)
         (NOT (FOLLOWS QUEUE
                       (UPPER (REDUCE (QUOTE WITH)
                                      (NULL)
                                      (FLATTEN PKT.IN))
                              (ADD1 NEXT))))
         (NOT (ALL (FAPPLY (QUOTE MPAIRP)
                           (FLATTEN PKT.IN))))
         (NOT (IN 0 (DOMAIN (FLATTEN PKT.IN))))
         (NOT (LESSP 0 NEXT))
         (NOT (CONSISTENT (QUOTE DOM)
                          (QUOTE EQUAL)
                          (JOIN (FLATTEN PKT.IN) PKT)))
         (NOT (ALL (FAPPLY (QUOTE CONTIG)
                           (APR PKT.IN PKT))))
         (NOT (PMAPP QUEUE))
         (NOT (NUMBERP NEXT))
         (NOT (NUMBERP EDGE))
         (NOT (EQUAL M
                     (REDUCE (QUOTE WITH)
                             QUEUE
                             (UPPER (LOWER PKT EDGE) NEXT))))
         (LESSP (DOM (LST PKT)) NEXT)
         (LESSP NEXT (DOM (FIRST PKT)))
         (NOT (SEQP PKT))
         (LESSP EDGE NEXT)
         (NOT (PMAPP PKT)))

This formula can be simplified, using the abbreviations NOT and OR,
to:

      (IMPLIES
            (AND (LESSP (REACH (REDUCE (QUOTE WITH)
                                       (QUOTE (1QUOTE NULL))
                                       (JOIN (FLATTEN PKT.IN) PKT)))
                        (REACH M))
```

```
               (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                          (QUOTE (1QUOTE NULL))
                                          (FLATTEN PKT.IN)))
                           NEXT))
               (FOLLOWS QUEUE
                        (UPPER (REDUCE (QUOTE WITH)
                                       (QUOTE (1QUOTE NULL))
                                       (FLATTEN PKT.IN))
                               (ADD1 NEXT)))
               (ALL (FAPPLY (QUOTE MPAIRP)
                            (FLATTEN PKT.IN)))
               (IN 0 (DOMAIN (FLATTEN PKT.IN)))
               (LESSP 0 NEXT)
               (CONSISTENT (QUOTE DOM)
                           (QUOTE EQUAL)
                           (JOIN (FLATTEN PKT.IN) PKT))
               (ALL (FAPPLY (QUOTE CONTIG)
                            (APR PKT.IN PKT)))
               (PMAPP QUEUE)
               (NUMBERP NEXT)
               (NUMBERP EDGE)
               (EQUAL M
                      (REDUCE (QUOTE WITH)
                              QUEUE
                              (UPPER (LOWER PKT EDGE) NEXT)))
               (NOT (LESSP (DOM (LST PKT)) NEXT))
               (NOT (LESSP NEXT (DOM (FIRST PKT))))
               (SEQP PKT)
               (NOT (LESSP EDGE NEXT)))
          (NOT (PMAPP PKT))),
```

which simplifies, applying REDUCE.JOIN, APPLY1.CONTIG, LST.APR, and NLST.APR, and opening up the functions EQUAL, LESSP, CONTIG, FAPPLY, and ALL, to:

```
      (IMPLIES
         (AND (LESSP (REACH (REDUCE (QUOTE WITH)
                                    (REDUCE (QUOTE WITH)
                                            (QUOTE (1QUOTE NULL))
                                            (FLATTEN PKT.IN))
                                    PKT))
                     (REACH (REDUCE (QUOTE WITH)
                                    QUEUE
                                    (UPPER (LOWER PKT EDGE) NEXT))))
              (NOT (LESSP (REACH (REDUCE (QUOTE WITH)
                                         (QUOTE (1QUOTE NULL))
                                         (FLATTEN PKT.IN)))
                          NEXT))
              (FOLLOWS QUEUE
                       (UPPER (REDUCE (QUOTE WITH)
                                      (QUOTE (1QUOTE NULL))
                                      (FLATTEN PKT.IN))
                              (ADD1 NEXT)))
              (ALL (FAPPLY (QUOTE MPAIRP)
                           (FLATTEN PKT.IN)))
              (IN 0 (DOMAIN (FLATTEN PKT.IN)))
              (NOT (EQUAL NEXT 0))
```

```
            (CONSISTENT (QUOTE DOM)
                        (QUOTE EQUAL)
                        (JOIN (FLATTEN PKT.IN) PKT))
            (CONSECP (DOMAIN PKT))
            (ALL (FAPPLY (QUOTE CONTIG) PKT.IN))
            (PMAPP QUEUE)
            (NUMBERP NEXT)
            (NUMBERP EDGE)
            (NOT (LESSP (DOM (LST PKT)) NEXT))
            (NOT (LESSP NEXT (DOM (FIRST PKT))))
            (SEQP PKT)
            (NOT (LESSP EDGE NEXT)))
       (NOT (PMAPP PKT))),
```

which again simplifies, using linear arithmetic, applying the lemmas IN.DOMAIN.REDUCE.WITH, LESSP.REACH.REDUCE.WITH.2, DOM.FIRST.REDUCE.UPPER.LOWER, FOLLOWS.REDUCE.CONSISTENT.JOIN, FOLLOWS.REDUCE.WITH.SAME, CONSISTENT.DOM.EQUAL.JOIN.REDUCE, FOLLOWS.UPPER, FOLLOWS.REDUCE.REDUCE.WITH, FOLLOWS.TRANS, FOLLOWS.LOWER, FOLLOWS.SAME, PMAPP.REDUCE.WITH, ALL.FAPPLY.MPAIRP, and LESSP.REACH.REACH.2, and expanding the functions PMAPP and FOLLOWS, to:

```
      T.
```

Q.E.D.

```
156478 conses
188.374 seconds
21.06 seconds, garbage collection time
[579 cns / 1.2 s + 0.0 gc + 0.0 io (= 1  1)]
[429 cns / 1.0 s + 0.0 gc + 0.0 io (= 1  1)]
```

Bibliography

[Bochmann 77] G. V. Bochmann and J. Gecsei.
A Unified Method for the Specification and Verification of Protocols.
In *Proceedings of IFIP 77*. North Holland, 1977.

[Bochmann 80] G. V. Bochmann and C. A. Sunshine.
Formal Methods in Communication Protocol Design.
IEEE Transactions on Communications COM-28(4), April, 1980.

[Boyer 79] R. S. Boyer and J. S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Brand 78] D. Brand and W. H. Joyner.
Verification of Protocols Using Symbolic Execution.
Computer Networks 2(4/5), September, 1978.

[ChaoChen 81] Z. Chao Chen and C. A. R. Hoare.
Partial Correctness of Communicating Processes and Protocols.
Technical Monograph PRG-20, Oxford University, Computing Laboratory, Programming Research Group, May, 1981.

[Danthine 80] A. Danthine.
Protocol Representation with Finite State Models.
IEEE Transactions on Communications COM-28(4), April, 1980.

[Dershowitz 79] N. Dershowitz and Z. Manna.
Proving Termination with Multiset Orderings.
CACM 22(8), August, 1979.

[Dijkstra 75] E. W. Dijkstra.
Guarded Commands, Nondeterminacy, and Formal Derivation of Programs.
CACM 18(8), August, 1975.

[DiVito 81] B. L. DiVito.
A Mechanical Verification of the Alternating Bit Protocol.
Technical Report ICSCA-CMP-21, University of Texas at Austin, June, 1981.

[DiVito 82a] B. L. DiVito.
Verification of the Stenning Protocol.
Technical Report 26, Institute for Computing Science, University of Texas at Austin, July, 1982.

[DiVito 82b] B. L. DiVito.
Verification of TCP-like Data Transport Functions.
Technical Report 27, Institute for Computing Science, University of Texas at Austin, July, 1982.

[Floyd 67] R. W. Floyd.
Assigning Meanings to Programs.
In *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967. Vol. 19.

[Gerhart 80] S. L. Gerhart, et al.
An Overview of AFFIRM: A Specification and Verification System.
In *Proceedings IFIP 80*, pages 343-348. October, 1980.

[Good 77] D. I. Good.
Constructing Verified and Reliable Communications Processing Systems.
ACM Software Engineering Notes 2(5), October, 1977.

[Good 78] D. I. Good and R. M. Cohen.
Verifiable Communications Processing in Gypsy.
In *Proceedings of Compcon '78*. IEEE, September, 1978.

[Good 79] D. I. Good, R. M. Cohen, and J. Keeton-Williams.
Principles of Proving Concurrent Programs in Gypsy.
In *Proceedings of 6th Symposium of Principles of Programming Languages*. ACM, January, 1979.

[Good 80] D. I. Good.
The Problem with Verification is Computer Science.
ACM Software Engineering Notes 5(3), July, 1980.

[Hailpern 80] B. T. Hailpern.
Verifying Concurrent Processes Using Temporal Logic.
PhD thesis, Stanford University, 1980.
Technical Report 195, Computer Systems Laboratory.

[Hailpern 81] B. Hailpern and S. Owicki.
Modular Verification of Computer Communication Protocols.
Research Report RC 8726, IBM, Yorktown Heights, NY, March, 1981.

[Hoare 69] C. A. R. Hoare.
An Axiomatic Basis for Computer Programming.
CACM 12(10), October, 1969.

[Hoare 78] C. A. R. Hoare.
Communicating Sequential Processes.
CACM 21(8), August, 1978.

[Hoare 81] C. A. R. Hoare.
A Calculus of Total Correctness for Communicating Processes.
Technical Monograph PRG-23, Oxford University, Computing Laboratory, Programming Research Group, April, 1981.

[Howard 76] J. Howard.
Proving Monitors Correct.
CACM 19(5), May, 1976.

[King 69] P. J. H. King.
The Interpretation of Limited Entry Decision Table Format and Relationships among Conditions.
Computer Journal 12, November, 1969.

[Knuth 70] D. E. Knuth and P. B. Bendix.
Simple Word Problems in Universal Algebras.
In *Computational Problems in Abstract Algebra*. Pergamon Press, New York, 1970.

[Kroghdal 78] S. Kroghdal.
Verification of a Class of Link-level Protocols.
BIT 18, 1978.

[Lam 82] S. S. Lam and A. U. Shankar.
Verification of Communication Protocols via Protocol Projections.
In *Proceedings of INFOCOM '82*. IEEE, 1982.

[Lamport 80] L. Lamport.
'Sometime' is Sometimes 'Not Never'.
In *Proceedings of 7th ACM Symposium on Principles of Programming Languages*. ACM, January, 1980.

[LeMoli 73] G. LeMoli.
A Theory of Colloquies.
Alta Frequenza Vol. 42:493-500, 1973.

[McQuillan 78] J. M. McQuillan and V. G. Cerf.
Tutorial: A Practical View of Computer Communications Protocols.
IEEE Computer Society, 1978.

[Merlin 79] P. M. Merlin.
Specification and Validation of Protocols.
IEEE Transactions on Communications COM-27(11), November, 1979.

[Metzner 77] J. R. Metzner and B. H. Barnes.
Decision Table Languages and Systems.
Academic Press, New York, 1977.

[Misra 81] J. Misra and K. M. Chandy.
Proofs of Networks of Processes.
IEEE Transactions on Software Engineering SE-7(4), July, 1981.

[Musser 80] D. R. Musser.
Abstract Data Type Specification in the AFFIRM System.
IEEE Transactions on Software Engineering SE-6(1), January, 1980.

[Owicki 75] S. S. Owicki.
Axiomatic Proof Techniques for Parallel Programs.
PhD thesis, Cornell University, Ithaca, N.Y., August, 1975.

[Owicki 76] S. S. Owicki and D. Gries.
Verifying Properties of Parallel Programs: An Axiomatic Approach.
CACM 19(5), May, 1976.

[Pnueli 77] A. Pnueli.
The Temporal Logic of Programs.
In *Proceedings of 18th Symposium on the Foundations of Computer Science*. ACM, October, 1977.

[Pooch 74] U. W. Pooch.
Translation of Decision Tables.
Computing Surveys 6, June, 1974.

[Postel 76] J. Postel and D. Farber.
Graph Modeling of Computer Communications Protocols.
In *Proceedings of the Fifth Texas Conference on Computing Systems*. Austin, Texas, 1976.

[Postel 80] J. Postel, editor.
DoD Standard Transmission Control Protocol.
ACM SIGCOMM 10(4), October, 1980.

[Rudin 78] H. Rudin, C. H. West and P. Zafiropulo.
Automated Protocol Validation: One Chain of Development.
Computer Networks 2(4/5), September, 1978.

[Schwabe 81a] D. Schwabe.
Formal Specification and Verification of a Connection Establishment Protocol.
In *Proceedings of Seventh Data Communications Symposium*. IEEE, October, 1981.

[Schwabe 81b] D. Schwabe.
Formal Techniques for the Specification and Verification of Protocols.
PhD thesis, University of California, Los Angeles, 1981.
Technical Report CSD-810401, Computer Science Department.

[Schwartz 81] R. L. Schwartz and P. M. Melliar-Smith.
Temporal Logic Specification of Distributed Systems.
In *Proceedings of the Second International Conference on Distributed Systems*. INRIA, Paris, France, April, 1981.

[Stenning 76] N. V. Stenning.
A Data Transfer Protocol.
Computer Networks 1(2), September, 1976.

[Sunshine 79] C. A. Sunshine.
Formal Methods for Protocol Specification and Verification.
Computer 12(9), September, 1979.

[Sunshine 81] C. A. Sunshine, editor.
Communication Protocol Modeling.
Artech House, Dedham, Mass., 1981.

[Thompson 81] D. H. Thompson, C. A. Sunshine, R. W. Erikson, S. L. Gerhart, and D. Schwabe.
Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models.
Technical Report ISI/RR-81-88, USC/Information Sciences Institute, March, 1981.

vita

Benedetto Lorenzo DiVito was born in Detroit, Michigan, on December 31, 1951, the son of Antonio Michele DiVito and Maria Teresa DiVito. He graduated from St. Mary of Redford High School, Detroit, Michigan in 1970. From September 1970 to May 1974, he attended the University of Michigan in Ann Arbor, where he earned the Bachelor of Science degree in computer engineering. In August 1974, he was married to Lynn Havins. In 1975, he received the Master of Science degree in computer, information and control engineering, also from the University of Michigan. He was employed by ITT Telecommunications of Des Plaines, Illinois from July 1975 to February 1977, where he performed software engineering work. Subsequently, he worked on software development for Texas Instruments in Austin, Texas until August 1978. In September 1978, he entered the Graduate School of The University of Texas at Austin. During the 1978-1979 academic year, he was a teaching assistant in the Computer Science Department. Since June 1979, he has worked as a research assistant in the Institute for Computing Science at The University of Texas at Austin.

VERIFICATION OF COMMUNICATIONS PROTOCOLS AND ABSTRACT PROCESS MODELS

Table of Contents

List of Figures

List of Tables