(12)

ISI/RR-81-98

December 1981

AD A111744

Bernard Berthomieu

Algebraic Specification of Communication Protocols

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA ISI 4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

8

82 03 08 011

Bernard Berthomieu   Algebraic Specification of Communication Protocols   ISI/RR-81-98

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-81-98	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Algebraic Specification of Communication Protocols		5. TYPE OF REPORT & PERIOD COVERED Research Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bernard Berthomieu		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 03(
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order # 2223
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 50
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release and sale: distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

....

18. SUPPLEMENTARY NOTES

The author's current address is: Centre National de la Recherche Scientifique
Laboratoire d'Automatique et d'Analyse des Systemes
7, Avenue du Colonel Roche, 31400 Toulouse, France

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

abstract data types, axiomatic algebraic specifications, specification and verification of communication protocols, experience with natural deduction theorem proving, Affirm

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. ABSTRACT

This report is concerned with specifying communication protocols and proving their behavior correct. Many different approaches have been taken for modeling communication protocols; the approach taken here is algebraic. Communication protocols are seen as complex data types and are defined using the algebraic method. The modeling techniques devised are applied to the specification and verification of a complex data transfer protocol. Properties proven include system invariants and those of an operational nature. These experiments have been carried out using the Affirm system for designing abstract data types algebraically and proving their properties.

This report is also being published by Centre National de la Recherche Scientifique. Laboratcire d'Automatique et d'Analyse des Systemes as Note Technique 81 T 26.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ISI/RR-81-98
December 1981

Bernard Berthomieu

Algebraic Specification of Communication Protocols

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

This research is supported by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

CONTENTS

ACKNOWLEDGMENTS

Many discussions with Jonathan Postel, Susan Gerhart, Carl Sunshine, Daniel Schwabe, Rod Erickson, and David Thompson have certainly helped clarify the ideas presented here. Special thanks are due to Jon Postel for making possible my stay at ISI.

1. INTRODUCTION

As elaborate communication systems continue to proliferate, there is an increased need for methods and tools to assist in their design. The availability of powerful means of communication, such as satellites, makes it possible to envision more sharing of geographically distributed computer resources. Successful sharing of these resources depends on high-performance and reliable communication software systems.

Many different approaches have been taken for specifying communication protocols. These approaches include programming languages, transition models, and various more specialized techniques. This report will discuss other techniques, based on the concepts of data abstractions (or abstract data types).

Modularity and self-containment of the specifications are key in designing reliable software systems. The concept of abstract data types provides a consistent theoretical foundation for these ideas. Recently available tools, such as *Affirm*, greatly facilitate the use of these specification techniques. The purpose of this report is to show how the design of communication protocols can benefit from these advances.

The work presented here is part of an effort involving several projects at the Information Sciences Institute for developing formal methods for protocol verification. Various aspects of algebraic specification techniques for communication protocols have been considered in [TSEGS 81], [Schw 81-1], and [Bert 80-2]. We will make these more precise in this report.

What is meant by proving correct the behavior of a communication protocol is explained in the following section. Section 3 gives an overview of the algebraic technique for defining abstract data types and briefly discusses the main features of the *Affirm* system. The algebraic technique will be applied to the specification of communication protocols in Section 4, where it is illustrated with a trivial data transfer protocol. A more significant complex protocol will be treated in Section 5.

2. VERIFYING COMMUNICATION PROTOCOLS

2.1 Layers, Protocols, Abstract Specifications

It is well known that communication systems may be structured as a hierarchy of *protocol layers*, each layer providing a set of *services* to the users of the layer above [BoSu 80]. To specify the service of a protocol at a given layer is to describe the input/output behavior of the layer; this usually consists of defining a certain number of service primitives which abstractly describe the functions provided by the protocol of that layer. For a transport service, for instance, this set of primitives may be Connect, Disconnect, Send, and Receive.

Each layer of the communication system comprises a number of communicating entities. A protocol at a given layer may be seen as the set of rules governing the communication between entities that provide the service of the layer. Specifying a protocol at a layer describes how the entities involved respond to the commands from the layer above and communicate using the service provided by the layer below.

The designer need not be concerned at the specification level with a particular implementation of the protocol but rather with an abstract view of the functions it performs. Specifications should capture only the essential mechanisms of the protocol.

The next section clarifies what we mean by protocol verification and discusses the verification techniques available.

2.2 Correctness Issues

Verification of communication protocols has several aspects. The goal may be to prove that the behavior of the protocol is correct, or to verify that a protocol provides the intended service, or to prove that an implementation of the protocol is correct. This section makes these different goals more precise.

Functional aspects

Verification of the functional aspects of a protocol is carried out on the abstract specifications. It involves verifying that these specifications satisfy a certain number of properties characterizing the correct behavior of the protocol. These properties may serve various purposes:

- The designer of a protocol has an intuitive understanding of what is required of that protocol. After these requirements are translated into assertions written in the same language as the specifications, proving that the behavior of the protocol is correct verifies that the specification of the protocol satisfies these properties.
- The properties a protocol must satisfy may also represent constraints in using the protocol as part of a complex communication system. In a stepwise-refinement methodology, for instance, assumptions may have been made about the behavior of a protocol when the upper layers of the communication system were specified; the protocol must satisfy these assumptions.

From a formal point of view, this set of desirable properties generally may be split into *static* and *dynamic* properties.

Static properties (or Safety properties)

These properties ensure that no bad behavior occurs: If the system does something, does it do what it is supposed to do? They are not concerned with whether the system actually does anything.

Static properties are most often specific to the application under consideration. They are expressed as assertions written in the same language as the specifications, and they characterize the set of values (or global states) of the protocol that may be considered correct.

The following is an example of a static property for a data transfer protocol: The set of messages an entity can receive must be a subset of the set of messages that were sent to it. This property says that if there is a transfer of messages between entities, then this transfer is correct in that messages received are among the set of messages sent; but it does not suppose that a transfer actually occurs.

Note that the above characterization of static properties does not suppose an operational semantic for the specifications. (We will see in Section 4 that axiomatic specifications, for instance, have no obvious operational semantics.) If an operational semantic is defined for the specification, static properties may also address certain operational aspects.

An example of a static and operational property is the absence of undesired deadlocks. A deadlock (or total block) is a state of the protocol that permits no further operation. For a given protocol, some deadlocks may be acceptable, but some others not. This property is specific to that protocol: It is static in that it does not imply the existence of deadlocks (but rather says that if there are some, then they are acceptable terminating values); it is operational since characterizing a deadlock needs some operational interpretation of the specification.

Dynamic properties (or Liveness properties)

These properties require that something happen during the execution of the protocol. They suppose an operational interpretation of the specifications, and they concern the computational aspects of the system: Is the system really doing something; will it stop after completing its tasks? Dynamic properties may generally be formulated using the concept of state reachability.

The absence of partial blocking (i.e., starvation of an entity) and the termination property (only a finite number of computational steps are possible from the initial value) are dynamic properties.

Implementation aspects

Here, verification involves checking the adequacy of an implementation of an abstract specification. For our purposes this means either verifying that a protocol provides a given service or proving that an implementation of the primitives of a protocol by a set of programs is adequate. Stated more formally, two specifications, an abstract one and a more detailed one, must be proved compatible, i.e., the detailed one satisfies all the properties the abstract one satisfies.

Comparing two specifications requires that they be expressed in the same language. The usual first step for these verifications is thus to exhibit a correspondence between the functions and data that appear in the abstract specification and those that appear in the implementation. Such a correspondence is often called a "representation function." Then, the implementation is correct relative to the abstract specification if the representation of any theorem of the abstract specification (via the representation function) is a theorem of the implementation. If the abstract specification consists of a set of axioms, then it is sufficient to show that the representation of every axiom of the abstract specification is a theorem of the implementation.

Imple :entation is an important issue in protocol design; the specification methods that will be introduced later are suitable for expressing and verifying implementations. These issues are discussed at length in [TSEGS 81] and will not be addressed here where the focus is the verification of functional properties of protocols.

2.3 Formal Methods for Specifying Communication Protocols

Verifying the properties discussed above implies the use of a rigorous formalism for expressing the specifications. Several formalisms have been used for that purpose, and the most widely used are discussed in the following where they are designated as *linguistic models*, *transition models*, and *mixed models*. The principle and main advantages and limitations for each are briefly explained.

Linguistic models

These methods use specification languages, usually close to regular programming languages, for expressing protocols. A typical example of the application of this technique is [Sten 76], in which a data transfer protocol is specified and some static properties are manually proven.

The verification techniques used with these specification methods are the usual assertion techniques for program verification [Floy 67] [Hoar 69], extended to parallel programs [Lamp 77], since protocols are usually specified as sets of programs running concurrently. These methods permit compact expression of rather complex communication systems, and the specification they produce can be easily read (everybody knows at least one programming language or is familiar with programming).

The main criticism of using programming languages for specifying is that they usually do not permit complex data structures and behaviors to be expressed abstractly enough. When applied to protocols, these languages must be at a level of detail too low to be suitable for good comprehension and utilization of the specifications. Furthermore, the verification of such specifications is often laborious due to both the low level of abstraction of the languages used and the inherent difficulty of proving parallel programs.

Transition models

With transition models [Boch 78] [AUYa 79] [Merl 79] [Dant 80], each entity involved in the communication system is represented as a machine that stands in one of a number of states. Execution of primitives or occurrences of events are associated with state transitions of the entities. The interrelation between entities is expressed separately [Boch 78] or using the same formalism [Dant 80].

Analysis of the properties of a system represented with such models is carried out using the existing methods for the analysis of Petri nets or equivalent models. Several kinds of methods may be used: enumeration methods (the set of composite states of the system is enumerated and analyzed using the state reachability relation) or structural methods [LaSc 74] [ABDe 80] (using integer programming or graph analysis techniques). These methods make it possible to check virtually any property of the system we wish to verify, including absence of blockings and liveness properties: this is their main advantage.

The limitations of transition models are mainly the consequences of the poor kind of abstraction that they allow to be expressed. The apparent simplicity of the representations derived with such models is often a result of designers only representing the control structure of the system, neglecting the data transmission aspects. This simplicity is thus generally obtained at the expense of the quantity of information put into the model. In fact, if complex mechanisms are represented, such as the management of the sequence numbers in a transport protocol, these models often become unmanageable either because of their size or because of the size of their state space. This is a strong limitation, since we need formalism and automated verification techniques mainly to express (and to verify the correctness of) complex mechanisms.

Mixed models

Several remedies have been investigated to augment the possibilities of abstraction of the transition models; among them are macro-nets and mixed models.

Macro-nets are built by enriching the graphic language of transition models with specific primitives such as emission or diffusion of messages [Nutt 72] or with symbols representing complex data structures, such as queues. Other studies, without augmenting the description language, seek to express complex systems as interconnections of simpler systems, easy to analyze separately, and to devise methodologies for connecting them while preserving the properties of the whole system [PoFa 76] [AABe 78].

Mixed models have two parts: a transition model that represents the control part of the system, and an interpretation of the transitions over a set of operators working on some data structures. The interpretation usually consists of associating with every transition an enabling predicate and an operator. The predicate is determined by the current value of the data, and the operator expresses the transformation carried out on the data when the transition is activated.

Macro-nets and mixed models have been widely used for specifying certain kinds of mechanisms. Nutt nets, for instance, are very popular for representing update algorithms for distributed data bases. Mixed models have been used for representing communication protocols [Boch 77] [RaEs 80] and, more generally, as a model for concurrent systems [Kell 76] [Vale 77].

These extensions of transition models have brought some improvements in the specification phase, but their limitations arise when verification is attempted. In both formalisms, the possibility of reachability analysis of the "pure" transition models is lost. Little progress has been made in devising specific analysis techniques for macro-nets; assertion techniques for parallel program proving are difficult to apply to the verification of mixed models because they a priori allow the expression of a complex control structure.

The purpose of this report is to show that other specification methods may be used to bring about a slight improvement over the methods already described. They are based on the formalism of abstract data types. Section 3 presents the necessary background material for defining abstract data types algebraically, as can be done using *Affirm*; specific methods for modeling communication protocols will be presented in Section 4 and applied in Section 5.

3. ALGEBRAIC SPECIFICATION OF ABSTRACT DATA TYPES

3.1 Abstract Data Types as Algebras

A data type may be seen as a collection of objects and a set of operations over it [GHMu 78]. Its specification is axiomatic when the effects of the operations are expressed by axioms interrelating the objects created by these operations. It is algebraic when these axioms have the form of equations.

Data types are abstract in that we are not concerned with a particular implementation of the objects (e.g., integers as represented in some machines or stacks as implemented by arrays), but instead with their abstract, logical nature.

Liskov and Zilles [LiZi 75] discuss several methods for defining abstract data types. Here only the "algebraic axiomatic" method is considered. With this method, properties of the operators are expressed by axioms in the form of equations. Data types may then be seen as algebras. For a complete statement of the algebraic method for specifying abstract data types, see [GuHo 78] [GTWa 78] [GHMu 78] [Muss 80-1]. Familiarity with the concepts and methods developed in [GHMu 78] (at least) is assumed in what follows.

Synthetically, defining an abstract data type defines a first order theory. A theory consists of a language, a set of axioms, and a set of inference rules. The language is defined once we provide symbols for domains, variables, and operators, and define the syntax and typing of the operators and the typing of the variables. Here axioms are equations, and inference rules are substitution and replacement of equals. Interpretations of these theories are algebras (in this case, many sorted algebras).

Issues in these theories are logical consistency, validity, and completeness. These are discussed at length in the literature.

3.2 The *Affirm* System for Creating Abstract Data Types and Proving Their Properties

Presentation

The *Affirm* system developed by the Program Verification project at Information Sciences Institute includes:

- A language that permits the definition of axiomatic abstract data types. These user-defined data types may be used in programs written in a language close to Pascal.
- A natural deduction theorem prover that enables proving either properties of the defined data types or verification conditions for the programs.

It is out of the scope of this report to detail the characteristics of this data type manipulation system. However, a brief summary of the main commands is given in Appendix 1; it explains briefly the type definition commands and the proof development commands needed for a good understanding of the experiments that will be presented later. Musser [Muss 80-1], Gerhart [Gerh 80], and Thompson [TGELB 81] should provide any necessary additional information.

The next few paragraphs discuss some issues concerning algebraic specification of data types in general and with *Affirm* in particular. These comments should provide help in understanding how these methods are used in the following sections.

Defining types in Affirm

The main characteristic of *Affirm* is that axioms defining the operators of the types have the form of rewrite rules. The relationships between term-rewriting systems and algebras are discussed in [HuOp 80]; rewriting rules may be seen as "oriented equations." These rules have the following form:

term1 = = term2;

where term1 and term2 are symbolic terms of the same type. The symbol = = is the rewriting symbol. This rule is interpreted as follows: any term matching term1 is to be replaced by term2.

To allow a certain kind of conditional results, an **if-then-else** operator whose result is of the type being currently defined is part of the definition of every type. It is used in the following fashion:

term1 = = **if** cond **then** term2 **else** term3;

where term1, term2, and term3 are terms of the type being defined and cond is a boolean term.

If-then-elses can appear only on the right-hand side of the rewrite rules. Furthermore, they obey the following axioms (which never appear in the print-outs of the types):

if TRUE **then** term2 **else** term3 = = term2;

if FALSE **then** term2 **else** term3 = = term3;

Affirm allows users to interactively define its private data types. Besides syntax verifications (number of arguments and types of the operators, as in their interface declarations), the system offers some more specific help when the user is entering axioms; these aids are described below.

Aids from Affirm while specifying types

The two key concepts in term-rewriting systems are the *finite termination* property (any term may be rewritten only a finite number of times) and the *unique termination* property (if the rewriting terminates, then identical terms are obtained whatever the ordering in which the rewrite rules have been applied).

When a set of rewrite rules possesses these two properties (it is then sometimes called *convergent*), it may be shown that "rewriting reasoning" is complete [KnBe 69]. That is, two terms are equal (modulo the equality relation expressed by the set of rules) if and only if they reduce to the same "normal form."

The finite termination property is known to be undecidable; the unique termination property may be verified for finitely terminating term-rewriting systems. The system therefore cannot assure convergence when new axioms are entered, but it does something close to that in practice:

- When a new axiom is introduced, it is first checked to ensure that it does not possess some property that trivially implies nonfinite termination. If it does not satisfy any such property, the finite termination property is assumed for the set of rules augmented with the axiom. Some of these properties are easy to state and to check, and it is unlikely in practice that a user would state an axiom that could destroy the finite termination and not satisfy one of these sufficient conditions.
- Once the finite termination is assumed, *Affirm* proceeds to the verification of the unique termination. This procedure may cause the system itself to introduce new axioms for restoring the unique termination property, in case the entered axiom destroyed it. This procedure is based on the algorithm devised by Knuth and Bendix [KnBe 69]. Addition of new rules by the system is carried out interactively in *Affirm* (two terms are proposed and the rewriting direction is given by the user); this process was automatic originally but required definition of an ordering for the terms.

A last verification provided is logical consistency: the last axiom introduced by the user is discarded if an inconsistency is discovered during the verification of unique termination.

Proof theory, generators, and structural induction

A binary operator of Boolean result, denoted as the equality, is defined for each type. The following axiom is implicitly part of the definition of every type:

x=x == TRUE;

where x is a variable declared to be of the type being defined and TRUE is a constant of the type Boolean.

Propositions are, in the *Affirm* context, Skolemized terms of type Boolean. Proving a proposition involves reducing it to the constant TRUE of type Boolean, using for that purpose the rewrite rules entered as axioms and specific inference rules called induction schemas, (which are part of the definition of the type to which they apply and are used in proofs on explicit demand from the user).

Theorem proving in *Affirm* is interactive. The system does the laborious part of the proofs (substitutions, proof context management, type checking, etc.), but the user has to develop the proof. A set of proof development commands allows the user to apply various commands for splitting the proposition into several subgoals, adding clauses to the proposition, applying previously proven lemmas, applying specific induction schemas, and substituting equalities (some of these commands are briefly documented in Appendix 1). Some predefined proof strategies, discussed in [Muss 80-2], may also be invoked.

We said previously that "rewriting reasoning" is complete for convergent sets of rewrite rules. But such a nice property may not always be achieved in practice, and additional inference rules, specific to each type, must often be used in proofs. We explain below how one can derive such rules.

Let us call all operators whose result is of type T *generators* of type T, and all operators without argument whose result is of type T *constants* of type T. (The definition of a constant is actually more complex, but this approximation will suffice for giving an intuitive understanding of structural induction.) A proof by structural induction of a property P(x) (x being a variable of type T) will consist of proving the conjunction of the following properties:

P(C), for all constants C of type T;

P(x) implies P(Gen(x)), for all generators Gen of type T which are not constants.

In practice, the set of generators we must consider as part of an induction proof may be reduced to a set of "fundamental generators" (sometimes termed "constructors"). This set can be characterized as follows: it is the smallest set of generators such that all terms of type T may be shown equal to a term containing only generators of that set.

A complete justification of these induction schemas should make use of the *sufficient completeness* property for axiomatization presented in [GuHo 78]. Examples of specific induction schemas will be presented later with examples of type specifications.

An example of a type definition in Affirm

Appendix 2 contains the definition of several data types. The second part of Appendix 2 contains the definition of the type *SequenceOfElemtype* from the *Affirm* type library. The *sequence* structure is fairly general, and the structures of *stack* or *queue* may be seen as restricted *sequences*.

3.3 Building Complex Data Type Theories

Though no general method encompasses the construction of arbitrarily complex theories, some theory building techniques are worth having; they let the user build complex specifications in a more structured way. Some of them are discussed here together with some additional issues of algebraic specifications; most ideas about structured theory building are taken from [BuGo 77].

Hierarchy of theories

We may distinguish between several layers of theories, for instance, the theory of *integers* and the theory of *stacks of integers*. Though the former is formally included in the latter, it is convenient to build the theory of *integers* separately as we may wish to use it later as a subtheory of another complex theory, such as *set of integers* or *queue of integers*.

An operator belongs to the theory of a type, say T, if at least one of its arguments or its result is of type T. It may happen that not all generators of a type are defined in its theory. The length of a *sequence*, for instance, is of type *integer* but will be defined within type *sequence*.

Parameterized types, instantiations, and replications of types

Though distinct, several theories may share the same structure; we may define, for instance, the theories *stack of integers* and *stack of messages*. Though their components are distinct (assuming that types *integer* and *message* are distinct types), these theories will share the same *stack* behavior. The *stack* structure could be defined more abstractly as the parameterized type *stack of elements*, where *element* is a nondefined, arbitrarily named type. Building the type *stack of integers* would then consist of instantiating *elements* by the type *integer* in the *stack of elements* theory.

Construction of explicitly parameterized types is not yet allowed in the *Affirm* system. This would require all "low level" types to be defined when defining a type using them (it is a scheduled improvement). These constructions may, however, be mimicked with sufficient rigorousness.

Replication of types is another way of building types that share the same structure. It is just a replication of their theories, since they have the same forms but are different instances.

Enrichment of theories

Several theories may share some subtheory but ultimately be different. For instance, the types *sequence of integers and sequence of messages* share the basic *sequence* operators and behavior, but the type *sequence of integers* may include some specific generators such as the sum or difference of two sequences.

Assuming that we have the parameterized type *sequence of elements*, the type *sequence of integers* may be built by first instantiating the type *sequence of elements* as *sequence of integers* and then enriching this last theory with the specific operators.

Hidden functions, exception handling

It has been shown that operators not explicitly needed must sometimes be introduced for expressing the behavior of needed operators with only a finite number of equations. These operators are often referred to as *hidden* or *auxiliary* operators. Besides this theoretical reason, auxiliary operators may also be introduced simply as abbreviations, to simplify the expression of the needed operators.

Not every application of an operator to its arguments may give a meaningful result (for instance, extracting the first element of an empty *queue*). If such applications may occur, then we must consider them, for example by including error specifications and error handling in the axiomatic specification, as done in [GHMu 78] and [Gogu 77].

Another way to proceed is to leave such erroneous terms unspecified. This should not cause any trouble other than affecting the completeness of the defined theory. That is, if we leave some terms unspecified, we will surely have propositions that we will not be able to show TRUE or FALSE. This is the choice made in this report.

4. ALGEBRAIC SPECIFICATION OF COMMUNICATION PROTOCOLS

4.1 Communication Protocols as Complex Data Types

In this discussion, protocols are viewed as complex data structures. The initial data (i.e., the constants, of which we will usually define only one) and the data resulting from application of the operators will constitute the set of values, or global states, of the type protocol.

These global states may include the current contents of the data storage components of the system as well as some information about the history of the computation. The usual concepts of control and data are not part of the specification language, but may be expressed within this global state notion.

Most of the time protocols will have a structure of type record (in the sense of a record in Pascal). Various selection functions will permit the different fields of the elements of type protocol to be extracted. We will not usually define an explicit constructor for the record, in order to allow the specifications to be enriched by adding new selectors.

The first step in specifying a protocol as a data type is to devise a set of *selectors* and a set of *generators* for it:

- *Selectors* may be seen as value-returning functions. Selectors do not modify the values of the protocol, which are known only through the set of selectors.
- *Generators* modify the current value or create values. They correspond to the primitives of the protocol; one of these generators will be a constant that defines an "initial value" for the protocol.

The behavior of the protocols will be expressed by a system of symbolic identities, much like arithmetical identities. These equations are axioms, rule lemmas, or definitions in the *Affirm* context, and will be treated as left-to-right rewrite rules. These axioms, besides defining an equivalence relation, are thus also "simplification rules" for the terms.

One expression is derived for each selector/generator pair. Each expression defines the symbolic value of the particular field of the global state that the selector addresses, after application of the generator. These expressions will have one of the following general forms:

For all constant generator Const:

```
    IthSelector(Const) == IthSelectorInitialValue;
```

For all nonconstant generators:

```
    IthSelector(JthGenerator(p)) == if SomeCondition(p)
                    then NewExpressionOfIthSelector(p)
                    else IthSelector(p);
```

where p denotes the current value of the protocol.

Though all axioms have one of these forms, we will simplify the expressions for which NewExpressionOfIthSelector(p) is identical to IthSelector(p). These "no change" axioms have the simple *form:*

```
    IthSelector(JthGenerator(p)) == IthSelector(p);
```

An alternative method for specifying record data types is to employ an explicit constructor function for the record and to give one axiom per generator instead of one per selector/generator pair. This axiom would express in a unique equation the transformations that the generator induces on all the selectors. We believe the former method (of individually expressing the transformations on the selectors) is preferable since, although many more axioms are needed, they have a simpler structure and lead to more readable specifications.

Protocol specifications must include specific induction schemas which have the following general form:

```
    ProtocolInduction(p) = =
        cases(      Prop(1stConst),
                          • • •
                    Prop(MthConst),
                    all pp(Prop(pp) imp Prop(1stGenerator(pp))),
                          • • •
                    all pp(Prop(pp) imp Prop(NthGenerator(pp))) );
```

where Const denotes the constants of the protocol (generally, only one will be defined), Generators are all the nonconstant generators, and Prop denotes the proposition to be proved by induction.

4.2 The *SimpleProtocol* Example

SimpleProtocol is a very simple data transfer protocol that will be used as an illustrative example for the specification and verification methods we demonstrate in this section. A more significant, complex data transfer protocol will be treated in Section 5.

SimpleProtocol transfers a sequence of messages from a fixed sender to a fixed receiver over a perfect transmission medium.

It has three selector functions defined:

ToSend, which denotes the sequence of messages not yet sent.

Transit, which gives the sequence of messages held by the transmission medium.

Received, which holds the sequence of messages received by the receiving entity.

Three generators represent the primitives of the protocol:

- *InitialProtocol* is a constant of the type. Its meaning is the "initial value" of the protocol, before transfer is executed. InitialProtocol initializes the field *ToSend* with a (symbolic) constant sequence of messages and initializes the fields *Transit* and *Received* with empty sequences of messages.
- *Send* sends a message over the transmission medium for delivery to the receiving entity; if there are still messages to be sent, it picks the first and gives it to the transmission medium for delivery to the receiver.
- *Receive* accepts messages from the transmission medium; if some messages are in transit, it picks the first and appends it to the received messages.

The axioms must express the above meanings as transformations of the values of the selectors after each generator is applied.

In order to prove the propositions that assert the correctness of the protocol, an induction schema must be included in the specification. It states that proving a property Prop(p) of the protocol is equivalent to proving the following:

```
            Prop(InitialProtocol)
    and     (Prop(p) imp Prop(Send(p)))
    and     (Prop(p) imp Prop(Receive(p)));
```

where p denotes a variable of type protocol.

The protocol is formally defined as the type *SimpleProtocol* and is built upon the types *Boolean*, *Integer*, *Message*, and *SequenceOfMessage*. The first two are built-in types of the *Affirm* system; the construction of the last two is explained in Appendix 2.

The Boolean operators PreSend and PreReceive appear in the specification only as abbreviations. The formal definition of the type *SimpleProtocol*, as submitted to *Affirm*, is the following:

```
type SimpleProtocol;
   needs types Message, SequenceOfMessage;
   declare dummy, p, pp: SimpleProtocol;
   interfaces ToSend(p), Transit(p), Received(p), InitialSequenceOfMessage : SequenceOfMessage;
   interfaces InitialProtocol, Send(p), Receive(p): SimpleProtocol;
   interfaces PreSend(p), PreReceive(p), SimpleProtocolInduction(p): Boolean;
   axioms
      dummy=dummy == TRUE,
      PreSend(p) == ToSend(p) ~= NewSequenceOfMessage,
      PreReceive(p) == Transit(p) ~= NewSequenceOfMessage;
   axioms
      ToSend(InitialProtocol) == InitialSequenceOfMessage,
      ToSend(Send(p)) == if PreSend(p)
          then LessFirst(ToSend(p))
          else ToSend(p),
      ToSend(Receive(p)) == ToSend(p);
   axioms
      Transit(InitialProtocol) == NewSequenceOfMessage,
      Transit(Send(p)) == if PreSend(p)
          then Transit(p) apr First(ToSend(p))
          else Transit(p),
      Transit(Receive(p)) == if PreReceive(p)
          then LessFirst(Transit(p))
          else Transit(p);
   axioms
      Received(InitialProtocol) == NewSequenceOfMessage,
      Received(Send(p)) == Received(p),
      Received(Receive(p)) == if PreReceive(p)
          then Received(p) apr First(Transit(p))
          else Received(p);
   schema
      SimpleProtocolInduction(pp) == cases(
          Prop(InitialProtocol),
          all p (Prop(p) imp Prop(Send(p))),
          all p (Prop(p) imp Prop(Receive(p))) );
end {SimpleProtocol};
```

4.3 Proof of Correctness (Nonoperational Aspects)

In Section 2 we distinguished two classes of functional properties: static and dynamic. The following discusses the expression and proof of static properties that do not involve operational aspects (or progress in the computation).

It is assumed that the system represented has to satisfy a set of assertions characterizing the class of functionally acceptable values of the protocol. (As discussed in Section 2, these assertions may be stated by the designer or may represent assumptions made about the behavior of the protocol at higher specification levels.) In practical cases, only particular aspects of the system are of some functional value, so the designer wants to leave a wide latitude for implementation by expressing the requirements of the system at an abstract level. Correctness assertions should capture only the fundamental functions of the system.

For the *SimpleProtocol* protocol presented previously, these properties should specify that the data transfer, if done, be done correctly. That is, the sequence of messages received is always an initial subsequence of the sequence of messages that were initially to be sent.

This may be written as follows:

> **theorem** TransferCorrect,
> Received(p) = Initial(InitialSequenceOfMessage,Length(Received(p)));

(Initial(s,k) is an operator of type *SequenceOfMessage* that gives the initial subsequence of length k of sequence s.)

This assertion may be verified using the properties of the type *SequenceOfMessage* and the following lemma:

> **theorem** TransferLemma,
> (Received(p) join Transit(p) join ToSend(p)) = InitialSequenceOfMessage;

(The join operator is the concatenation of two sequences of messages.)

TransferLemma implies that no messages are lost or added during the transfer and that they are kept in the sequence constituted by the concatenation of sequences Received, Transit, and ToSend, in the same order they had in the sequence InitialSequenceOfMessage. This lemma may be proven easily by using the induction schema specific to the type *SimpleProtocol* and the properties of the type *SequenceOfMessage.*

The proof transcripts of these theorems may be found in Appendix 3.

4.4 Enabling Conditions and Operational Interpretation

In Section 2 it was mentioned that desirable properties of communication protocols should include certain properties, static or dynamic, of an operational nature. We have not yet given any explicit operational interpretation to our axiomatic specifications; to do so is the purpose of what follows.

It is clear that the rewrite rules in the specifications may be used to compute. We will show here how the set of rewrite rules that constitutes the definition of the type *SimpleProtocol* may be seen as an algorithm that computes a final value of the protocol from the initial value InitialProtocol, by using the generators Send and Receive applied in any order.

For that final value to exist, a necessary condition is that its computation stops; this is not the case if the function we associated with the specification is defined as it is in the statement above. The reason is that all generators may always be applied from all values of the protocol, since in rewriting systems, no conditions may be stated for the application of rewrite rules (the only and implicit condition is "pattern matching"). To express conditional application of the generators, we have to make enabling conditions explicit and give an operational interpretation for the specification outside the axiomatic definition.

Let us first note that one can always find predicates PreGen such that all generators Gen of the *SimpleProtocol*, except the constant InitialProtocol, may be written as follows:

> Gen(p) = = if PreGen(p) then NewValue(p) else p;

where p denotes a variable of type *SimpleProtocol*, and NewValue(p) the new symbolic value of the protocol after application of the generator Gen when condition PreGen is satisfied by p.

This shows that nonconstant generators have an effect on the current value of the protocol only when a certain condition, specific for each generator, is satisfied by the current value.

For giving an operational semantic to a set of generators, we will first associate with every generator Gen, except the constants, an enabling condition denoted PreGen. These conditions are represented by the predicates PreSend and PreReceive in the specification of the type *SimpleProtocol*. No condition has been associated with the generator InitialProtocol; we will consider that this last generator is applied only once and that the value of the protocol is significant only after its application.

Now we can state the operational interpretation we give to the specification of the type *SimpleProtocol:* axioms of the type *SimpleProtocol* compute the value F(InitialProtocol), the function F being defined as the following nondeterministic recursive function:

```
F(p) =
    if PreSend(p) then F(Send(p))
    if PreReceive(p) then F(Receive(p))
    else p;
```

This function is nondeterministic; if it terminates, its final value may be one of several possible final values.

The operational properties we will state for the type *SimpleProtocol* must actually be understood as properties of the above function applied to the value InitialProtocol.

The main purpose of the introduction of conditions is to permit the definition of operational properties for the protocol. From the rewriting standpoint, conditions have no meaning and may be considered as simple abbreviations for more complicated expressions. We will say more below on what a condition should contain; roughly, conditions are stated such that, when TRUE, the associated generator actually does something "constructive."

4.5 Progress Properties and How To Prove Them

We will assume that it is suitable for the computation defined by our specifications to terminate; we use terminating models for protocols rather than cyclic models. Operational correctness for terminating systems concerns mainly the termination and freedom from undesired deadlock properties discussed in Section 2. The conjunction of these two properties assures that the transfer of messages terminates (according to the operational interpretation we defined), and that it terminates in an acceptable terminating state (this last criterion is specific to the protocol).

Freedom from undesired deadlocks

Though the state of the protocol may be correct wherever the computation stops (according to some functional correctness assertions), it should stop only in some states: the states we recognize as functionally valid terminating states. We call this property DeadlockFreeness (shorthand for "freedom from undesired deadlocks").

Let us denote *Progress*(p) as the predicate formed by the disjunction of all the enabling conditions associated with the generators. *Progress*(p) is true for the value p if at least one among the enabling conditions of the generators is true for that value; this means that a further computation step is possible from that value in the function we associated with the specification. A deadlock is a value p such that *Progress*(p) is false.

Let us define *Final*(p) as a predicate true iff the value p of the protocol may be considered a valid terminating value (this value does not have to be a deadlock).

Freedom from undesired deadlock may then be expressed as the following theorem (which does not imply that a deadlock is actually reachable).

> **theorem** DeadlockFreeness, not(Progress(p)) imp Final(p);

For the *SimpleProtocol* example we have:

> Progress(p) = = PreSend(p) or PreReceive(p);

and

> Final(p) = = Received(p) = InitialSequenceOfMessage;

An acceptable termination value here is a value such that all messages initially to be sent have been received.

As discussed in Section 2, the DeadlockFreeness theorem is a static property and hence may be proven by induction using the schema specific to the protocol. The transcript of its proof is given in Appendix 3.

Termination

Termination is a dynamic property; it states that a deadlock value is necessarily reached during the computation of the function we associated with the specification. Rather than trying to state the termination property more formally, we will focus on methods for proving it. The basic idea for these proofs comes from Floyd [Floy 67], with the method known as the *well-founded set* method.

The first step is to exhibit a mapping from the set of values of the type into a well-founded set; such a mapping is often called a *measure function.* Well-founded sets are ordered sets that contain no infinite length decreasing sequence of elements. For example, the set of natural numbers with regular ordering or the set of tuples of naturals with lexicographic ordering are well-founded sets; the set of all integers with regular ordering is not well-founded.

With this mapping, proving termination involves proving that the application of any generator, when enabled by its condition, will decrease the value of the measure function.

Termination proofs using this method may take very different forms, because the measure function that allows the proof depends on some specific properties of the represented system. For proving that the *SimpleProtocol* specification satisfies the termination property, we used a measure function defined as follows:

> Measure(p) = = 2*Length(ToSend(p)) + Length(Transit(p));

The measure function is a weighted sum of the number of messages not yet sent and the number of messages in transit.

The termination theorem may be stated as follows for the type *SimpleProtocol*:

```
theorem Termination, Measure(p) ge 0
    and PreSend(p) imp (Measure(Send(p)) lt Measure(p))
    and PreReceive(p) imp (Measure(Receive(p)) lt Measure(p));
```

The first clause and the fact that the range of the measure function is of type *Integer* imply that the range of the measure function is a well-founded set; the two following clauses assure that the value of the measure function will decrease every time an enabled generator is applied.

The first clause may be proved using the properties of type *SequenceOfMessage*; the following clauses by themselves express an induction on the set of generators of the type and may be proved using the properties of the types *Integer* and *SequenceOfMessage*. The transcripts of the proofs may be found in Appendix 3.

Note on the choice of the enabling conditions:

From the statement of the termination theorem above, it appears that the choice of the conditions associated with the generators is of some importance: a bad choice of conditions could affect the termination property of the protocol by forbidding the existence of an adequate measure function.

The semantics of the conditions is "progress"; the computation should progress whenever an enabled generator is applied. If no actual progress is made from the application of a generator, then we may consider that its condition, or the generator itself, has an erroneous definition.

Finally, the choice of the conditions for the specification is critical and determines, for the most part, the provability of the termination property.

5. A COMPLETE EXAMPLE: A SELECTIVE REPEAT PROTOCOL

5.1 Presentation

In this section, the methods devised in Section 4 are applied to the specification and verification of a significantly complex data transfer protocol. Properties proved include correctness of the transfer of messages, freedom from undesired deadlocks, and termination. On a scale of complexity, this protocol may be rated between the alternating bit protocol modeled in [Boch 77] and the data transfer protocol described in [Sten 76].

The protocol ensures the transmission of a sequence of messages from a fixed sender to a fixed receiver, over an unreliable transmission medium. This medium may lose packets or their acknowledgments, but cannot reorder, duplicate, or corrupt the packets in transit (nor their acknowledgments). To recover from the loss of messages, the protocol uses a selective repeat retransmission technique.

Selective repeat retransmission techniques are discussed in the literature (see for instance [East 79]); their name comes from the fact that only the messages lost in transit are retransmitted. Evidently, these techniques are based on some assumptions about the behavior of the transmission medium: We define these assumptions below.

No connection-establishment procedures are considered; the specification is concerned only with the data transmission aspects of the protocol. No flow control is considered either; storage space in the sender, transmission medium, and receiver are supposed unbounded.

The protocol allows several messages to be in transit at the same time. When sending a message over the transmission medium, the sender attaches to it a sequence number to form a packet and keeps a copy of the message. The receiver can rebuild the order in which the messages have been sent using the sequence numbers attached to the messages it accepts from the transmission medium. The receiver holds packets arriving out of order but releases the messages in order to the user; a counter that holds the sequence number of the last message released is used for that purpose. Both sender's and receiver's sequence numbers are initially set to zero and are supposed to increase monotonically.

The delivery of every message sent is confirmed by an acknowledgment from the receiver. The acknowledgments, as well as the data packets, may be lost in transit. The data or acknowledgment losses are recovered using a time-out mechanism: Each pending packet kept by the sender has attached to it the time it was sent, and will be resent if the confirmation of its reception has not arrived within a given constant interval.

The transmission medium is assumed to operate as follows in selective repeat protocols: A data packet or its acknowledgment has been lost in transit if the confirmation of its delivery does not arrive in time. This actual loss hypothesis may appear very strong; it may be implemented by a careful selection of the retransmission delays for the data packets (they must be set greater than the maximum transit time). Selective repeat protocols are primarily intended as link level protocols where the actual loss hypothesis is quite reasonable. It is clear that they would produce an extremely slow behavior if used at the transport level in a packet switching network, for instance.

The following presents an informal description of the primitives of the protocol, the axiomatization of the protocol, and a discussion of the correctness theorems and their proofs. The definitions of the types involved may be found in Appendix 2. For brevity, the proof transcripts of the proved theorems are not given here; they may be found in [Bert 80-1].

5.2 Informal Description of the Primitives of the Protocol

The elementary procedures of the protocol may be sketched as follows, where they are described as pairs:

<condition> --> <action>

An action is said to be "enabled" when its corresponding condition is true.

InitialProtocol:

Describes the initial state of the protocol.

No enabling condition is associated. Initializes the various data structures, including the sequence of messages to be sent, the sender's sequence number, and the receiver's sequence number.

Send:

Sends data packets to the transmission medium for delivery to the receiver.

Enabled if there is still at least one message to be sent. Forms a packet with the first of these messages and the current sender's sequence number; gives a copy of this packet to the medium, for delivery to the receiver, and another copy to the sender; starts a timer attached to this sequence number and increments the sender's current sequence number.

We assume that the medium keeps track of the order in which it received the packets from the sender and releases them to the receiver in the same order, except for the lost packets (i.e., it has a first-in/first-out discipline for the unlost packets).

Receive:

Receives data packets from the sender through the medium.

Enabled if at least one packet is to be delivered by the medium to the receiver. Accepts the first of these packets, gives the sequence number of this packet back to the medium for delivery to the sender (i.e., each packet received is acknowledged). If this packet has not been received previously, a copy is kept by the receiver for later delivery to the user.

Release:

Releases data packets held by the receiver to the user.

Enabled if the receiver holds the next expected (by the user) data packet; delivers the message of this packet to the user and increments the receiver's sequence number.

Update:

Keeps track of the acknowledged packets held by the sender.

Enabled if the oldest packet held by the sender has the same sequence number as the oldest acknowledgment; deletes this packet from the sequence of pending packets held by the sender and deletes the acknowledgment.

Resend:

Resends the data packets whose acknowledgments have not been received within a given time.

Enabled if the time-out of the oldest pending packet has expired; gives a copy of it to the medium for delivery to the receiver and reenables the time-out associated with this packet.

LosePkt:

Enabled if there is some packet in transit from the sender to the receiver; removes the oldest packet.

The LosePkt and the following LoseAck operation simulate the loss of messages and acknowledgments by the transmission medium.

LoseAck:

Enabled if there are some acknowledgments in transit; removes one of these from the sequence of acknowledgments held by the medium.

In the next section, the above informal descriptions of the primitives of the protocol are translated into an axiomatic specification.

5.3 Axiomatization of the Selective Repeat Protocol

Generators, selectors, and conditions:

The protocol will be defined as the *SelectiveRepeatProtocol* data type. It has a "record" structure, with a number of selector functions and a set of generators corresponding to the primitives described above.

In the axiomatic specification, the generators are given the same names as the corresponding procedures in Section 5.2; the conditions are given the names of the corresponding generators prefixed by "Pre" (but note that *InitialProtocol* has no condition associated); the selectors are given the following names:

ToSend:

Denotes the sequence of messages to be sent.

Pending:

The sequence of packets held by the sender. Packets belonging to the *Pending* sequence have been sent to the receiver, but their acknowledgments have not yet been received.

PktBuf:

Abstracts the storage of the data packets by the transmission medium.

Received:

Denotes the sequence of packets held by the receiver. Packets belonging to this sequence have been received by the receiver but not yet delivered to the user.

Released:

The sequence of messages delivered to the user by the receiver.

AckBuf:

Abstracts the storage of the acknowledgments by the medium.

SSN:

Denotes the sender's sequence number.

RSN:

Denotes the receiver's sequence number.

Figure 1 should assist in recalling these names. The figure has no pretension of formality and is intended to show, in a "data flow" fashion, the different data structures and operations involved.

Handling of time-outs

The model we use for representing the protocol does not handle time, so we have to interpret time-outs differently. The hypothesis we made for the loss of packets (that a packet whose time-out expires has actually been lost in transit) leads to the following abstraction for the transmission medium and the time-out functions:

- A *Lost* sequence holds the sequence numbers of the data packets and the acknowledgments that have been lost.
- The first packet in the *Pending* sequence is the oldest sent.
- The *Resend* operation is enabled if the sequence number of the first packet of *Pending* is present in *Lost*. It moves this packet from the first to the last position of the *Pending* sequence (reenables its time-out) and deletes the copy of its sequence number from *Lost*.
- The *LosePkt* (*LoseAck*) operation puts the sequence number of the first packet of *PktBuf* (the first element of *AckBuf*) in the *Lost* sequence.

Further hypotheses on the behavior of the medium

Although it is premature at this point, we are interested in proving termination of the data transfer. With the chosen characteristics of the transmission medium (it may lose an infinite number of data packets or acknowledgments), it is clear that the data transmission may never terminate.

To guarantee termination, we make the assumption that the number of items (data packets or acknowledgments) that the medium may lose is bounded by some constant. This seems reasonable since we don't give any fixed value to this maximum number of losses, though we suppose it bounded. This assumption implies that correct behavior of the transmission medium ultimately resumes after periods in which it loses items in transit. This behavior is represented in the axiomatic definition as follows:

- A *ToLose* counter, initially given an arbitrary but finite and positive integer value, is decremented by one every time an item is lost in transit.
- The enabling condition for the operations *LosePkt* and *LoseAck* are further augmented by the requirement that *ToLose* be positive, and the execution of any of these actions decrements the *ToLose* counter by one.

Building the type* SelectiveRepeatProtocol, *hierarchy of types

Type SelectiveRepeatProtocol

The type *SelectiveRepeatProtocol* is an axiomatization of the protocol described in Section 5.2, with the hypotheses and artifacts given in Section 5.3.

To build the complex *SelectiveRepeatProtocol* type, theory building operations discussed in Section 3.3 have been used. The axiomatization of the protocol is the top level type of a hierarchy composed of types *Boolean*, *Integer*, *Message*, *Packet*, *SequenceOfInteger*, *SequenceOfMessage*, and *SequenceOfPacket*. The construction

```
        [SSN]
          |
+-----\   |              \---------\
|     /->Send----------->/         /------>--Receive--+
| (1) \   |          +-->\   (2)   \->+         |     |
+-----/   |          |   /---------/  |         |     |
          | +-<--+   |                |         |     |
          | |    |   |               LosePkt    |     |
          V V   Resend                |   |     V     V
        \/\/\/\  |   |  +---------\   |   |     |  \/\/\/\
        |     |  |   |  |        /<--+    |     |  |     |
        |     |  |   |  |   (7)  \    [ToLose]  |  |     |
        | (3) |  |   |  +---------/<--+   |     |  | (4) |
        |     |  |   |        |       |   |     |  |     |
        \/\/\/\  |   +--<-----+       |   |     |  \/\/\/\
          | |    |                   LoseAck    |     |
          | |    |                    |         |     V
          | +->--+   +----------------+         |  Release
          |          |                          |   | |
          |          |  \----------\            |   | |    \-----+
          |          +<-/          /            |   | |    /     |
          | +----<------\     (6)  \<-----------+   | +--->\ (5) |
          | |           /----------/                |      /-----+
          V V                                       |
        Update                                    [RSN]


|<-------------------->|<---------------------->|<--------------->|
        SENDER            TRANSMISSION MEDIUM        RECEIVER
```

```
(1):  ToSend
(2):  PktBuf
(3):  Pending
(4):  Received
(5):  Released
(6):  AckBuf
(7):  Lost
```

Figure 1. Diagram of the *Selective Repeat Protocol*

of these types is briefly described below; their formal definitions and the definition of type *SelectiveRepeatProtocol* may be found in Appendix 2.

We will make use of some *Affirm* library predefined types. This library includes types *Boolean*, *Integer*, and a set of instantiable type structures such as *Queue*, *Set*, and *Sequence* of *ElemType* (*ElemType* is the parameter, the default constituent of these data structures).

Types Integer *and* SequenceOfInteger

The sender's and receiver's sequence numbers and the ToLose counter are of type *Integer*. The type we used is the *Affirm* built-in version: it is an axiomatization of the main facts about integers, including ordering.

The sequences Lost and AckBuf are sequences of integers. The type *SequenceOfInteger* has been built from an instantiation of the *Affirm* library *SequenceOfElemType*, enriched by some operators aimed at facilitating the expression of properties.

Types Message *and* SequenceOfMessage

Message is a minimally defined type: it has neither generators nor selectors, and its sole axiom states that identical messages are equal.

The sequences InitialSequenceOfMessage (the sequence of messages initially to be sent) and the fields ToSend and Released are sequences of messages. Type *SequenceOfMessage* is a straightforward instantiation of the type *SequenceOfElemType* from the *Affirm* library.

Types Packet *and* SequenceOfPacket

A *Packet* is defined as a record of an integer and a message, with two selector functions (Seq and Text) and a constructor (Pack).

The sequences Pending, PktBuf, and Received are sequences of packets. Type *SequenceOfPacket* is built from an instantiation of the *SequenceOfElemType*, enriched with some additional operators.

5.4 Proof of Correctness

Correctness invariants (nonoperational properties)

Correctness assertions

The purpose of the protocol is to transmit a sequence of messages from one location to another, regardless of the chosen strategy. Translated into our specification language, the correctness assertions should express that the sequence of messages released to the user is always some initial subsequence of the sequence of messages initially to be sent. This requirement has been proved; it is implied by the conjunction of theorems Output and RSNValue presented below.

Rather than simply proving this requirement, we derived and proved a set of theorems that it seems worthwhile to present. These theorems form a set of invariants that describes with enough precision how the protocol behaves in such aspects as Input/Output relationships (the correctness requirement), the relationships and interpretations of the values of the sender's and the receiver's sequence numbers, and the behavior of the transmission medium.

In addition to characterizing the behavior of some mechanisms of the protocol, we may consider that the invariants stated below are sufficient to convince the designer that the protocol is functionally correct. They confirm the intuitive understanding of the protocol.

In the interpretation of the invariants we will often refer to the "sequence of messages initially to be sent" (or Initial Sequence). This sequence is a parameter of the protocol, represented as the constant InitialSequenceOfMessage in the axiomatic specification. It is the initial value of the sequence ToSend and, if the protocol behaves properly, the final value of the sequence Released.

For each of the theorems in the list that follows we give its formal expression, a short informal comment, and a note about its proof. Theorems are separated into several groups, concerning the transmission medium characteristics, the sequence numbers, and the input/output sequence relationships respectively.

Transmission medium characteristics

These invariants state properties of the packets in transit and of the transmission medium behavior. They make explicit how the sequence numbers in the different storage components of the system are distributed, what their values are, their ordering, and the number of copies of each.

theorem RangeTransit1,
u in Seq(Pending(p) join PktBuf(p) join Received(p)) join Lost(p) join AckBuf(p)
imp u < SSN(p);

The sequence number of any packet in transit is smaller than the current sender's sequence number.

RangeTransit1 has been proved by induction on the set of generators.

theorem RangeTransit2,
(u < SSN(p)) and (u >= RSN(p))
imp u in Seq(Pending(p) join Received(p));

Any packet(s) whose sequence number is smaller than SSN and not smaller than RSN is necessarily pending or received and not yet released; this invariant implies that no packets ever disappear completely from the system.

Proven by induction and using the following RangeTransit3 theorem as a lemma.

theorem RangeTransit3, u in AckBuf(p) and (u >= RSN(p)) imp u in Seq(Received(p));

If an acknowledgment not smaller than RSN is present in the transmission medium, a packet with this sequence number is received and not yet released.

Proven by induction.

theorem nodupsTransit, nodups((Seq(PktBuf(p)) join Lost(p)) join AckBuf(p));

There are no duplicates in the sequence number field of the sequence of packets held by the medium (consequently there are no duplicated packets in the medium).

Proven by induction and using RangeTransit1.

theorem nodupsSeqReceived, nodups(Seq(Received(p)));

There are no duplicates in the sequence number field of the sequence of packets received and not yet released.

Proven by induction.

theorem nodupsSeqPending, nodups(Seq(Pending(p)));

There are no duplicates in the sequence number field of the sequence of pending packets.

Proven by induction and using RangeTransit1.

theorem Medium, Seq(Pending(p)) Except Lost(p) = AckBuf(p) join Seq(PktBuf(p));

This invariant states that the messages in transit and not lost are kept in the medium in the same order as their copies are held in the pending sequence. If we recall the association between the location of a packet in the pending sequence and the order in which it was sent, this invariant implies that there is no reordering of packets in the medium.

The Medium theorem has been proven by induction, and using RangeTransit1, nodupsSeqPending, and nodupsTransit as lemmas.

SSN and RSN range theorems

These invariants make explicit the relationships between the sender's and the receiver's sequence numbers and interpret their values and ranges in terms of the numbers of messages to be sent and released.

theorem RSNValue, RSN(p) = Length(Released(p));

RSNValue is proven by induction.

theorem RSNMin, RSN(p) >= 0;

Consequence of RSNValue.

theorem SSNValue, SSN(p) = Length(InitialSequenceOfMessage) - Length(ToSend(p));

SSN is the number of messages to be sent minus the number of messages not yet sent.

Proven by induction.

theorem SSNMin, SSN(p) >= 0;

Proven by induction.

theorem SSNMax, SSN(p) <= Length(InitialSequenceOfMessage);

Consequence of SSNValue.

theorem SSNvsRSN, SSN(p) >= RSN(p);

Uses RangeTransit1 and Induction.

theorem RSNMax, RSN(p) <= Length(InitialSequenceOfMessage);

Consequence of SSNvsRSN and SSNMax.

Input / Output relationships

These invariants relate the contents of the input sequence of messages (messages not yet sent), the output sequence of messages (released messages) and the contents of the Initial Sequence and the values of the sender's and the receiver's sequence numbers. Theorem Output below and theorem RSNValue above imply the correctness requirement.

theorem Input, ToSend(p) = LessInitial(InitialSequenceOfMessage, SSN(p));

The sequence of messages not yet sent is a final subsequence (whose length is the length of Initial Sequence less the sender's sequence number) of the initial sequence of messages.

Proven by induction and using SSNMin.

theorem TextTransit,
```
        i in ((Pending(p) join PktBuf(p)) join Received(p))
        imp Text(i) = pth(InitialSequenceOfMessage, Seq(i) + 1);
```

The text of any packet in transit is the text of the message whose rank in the initial sequence is given by the sequence number of the packet (plus one since we choose 0 as the origin for sequence numbers and the smallest rank is one).

Proven by induction; uses Input and SSNMin.

theorem Output, Released(p) = Initial(InitialSequenceOfMessage, RSN(p));

The sequence released is an initial subsequence, whose length is the receiver's sequence number, of the initial sequence. In conjunction with theorem RSNValue, Output implies that the Released sequence is an initial subsequence of the Initial Sequence.

Proven by induction, using RangeTransit1, TextTransit, RSNValue, and SSNValue.

theorem InputOutput,
```
        SSN(p) = RSN(p) imp Released(p) join ToSend(p) = InitialSequenceOfMessage;
```

When sender's and receiver's sequence numbers are equal, any message in the initial sequence is either not yet sent or released (and, incidentally, all messages sent, if any, have been released).

Theorem InputOutput also implies that the ordering of the messages in the initial sequence is kept in the Released sequence.

InputOutput is a consequence of SSNMin, Input, and Output.

Operational properties

Operational interpretation for the SelectiveRepeatProtocol *specification*

As for the type *SimpleProtocol*, we must define an operational interpretation for the specification of the type *SelectiveRepeatProtocol*. Axioms of *SelectiveRepeatProtocol* compute the value of the following nondeterministic function applied to the argument InitialProtocol:

```
F(p) = =
        if PreSend(p) then F(Send(p))
        if PreReceive(p) then F(Receive(p))
        if PreRelease(p) then F(Release(p))
        if PreUpdate(p) then F(Update(p))
        if PreResend(p) then F(Resend(p))
        if PreLosePkt(p) then F(LosePkt(p))
        if PreLoseAck(p) then F(LoseAck(p))
        else p;
```

For proving operational correctness here, we closely follow the concepts and methods discussed in Section 4, where we considered the properties of termination and of freedom from undesired deadlocks to be adequate for protocols.

Freedom from undesired deadlocks

The desirable deadlocks are the values of the protocol for which the predicate Final is true, and the predicate Progress is the disjunction of all the conditions associated with the constructors. The Freedom from undesired deadlocks property can therefore be stated:

```
not(Progress(p)) imp Final(p);
```

The Final predicate should be true for all protocol values considered as acceptable terminating values. For the *SelectiveRepeatProtocol*, a valid terminating value is a value for which the whole initial sequence of messages has been released to the user. In the context of type *SelectiveRepeatProtocol*, this predicate may be written:

Released(p) = InitialSequenceOfMessage;

The formulation of the DeadlockFreeness theorem is then:

```
theorem DeadlockFreeness,
                    PreSend(p)
        or          PreReceive(p)
        or          PreRelease(p)
        or          PreUpdate(p)
        or          PreResend(p)
        or          PreLosePkt(p)
        or          PreLoseAck(p)
        or          (Released(p) = InitialSequenceOfMessage);
```

Note that this characterization says nothing about the terminating states, the content of the transmission medium, or the various temporary storage components. If some requirements on the final state of these components are desirable, then the Final predicate must be augmented with these.

The DeadlockFreeness theorem has been proved using the theorems InputOutput, SSNvsRSN, RangeTransit2, and Medium.

Termination

This property states that a value of the protocol from which no progress is possible is necessarily reached in the computation.

The proof schema we use here for proving termination of the *SelectiveRepeatProtocol* is similar to the schema we used in Section 4 for proving termination of the *SimpleProtocol*. Both these proofs make use of the well-founded set method. In the context of type *SelectiveRepeatProtocol*, the interpretation we gave to the measure function Measure (the function that maps the states of the protocol into the set of integers) is the following:

```
Measure(p) = = (5 times Length(ToSend(p)))
            + (4 times ToLose(p))
            + (3 times Length(PktBuf(p)))
            + Length(AckBuf(p))
            + Length(Pending(p))
            + Length(Received(p))
            + (4 times Length(Lost(p)));
```

To prove termination, it is sufficient to prove that the value of Measure is always nonnegative and that it decreases every time an enabled constructor is applied, as expressed by the following Termination theorem: (The formal definitions of the conditions involved in the theorem may be found in Appendix 2, with the definition of the protocol.)

```
theorem Termination, Measure(p) >= 0
        and PreSend(p) imp ( Measure(Send(p)) < Measure(p) )
        and PreReceive(p) imp ( Measure(Receive(p)) < Measure(p) )
        and PreRelease(p) imp ( Measure(Release(p)) < Measure(p) )
        and PreUpdate(p) imp ( Measure(Update(p)) < Measure(p) )
and PreResend(p) imp ( Measure(Resend(p)) < Measure(p) )
and PreLosePkt(p) imp ( Measure(LosePkt(p)) < Measure(p) )
and PreLoseAck(p) imp ( Measure(LoseAck(p)) < Measure(p) );
```

The termination theorem includes its own structural induction schema. In addition, the proof makes use of the following lemmas:

theorem ToLosePositive, ToLose(p) >= 0;

Proven by induction, and using ToLoseNumbPositive.

theorem ToLoseNumbPositive, ToLoseNumb >= 0;

Assumed. ToLoseNumb is the initial value of ToLose(p), that is, the maximum number of items the medium is allowed to lose. We can assume this number is nonnegative; this seems obvious but the termination proof depends on this assumption.

5.5 Comments

The *SelectiveRepeatProtocol* experiment tested both the convenience of the specification methods discussed in Sections 3 and 4 and the interest and feasibility of semiautomated theorem proving for verifying the properties of a significantly complex protocol.

On the* SelectiveRepeatProtocol *experiment

Overall we have been quite satisfied with the methods devised. The example treated in the present section shows that complex mechanisms can be satisfactorily represented and managed; furthermore, these methods allow powerful analysis techniques.

The example chosen for this experiment has already resulted in a large-size specification, but the assumptions we made (no bounds for the storage components or the sequence numbers, for instance) are not completely realistic. The success of this verification convinces us that much more interesting protocol verification may be attempted, either on real-world selective repeat protocols (see, for example, [East 79] for some specification problems) or on protocols with different retransmission strategies (e.g., the data transfer protocol in [Sten 76] or the fundamental functions of TCP [CeKa 74] [Post 80]).

The reasons for choosing a protocol with this retransmission strategy to test the techniques presented in Section 4 are purely technical: For a first nontrivial experiment, this protocol seemed a good compromise between physical complexity (the size of the specification) and conceptual complexity.

We made further experiments, focusing only on the verification of operational properties, with protocols using "systematic retransmission" strategies, such as the protocols mentioned above. In these examples, the transmission medium was allowed to lose, reorder, and duplicate packets and acknowledgments in transit. The difficulty with these retransmission strategies is to prove the termination of the data transfer (assuming some reasonable hypotheses similar to those we made for the *SelectiveRepeatProtocol*). These experiments are described in [Bert 80-2] where it is shown that termination proofs for "systematic retransmission" protocols can also be handled, although they are more complex than for the SelectiveRepeatProtocol.

As mentioned earlier, various other specification and proof experiments on communication protocols have been carried out at Information Sciences Institute using the *Affirm* system with comparable specification methods. These experiments include the *alternating bit protocol* (specified, proven correct, and implemented in [TSEGS 81]), and a *three-way handshake* connection-establishment protocol [Schw 81-1] [Schw 81-2]. Specifications of higher level protocols are currently being investigated.

On the use of mechanical theorem proving

We must distinguish between several mechanical theorem proving techniques: those that build demonstrations and those that help the user carry them out. *Affirm*'s theorem prover is of the latter kind: the laborious part of the proofs is carried out by the system, but the user has to build the demonstrations and define the set of lemmas that will lead to successful proofs.

Devising a hierarchy of lemmas for proving a "high level" theorem may require a large amount of work. One must make the best use of both intuitive understanding of the behavior of the protocol and experience with the verification tool for deriving lemmas that are both right and useful for the proof. The main improvement over manual proofs seems to be that mechanical proofs are more reliable (assuming that the theorem prover is correct): errors in a manual proof are more likely to occur in the rewriting of expressions rather than in reasoning; rewriting is precisely what is carried out by the system.

An important issue with these methods is the cost, in user time and machine time. Machine time was not precisely measured, but, considering that several attempts were necessary to find a set of theorems giving a satisfying proof, the machine time ran to several (3 to 5) PDP-10 hours (the *Affirm* system runs on Interlisp-10). For the user, who was familiar with *Affirm* and with the properties of protocols, it took about three weeks to complete the specification of the *SelectiveRepeatProtocol* and proofs of the theorems as they appear in [Bert 80-1].

6. CONCLUSIONS

Contributions of this report are twofold: it presents techniques for specifying and verifying communication protocols that should offer some improvements over the methods currently used, and it constitutes an experiment in using algebraic methods and natural-deduction theorem-proving techniques for verifying real-world software systems.

Compared to techniques based on transition models, the algebraic methods permit compact specification of much more complex mechanisms. Abstraction is one of their natural characteristics. Furthermore, they permit introduction of parameters in the specifications, such as the sequence of messages to be sent (nothing was said here about its actual content; only its structure as *SequenceOfMessage* concerned us), or the number of communicating entities (limited to two in the *SelectiveRepeatProtocol*, but systems involving an arbitrary number of entities may be specified with little more complexity). Compared to linguistic methods, algebraic methods have many more possibilities for abstraction, and the encapsulated and self-contained aspects of the data type definitions greatly facilitate the verification of their properties.

The *Affirm* data type manipulation system used constantly in these experiments has proved to be of great value. The Program Verification project at ISI succeeded in making a very powerful tool that is smooth enough to be usable even by people unfamiliar with automated theorem proving. The mechanical verification may appear expensive in absolute costs; however, we believe that the quality of the results and the resulting degree of confidence in the specifications may justify this cost.

Omitted from this report were some important aspects of the specification of systems: interpretation of concurrency and nondeterminism, implementation of these specifications, abstraction of a protocol to be used in higher level systems, and many others. These should be considered in a more exhaustive work about algebraic specifications.

APPENDIX 1
Affirm Main Commands Summary

A1.1 Type definition commands

type *TypeName*;

Opens the definition of type *TypeName*.

edit *TypeName*;

Reopens the definition of TypeName for adding new material.

declare VarName {,VarName}: *TypeName*;

Declaration of the symbols of variables and their typing.

interface OpName(ArgList) {, OpName(ArgList)}: *TypeName*;

Declaration of the functional symbols, their syntax and typing.

infix OpName {,OpName};

Adds functional symbols to the list of those with infix syntax.

axioms/rulelemmas rule {, rule};

Each rule is an equation lhs == rhs. Adds rule(s) to the current set of rules. *Affirm* checks the unique termination property of the augmented set of rules.

define rule {, rule};

Similar to the axioms command except that these rules are applied on request from the user.

schemas rule {, rule};

Introduces specific inference rules (induction schemas). Induction schemas must be declared as Boolean operators. In the schemas, the inductive hypothesis may be written either Prop (it is then automatically expanded) or IH (it is expanded on request).

theorem PropName, Proposition;

Permits giving a name to a proposition. This proposition may then be called by this identifier in the proof commands.

end;

Closes the last open type definition.

A1.2 Proof development commands

The list of commands given in the following is far from being exhaustive: only the most often used proof development commands are given.

apply/use PropName;

These commands add the proposition denoted by PropName as a hypothesis to the current proposition.

The expression corresponding to the proposition will have its variables renamed. Typically, a put command will follow an apply command.

cases;

Raises embedded if-then-elses.
For instance,
    Op(if Cond then Term1 else Term2)
would become
    if Cond then Op(Term1) else Op(Term2)

employ SchemaName(VarName);

Request for applying the induction schema SchemaName. The induction will be carried out on variable VarName.

invoke OpName {,OpName};

Request for expanding operators entered via the define command or the inductive hypothesis entered as IH in induction schemas.

normint;

Applies a built-in integer simplifier to the current proposition.

put/let Var = Exp {, Var = Exp};

Instantiates variables. Sustitutes expressions to symbols of variables in the current proposition.

replace { Exp {, Exp}};

If Exp appears in the current proposition as the left-hand side of an equality, substitutes the right-hand side to all occurrences of the left-hand side in the proposition. If no argument is given, the left-hand side of the first occurring equality in the proposition is taken as the argument.

split;

If the current proposition contains alternatives, splits it into two subgoals, corresponding to the first alternative.

suppose Proposition;

Splits the current proposition into two subgoals:
    Proposition imp Current,
and
    Proposition or Current.

A1.3 Other commands

Affirm recognizes more than a hundred commands and subcommands, including the above, and program definition and verification commands, information display commands, user profile commands, cession control command, etc. All these are fully documented in [TGELB 81].

APPENDIX 2
Construction of the Type *SelectiveRepeatProtocol*

A2.1 Contents

Appendix 2 contains the formal definition of the type *SelectiveRepeatProtocol* and its auxiliary types.

All defined types make use of the basic *Boolean* and *Integer* types, which are built-in types of the *Affirm* system:

- Type *Boolean* is a two-valued type with constants TRUE and FALSE. It contains an axiomatization of the well-known boolean operators not, and, or, etc. Boolean expressions are internally transformed into if-then-else expressions as explained in [GHMu 78].
- Type *Integer* is an axiomatization of the usual facts about integers including arithmetic functions and ordering. A separate *Affirm* proof development command (**normint**) makes it possible to perform further simplifications on integer terms than the definition of the type *Integer* permits (this package helps prove nontrivial facts about integers).

Affirm does not yet support the definition of parameterized types (this is a scheduled improvement); however, it will mimic these constructions with sufficient rigorousness. The type definitions that follow (except those of types *ElemType* and *SequenceOfElemType*, which are straightforward *Affirm* print-outs) make free use of these theory-building techniques and thus have to be seen as the mental construction of the types rather than the full *Affirm* representations.

Some operators of the auxiliary types may not appear to be explicitly used in the definition or proofs of theorems of the *SelectiveRepeatProtocol* type; all types are actually considered to be autonomous.

A2.2 Types *ElemType* and *SequenceOfElemType*

The following are print-outs of the *Affirm* library types *ElemType* and *SequenceOfElemType*:

ElemType

ElemType is a minimally defined type. It has no generators, and the sole axiom is the equality of identical elements.

```
type ElemType;
    declare dummy: ElemType;
    axiom dummy=dummy == TRUE;
end {ElemType};
```

SequenceOfElemType

The *Sequence* structure is rather general. The *Stack* and *Queue* structures may be seen as restricted *Sequences*.

```
type SequenceOfElemType;
    needs type ElemType;
    declare dummy, ss, s, s1, s2: SequenceOfElemType;
    declare k, ii, i, i1, i2, j: ElemType;
declare k: Integer;
interfaces NewSequenceOfElemType, s apr i, i apl s, seq(i), s1 join s2, LessFirst(s), LessLast(s):
SequenceOfElemType;
infix join, apl, apr;
interfaces isNew(s), FirstInduction(s), Induction(s), NormalForm(s), i in s: Boolean;
infix in;
interfaces First(s), Last(s): ElemType;
interface Length(s): Integer;
interface pth(s, k): ElemType;
interfaces nodups(s): Boolean;
interfaces dedup(s), Initial(s, k), LessInitial(s, k), deletepth(s, k) : SequenceOfElemType;
axioms
    dummy=dummy == TRUE,
    NewSequenceOfElemType = s apr i == FALSE,
    s apr i = NewSequenceOfElemType == FALSE,
    s apr i = s1 apr i1 == ((s=s1) and (i=i1)),

    i apl NewSequenceOfElemType == NewSequenceOfElemType apr i,
    i apl (s apr i1) == (i apl s) apr i1,

    seq(i) == NewSequenceOfElemType apr i,

    NewSequenceOfElemType join s == s,
    (s apr i) join s1 == s join (i apl s1),

    LessFirst(s apr i) == if s = NewSequenceOfElemType
        then NewSequenceOfElemType
        else LessFirst(s) apr i,

    LessLast(s apr i) == s,

    isNew(s) == (s = NewSequenceOfElemType),

    i in NewSequenceOfElemType == FALSE,
    i in (s apr i1) == (i in s or (i=i1)),

    First(s apr i) == if s = NewSequenceOfElemType
        then i
        else First(s),

    Last(s apr i) == i,

    Length(NewSequenceOfElemType) == 0,
    Length(s apr i) == Length(s) + 1,

    nodups(s apr i) == (nodups(s) and ~(i in s)),
    nodups(NewSequenceOfElemType) == TRUE;
rulelemmas
    NewSequenceOfElemType = i apl s == FALSE,
    i apl s = NewSequenceOfElemType == FALSE,

    s join (s1 apr i) == (s join s1) apr i,
    s join NewSequenceOfElemType == s,
    (i apl s1) join s2 == i apl (s1 join s2),
        (s join (i apl s1)) join s2 = = s join (i apl (s1 join s2)),
        s join (s1 join s2) = = (s join s1) join s2,
        LessFirst(i apl s) = = s,
        LessLast(i apl s) = = if s = NewSequenceOfElemType
            then NewSequenceOfElemType
            else i apl LessLast(s),
        i in (i1 apl s) = = (i in s or (i = i1)),
        First(i apl s) = = i,
        Last(i apl s) = = if s = NewSequenceOfElemType
            then i
            else Last(s);
    define
        Initial(s, k) = = if (s = NewSequenceOfElemType) or (k <= 0)
            then NewSequenceOfElemType
            else First(s) apl Initial(LessFirst(s), k-1),
        LessInitial(s, k) = = if (s = NewSequenceOfElemType) or (k <= 0)
            then s
            else LessInitial(LessFirst(s), k-1),
        deletepth(s, k) = = if k <= 0
            then s
            else if k = 1
                        then LessFirst(s)
                        else First(s) apl deletepth(LessFirst(s), k-1),
        pth(s, k) = = if k = 1
            then First(s)
            else pth(LessFirst(s), k-1);
    schemas
        FirstInduction(s) = = cases(
            Prop(NewSequenceOfElemType),
            all ss, ii (IH(ss) imp Prop(ii apl ss)) ),
        Induction(s) = = cases(
            Prop(NewSequenceOfElemType),
            all ss, ii (IH(ss) imp Prop(ss apr ii)) ),
        NormalForm(s) = = cases(
            Prop(NewSequenceOfElemType),
            all ss, ii (Prop(ss apr ii)) );
end {SequenceOfElemType} ;
```

A2.3 Type *SequenceOfInteger*

SequenceOfInteger is an instantiation of the *Affirm* library *SequenceOfElemType* enriched by two operators whose formal definitions are the following:

edit *SequenceOfInteger*,

declare k,k':*Integer*,

declare s,s':*SequenceOfInteger*,

```
    interfaces s Delete k, s Except s':SequenceOfInteger;
    infix Delete, Except;
    axioms
        NewSequenceOfInteger Except s= =NewSequenceOfInteger,
        s Except NewSequenceOfInteger= =s,
        NewSequenceOfInteger Delete k= =NewSequenceOfInteger,
        (s apr k) Delete k'=if k=k'
                        then s
                        else (s Delete k') apr k;
    define
        (s apr k) Except s'= =if k in s'
                        then s Except s'
                        else (s Except s') apr k;
end;
```

Except removes from the sequence it has as its left argument the elements belonging to the sequence it has as its right argument; *Delete* deletes from the sequence it has as its left argument the last occurrence of the element it has as its right argument.

The complete *Affirm* definition of type *SequenceOfInteger* is the definition of type *SequenceOfElemType* in which occurrences of *ElemType* are replaced by *Integer*, augmented by the above part of the theory.

A2.4 Types *Message* and *SequenceOfMessage*

Message

Message is a minimally specified type; it has no generators and its only axiom states the equality of identical messages. It may be defined as a replication of *ElemType*.

SequenceOfMessage

SequenceOfMessage is a straightforward instantiation of the *Affirm* type library *SequenceOfElemType*; its definition is the same as *SequenceOfElemType* in which occurrences of *ElemType* are replaced by *Message*.

A2.5 Types *Packet* and *SequenceOfPacket*

Packet

A packet is defined as a record of an *Integer* and a *Message*, with an explicit construction operator Pack and two selection functions Seq and Text.

Selectors Seq and Text extract the *Integer* field and the *Message* field of a packet respectively; the constructor Pack builds a packet from an integer and a message.

Type *Packet* is defined as follows:

```
type Packet;
    needs type Message;
    declare dummy, p: Packet;
    declare i: Integer;
    declare m: Message;
    interface Pack(i, m): Packet;
    interface Seq(p): Integer;
    interface Text(p): Message;
    axioms
        dummy=dummy == TRUE,
        Seq(Pack(i, m)) == i,
        Text(Pack(i, m)) == m;
end;
```

SequenceOfPacket

Type *SequenceOfPacket* is built from an instantiation of the *Affirm* library *SequenceOfElemType* enriched by the operators Text, Seq, Except, and Such. Text and Seq extract the integer field and the message field of a sequence of packets respectively; Except is defined as in *SequenceOfInteger*, and Such extracts from a sequence of packets the last packet with a given sequence number. These operators are formally defined as follows:

```
edit SequenceOfPacket;
    declare k: Integer;
    declare ii, i, i1, i2, j: Packet;
    declare ss, s, s1, s2: SequenceOfPacket;
    interfaces s Except s', s Delete i:SequenceOfPacket;
    infix  Except, Delete;
    interfaces Such(s,k):Packet;
    interface Seq(s):SequenceOfInteger;
    interface Text(s):SequenceOfMessage;
    axioms
        Such(s apr i,k)==if k=Seq(i) then i else Such(s,k),
        NewSequenceOfPacket Delete i==NewSequenceOfPacket,
        (s apr i)Delete j==if i=j then s else (s Delete j) apr i;
        { Such(s,k) extracts from the sequence s a packet with a given sequence number k. The sequence s is
        supposed to hold such a packet.}
        { Delete is defined as in type SequenceOfInteger.}
    axioms
        Seq(NewSequenceOfPacket)==NewSequenceOfInteger,
        Seq(s apr i)==Seq(s) apr Seq(i),
        Seq(i apl s)==Seq(i) apl Seq(s),
        Seq(s1 join s2)==Seq(s1) join Seq(s2),
        Seq(LessFirst(s))==LessFirst(Seq(s)),
        Seq(First(s))==First(Seq(s)),
        Seq(LessLast(s))==LessLast(Seq(s)),
        Seq(Last(s))==Last(Seq(s));
```

{ Seq(s) extracts the sequence number field of a *SequenceOfPacket*; the sequence number field of a *SequenceOfPacket* is the sequence of integers constituted by the sequence number fields of the individual packets in the *SequenceOfPacket*.}

axioms

```
    Text(NewSequenceOfPacket) = = NewSequenceOfMessage,
    Text(s apr i) = = Text(s) apr Text(i),
    Text(i apl s) = = Text(i) apl Text(s),
    Text(s1 join s2) = = Text(s1) join Text(s2),
    Text(LessFirst(s)) = = LessFirst(Text(s)),
    Text(First(s)) = = First(Text(s)),
    Text(LessLast(s)) = = LessLast(Text(s)),
    Text(Last(s)) = = Last(Text(s));
```

{ Text(s) extracts the message field of a *SequenceOfPacket*; the message field of a *SequenceOfPacket* is the sequence of messages constituted by the text of the individual packets in the *SequenceOfPacket*.}

axiom

```
    NewSequenceOfPacket Except s = = NewSequenceOfPacket;
```

define

```
    (s apr i) Except s' = = if i in s'
        then s Except s'
        else (s Except s') apr i;
```

{ Except is defined as in *SequenceOfInteger*.}

end;

It may be noticed that we gave the same names to distinct functions belonging to the definitions of different data types. This does not lead to any ambiguity; the types of the arguments of the functions suffice to distinguish between them.

A2.6 Type *SelectiveRepeatProtocol*

The definition of the type *SelectiveRepeatProtocol* is given in the sequel. The set of axioms in the definition contains an important number of "no change" axioms, axioms that just say that some selectors are not modified by the application of some generators. In order to make the specification more readable, we use in the following definition a "pseudo generator" denoted *Others*(p). Any axiom containing this "generator" must be interpreted as denoting the set of axioms obtained by substituting to *Others* all generators that never appear elsewhere in the specification in this context.

Type *SelectiveRepeatProtocol* is defined as follows:

type *SelectiveRepeatProtocol*;

needs types *SequenceOfInteger, Message, SequenceOfMessage, Packet, SequenceOfPacket*;

declare p, pp: *SelectiveRepeatProtocol*;

declare s1, s2: *SequenceOfMessage*;

declare s, s': *SequenceOfPacket*;

interfaces

InitialProtocol, Send(p), Receive(p), Release(p), Update(p), Resend(p), LosePkt(p), LoseAck(p): *SelectiveRepeatProtocol*;

interfaces

```
    InitialSequenceOfMessage, ToSend(p), Released(p):SequenceOfMessage;
interfaces
    Pending(p), PktBuf(p), Received(p):SequenceOfPacket;
interfaces
    AckBuf(p), Lost(p):SequenceOfInteger;
interfaces
    ToLoseNumb, SSN(p), RSN(p), ToLose(p):Integer;
interfaces
    PreSend(p), PreReceive(p), PreRelease(p), PreUpdate(p), PreResend(p), PreLosePkt(p),
    PreLoseAck(p):Boolean;
interface
    SelectiveRepeatProtocolInduction(p):Boolean;
axioms
    PreSend(p) = = not(ToSend(p) = NewSequenceOfMessage),
    PreReceive(p) = = not(PktBuf(p) = NewSequenceOfPacket),
    PreRelease(p) = = not(Received(p) = NewSequenceOfPacket)
                   and (RSN(p) in Seq(Received(p))),
    PreUpdate(p) = = not(Pending(p) = NewSequenceOfPacket)
                   and not(AckBuf(p) = NewSequenceOfInteger)
                   and Seq(First(Pending(p))) = First(AckBuf(p)),
    PreResend(p) = = not(Pending(p) = NewSequenceOfPacket)
                   and not(Lost(p) = NewSequenceOfInteger)
                   and Seq(First(Pending(p))) in Lost(p),
    PreLosePkt(p) = = (ToLose(p) >0)
                   and not(PktBuf(p) = NewSequenceOfPacket),
    PreLoseAck(p) = = (ToLose(p) >0)
                   and not(AckBuf(p) = NewSequenceOfInteger);
axioms
    ToSend(InitialProtocol) = = InitialSequenceOfMessage,
    ToSend(Send(p)) = = if PreSend(p)
                   then LessFirst(ToSend(p))
                   else ToSend(p),
    ToSend(Others(p)) = = ToSend(p);
axioms
    Released(InitialProtocol) = = NewSequenceOfMessage,
    Released(Receive(p)) = = Released(p),
    Released(Release(p)) = = if PreRelease(p)
                   then Released(p) apr Text(Such(Received(p),RSN(p)))
                   else Released(p),
    Released(Others(p)) = = Released(p);
axioms
    Pending(InitialProtocol) = = NewSequenceOfPacket,
    Pending(Send(p)) = = if PreSend(p)
                   then Pending(p) apr Pack(SSN(p),First(ToSend(p)))
                   else Pending(p),
    Pending(Update(p)) = = if PreUpdate(p)
```

```
                    then LessFirst(Pending(p))
                    else Pending(p),
    Pending(Resend(p))= =if PreResend(p)
                    then LessFirst(Pending(p)) apr First(Pending(p))
                    else Pending(p),
    Pending(Others(p))= =Pending(p);
axioms
    Received(InitialProtocol)= =NewSequenceOfPacket,
    Received(Receive(p))= =if PreReceive(p)
                    then if (Seq(First(PktBuf(p))) ge RSN(p))
                    and not (Seq(First(PktBuf(p))) in Seq(Received(p)))
                            then Received(p) apr First(PktBuf(p))
                            else Received(p)
                    else Received(p),
    Received(Release(p))= =if PreRelease(p)
                    then Received(p) Except Such(Received(p),RSN(p))
                    else Received(p),
    Received(Others(p))= =Received(p);
axioms
    Lost(InitialProtocol)= =NewSequenceOfInteger,
    Lost(Resend(p))= =if PreResend(p)
                    then Lost(p) Except Seq(First(Pending(p)))
                    else Lost(p),
    Lost(LosePkt(p))= =if PreLosePkt(p)
                    then Lost(p) apr Seq(First(PktBuf(p)))
                    else Lost(p),
    Lost(LoseAck(p))= =if PreLoseAck(p)
                    then Lost(p) apr First(AckBuf(p))
                    else Lost(p),
    Lost(Others(p)) = = Lost(p);
axioms
    PktBuf(InitialProtocol)= =NewSequenceOfPacket,
    PktBuf(Send(p))= =if PreSend(p)
                    then PktBuf(p) apr Pack(SSN(p),First(ToSend(p)))
                    else PktBuf(p),
    PktBuf(Receive(p))= =if PreReceive(p)
                    then LessFirst(PktBuf(p))
                    else PktBuf(p),
    PktBuf(Resend(p))= =if PreResend(p)
                    then PktBuf(p) apr First(Pending(p))
                    else PktBuf(p),
    PktBuf(LosePkt(p))= =if PreLosePkt(p)
                    then LessFirst(PktBuf(p))
                    else PktBuf(p),
    PktBuf(Others(p))= =PktBuf(p);
axioms
    AckBuf(InitialProtocol)= =NewSequenceOfInteger,
    AckBuf(Receive(p))= =if PreReceive(p)
                    then AckBuf(p) apr Seq(First(PktBuf(p)))
                    else AckBuf(p),
    AckBuf(Update(p))= =if PreUpdate(p)
```

```
                then LessFirst(AckBuf(p))
                else AckBuf(p),
    AckBuf(LoseAck(p))= =if PreLoseAck(p)
                then LessFirst(AckBuf(p))
                else AckBuf(p),
    AckBuf(Others(p))= =AckBuf(p);
axioms
    SSN(InitialProtocol)= =0,
    SSN(Send(p))= =if PreSend(p)
                then SSN(p)+1
                else SSN(p),
    SSN(Others(p))= =SSN(p);
axioms
    RSN(InitialProtocol)= =0,
    RSN(Receive(p))= =RSN(p),
    RSN(Release(p))= =if PreRelease(p)
                then RSN(p)+1
                else RSN(p),
    RSN(Others(p))= =RSN(p);
axioms
    ToLose(InitialProtocol)= =ToLoseNumb,
    ToLose(LosePkt(p))= =if PreLosePkt(p)
                then ToLose(p)-1
                else ToLose(p),
    ToLose(LoseAck(p))= =if PreLoseAck(p)
                then ToLose(p)-1
                else ToLose(p),
    ToLose(Others(p))= =ToLose(p);
schema
    SelectiveRepeatProtocolInduction(pp)= =cases( Prop(InitialProtocol),
                all p(IH(p) imp Prop(Send(p))),
                all p(IH(p) imp Prop(Receive(p))),
                all p(IH(p) imp Prop(Release(p))),
                all p(IH(p) imp Prop(Update(p))),
                all p(IH(p) imp Prop(Resend(p))),
                all p(IH(p) imp Prop(LosePkt(p))),
                all p(IH(p) imp Prop(LoseAck(p))) );
end;
```

APPENDIX 3
Proof Transcripts of the *SimpleProtocol* Theorems

A3.1 Additional definitions

The following are additional definitions to the *SimpleProtocol* specification presented in Section 4. These operators will help in expressing the theorems.

```
edit SimpleProtocol;
   interfaces Progress(p), Final(p), TermCondSend(p), TermCondReceive(p): Boolean;
   interface Measure(p): Integer;
   define
      Progress(p) == (ToSend(p) = NewSequenceOfMessage
         imp Transit(p) ~= NewSequenceOfMessage),
      Final(p) == (Received(p) = InitialSequenceOfMessage),
      Measure(p) == Length(Transit(p)) + Length(ToSend(p)) * 2,
      TermCondSend(p) == (PreSend(p)
         imp  Measure(Send(p)) < Measure(p)),
      TermCondReceive(p) == ( PreReceive(p)
         imp  Measure(Receive(p)) < Measure(p));
end;
```

A3.2 Proof transcripts

This section contains the proof transcripts, as output by *Affirm*, of the *SimpleProtocol* correctness theorems and the lemmas for the types *SequenceOfMessage* and *Integer*.

SimpleProtocol theorems

```
theorem TransferCorrect,
         Received (p) = Initial(InitialSequenceOfMessage, Length(Received(p)));

   TransferCorrect uses TransferLemma! and InitialLemma!.

   proof tree:
   66:! TransferCorrect
        apply TransferLemma
   67:  6  put p'=p
   500: 7  replace InitialSequenceOfMessage
   507: 227 apply InitialLemma
   508: 231 put (s = Received(p))
                                      and (s' = Transit(p) join ToSend(p))
   508: (proven!)
   --------------------

   assume TransferLemma,
          (Received(p) join Transit(p)) join ToSend(p)
                       = InitialSequenceOfMessage;

   assume InitialLemma, Initial(s join s', Length(s)) = s;

   --------------------
```

PRECEDING PAGE BLANK-NOT FILMED

theorem TransferLemma,
```
          (Received(p) join Transit(p)) join ToSend(p)
                         = InitialSequenceOfMessage;
```

TransferLemma uses AplSplit! and JoinSubst!.

```
proof tree:
74:! TransferLemma
       employ SimpleProtocolInduction(p)
     InitialProtocol:
       immediate
75:  Send:
       31 cases
76:      33 apply AplSplit
77:      35 put s = ToSend(p)
78:      36 replace
       (proven!)
80:  Receive: {TransferLemma
       32 cases
81:      37 apply AplSplit
82:      38 put s = Transit(p)
85:->     39 apply JoinSubst
86:      41 put s = First(Transit(p)) apl LessFirst(Transit(p))
                                          and (s' = Transit(p))
                                          and (s" = ToSend(p))
87:      42 replace
          (proven!)
--------------------
```

```
assume AplSplit, s ~= NewSequenceOfMessage
        imp First(s) apl LessFirst(s) = s;

assume JoinSubst, s=s' imp s join s" = s' join s";

--------------------
```

theorem DeadlockFreeness, ~Progress(p) imp Final(p);

DeadlockFreeness uses TransferLemma!.

```
proof tree:
178:! DeadlockFreeness
       invoke Progress
179: 85 invoke Final
180: 86 apply TransferLemma
181: 87 put p'=p
182: 88 replace
       (proven!)
--------------------
```

```
assume TransferLemma, (Received(p) join Transit(p)) join ToSend(p)
        = InitialSequenceOfMessage;

--------------------
```

theorem Termination, (Measure(p) >= 0)
```
          and TermCondSend(p) and TermCondReceive(p);
```

Termination uses LengthNonNeg!, ProdNonNeg%, and LengthLessFirst!.

```
proof tree:
192:! Termination
       split
193:   first:
       89 split
194:     first:
        91  invoke Measure
195:       93  apply LengthNonNeg
196:       94  put s = Transit(p)
222:       95  apply LengthNonNeg
223:       104 put s = ToSend(p)
224:       105 apply ProdNonNeg
225:       106 put (u = Length(ToSend(p))) and (v=2)
226:       107 normint
        (proven!)
233:    second: {Termination, first:
        92  invoke TermCondSend
236:       108 invoke Measure|all|
237:       109 apply LengthLessFirst
238:       111 put s = ToSend(p)
239:       112 replace
        (proven!)
241:  second: {Termination
      90  invoke TermCondReceive
242:    113 invoke Measure|all|
243:    114 apply LengthLessFirst
244:    115 put s = Transit(p)
245:    116 replace
      (proven!)
--------------------
```

```
assume LengthNonNeg, Length(s) >= 0;

assume ProdNonNeg, (u >= 0) and (v >= 0) imp u*v >= 0;

assume LengthLessFirst, s ~= NewSequenceOfMessage
        imp Length(LessFirst(s)) = Length(s) - 1;

--------------------
```

SequenceOfMessage lemmas

theorem **InitialLemma**, Initial(s join s', Length(s)) = s;

InitialLemma uses LengthApl! and LengthNonNeg!.

```
proof tree:
510:! InitialLemma
      employ FirstInduction(s)
511:  NewSequenceOfMessage:
      233 invoke Initial
511:    (proven!)
514:  apl:
      234 invoke Initial
515:    235 cases
518:    236 apply LengthApl
519:    237 put (i=ii') and (s=ss')
520:    238 replace
521:    239 apply LengthNonNeg
522:    240 put s=ss'
523:    241 invoke IH
528:    242 put s=s'
529:    243 replace
->     (proven!)
--------------------
```

```
assume LengthApl, Length(i apl s) = Length(s) + 1;

assume LengthNonNeg, Length(s) >= 0;

--------------------
```

theorem **AplSplit**. s ~= NewSequenceOfMessage
imp First(s) apl LessFirst(s) = s:

```
proof tree:
89:! AplSplit
      employ Induction(s)
     NewSequenceOfMessage:
      immediate
90:   apr:
      43 cases
91:     44 invoke IH
91:     (proven!)
```

theorem **JoinSubst**. s=s' imp s join s'' = s' join s'':

```
proof tree:
94:! JoinSubst
      employ Induction(s'')
     NewSequenceOfMessage:
      immediate
95:   apr:
      46 invoke IH
96:     47 replace
      (proven!)
```

theorem **LengthApl**. Length(i apl s) = Length(s) + 1:

```
proof tree:
449:! LengthApl
      employ Induction(s)
     NewSequenceOfMessage:
      immediate
450:  apr:
      209 invoke IH
451:    210 put i'=i
452:    211 replace
      (proven!)
```

theorem **LengthLessFirst**. s ~= NewSequenceOfMessage
imp Length(LessFirst(s)) = Length(s) - 1:

```
proof tree:
252:! LengthLessFirst
      employ Induction(s)
     NewSequenceOfMessage:
      immediate
253:  apr:
      119 cases
254:    120 invoke IH
255:    121 replace
      (proven!)
```

theorem **LengthNonNeg**. Length(s) >= 0:

```
proof tree:
247:! LengthNonNeg
      employ Induction(s)
     NewSequenceOfMessage:
      immediate
248:  apr:
      117 invoke IH
249:    118 normint
      (proven!)
```

Integer lemmas

theorem **ProdNonNeg**. (u >= 0) and (v >= 0) imp u*v >= 0;

```
proof tree:
135:!  ProdNonNeg  employ  Induction(v)
      0:
           Immediate
136:   DIFFERENCE:
       45  invoke  first  IH
137:     47  put  u'=u
138:     48  normint
138:     (proven!)
140:   PLUS:{ProdNonNeg
       46  invoke  first  IH
141:     50  put  u'=u
142:     51  normint
142:     (proven!)
```

REFERENCES

[AUYa 79] Aho, A.V., J.D. Ulmann, and M. Yannakakis, "Modeling communication protocols by automata," *20th Symposium on Foundations of Computer Sciences*, pp. 267-273, October 1979.

[AABe 78] Azema, P., J.M. Ayache, and B. Berthomieu, "Design and verification of communication procedures: A bottom-up approach," *Proceedings of the Third Conference on Software Engineering*, pp. 168-174, 1978.

[ABDe 80] Azema, P., B. Berthomieu, and P. Decitre, "The design and validation by means of Petri nets of a mechanism for the invocation of remote servers," *Proceedings of IFIP Congress 80*, pp. 599-604, Melbourne, Australia, North Holland, October 1980.

[Bert 80-1] Berthomieu, B., "Selective repeat protocol, axiomatization and proofs," USC/Information Sciences Institute, Program Verification Project, *Affirm* memo 36, September 1980.

[Bert 80-2] Berthomieu, B., "Proving progress properties of communication protocols in *Affirm*," USC/Information Sciences Institute, Program Verification Project, *Affirm* memo 35, September 1980.

[Boch 77] Bochmann, G.V., "A unified method for the specification and verification of protocols," *Proceedings of IFIP Congress 77*, pp. 229-234, North Holland, 1977.

[Boch 78] Bochmann, G.V., "Finite state description of communication protocols," *Computer Networks* 2, 1978, 361-372.

[BoSu 80] Bochmann, G.V., and C.A. Sunshine, "Formal methods in Communication Protocol Design," *IEEE Transactions on Communications* COM-28 (4), April 1980, 624-631.

[BuGo 77] Burstall, R.M., and J.A. Goguen, "Putting theories together to make specifications," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 1045-1058, MIT, Cambridge, Mass., 1977.

[CeKa 74] Cerf, V.G., and R.E. Kahn, "A protocol for packet network intercommunication," *IEEE Transactions on Communications* COM-22 (5), May 1974, 647-648.

[Dant 80] Danthine, A., "Protocol representation with finite-state models," *IEEE Transactions on Communications* COM-28 (4), April 1980, 632-643.

[East 79] Easton, M.C., *Design Choices for Selective-Repeat Retransmission Protocols*, IBM Research Report RC 7808, 1979.

[Floy 67] Floyd, R.W., "Assigning meaning to programs," in J.T. Schwartz (ed.), *Proceedings of Symposia in Applied Mathematics*, pp. 19-32, American Mathematical Society, 1967.

[Gerh 80] Gerhart, S.L., "An overview of *Affirm*: A specification and verification system," in *Proceedings of IFIP Congress 80*, pp. 343-347, Melbourne, Australia, North Holland, October 1980.

[Gogu 77] Goguen, J.A., "Abstract errors for abstract data types," in E. Neuhold (ed.), *Formal Description of Programming Concepts: Proceedings of the Working Conference*, pp. 491-526, St. Andrews N.B. Canada, 1977.

[GTWa 78] Goguen, J.A., J.W. Thatcher, and E.G. Wagner, "Abstract data types as initial algebras and the correctness of data representations," in R.T. Yeh (ed.), *Current Trends in Programming Methodology*, Volume 4, pp. 80-149, Prentice Hall, 1978.

[GuHo 78] Guttag, J.V., and J.J. Horning, "The algebraic specification of abstract data types," *Acta Informatica* 10, 1978, 27-52.

[GHMu 78] Guttag, J.V., E. Horowitz, and D.R. Musser, "Abstract data types and software validation," *Communications of the ACM* 21, December 1978, 1048-1064.

PRECEDING PAGE BLANK-NOT FILMED

[Hoar 69] Hoare, C.A.R., "An axiomatic basis for computer programming," *Communications of the ACM* 12 (10), October 1969, 576-580.

[Hoar 72] Hoare, C.A.R., "Proof of correctness of data representation," *Acta Informatica* 1 (4), 1972, 271-281.

[HuOp 80] Huet, G., and D.C. Oppen, "Equations and rewrite rules: A survey," in R. Book (ed.), *Formal Language Theory: Perspectives and Problems*, Academic Press, 1980. Also TR CSL-111, SRI International, January 1980.

[Kell 76] Keller, R.M. "Formal verification of parallel programs," *Communications of the ACM* 17 (7), July 1976, 371-384.

[KnBe 69] Knuth, D.E., and P.B. Bendix, "Simple word problems in universal algebras," in *Computational Problems in Abstract Algebras*, pp. 263-297, Pergamon Press, 1969.

[Lamp 77] Lamport, L., "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering* SE-3 (2), March 1977, 125-143.

[LaSc 74] Lautenbach, K., and H.A. Schmidt, "Use of Petri nets for proving correctness of concurrent systems," *Proceedings of IFIP Congress 74*, North Holland, 1974.

[LiZi 75] Liskov, B. and S.N. Zilles, "Specification techniques for data abstractions," *IEEE Transactions on Software Engineering* SE-1 (1), March 1975, 7-19.

[Merl 79] Merlin, P., "Specification and validation of protocols," *IEEE Transactions on Communication* COM-27 (11), November 1979, 1671-1680.

[Muss 80-1] Musser, D.R., "Abstract data type specification in the *Affirm* system," *IEEE Transactions on Software Engineering* SE-6 (1), January 1980, 24-32.

[Muss 80-2] Musser, D.R., "On proving inductive properties of abstract data types," *Seventh ACM Symposium on Principles of Programming Languages*, pp. 154-162, Las Vegas, Nevada, January 1980.

[Nutt 72] Nutt, G.J., *The Formulation and Application of Evaluation Nets*, Thesis, University of Washington, Computer Science Group, 1972.

[Parn 72] Parnas, D.L., "A technique for software module specification with examples," *Communications of the ACM* 15 (5), May 1972, 330-336.

[Post 80] Postel, J. (ed.), "DOD Standard Transmission Control Protocol", IEN 129, RFC 761, USC/Information Sciences Institute, NTIS ADA082609, January 1980. Appears in *ACM Computer Communication Review*, Special Interest Group on Data Communications 10 (4), October 1980, 52-132.

[PoFa 76] Postel, J.B., and D. Farber, "Graph modeling of communication protocols," *Fifth Texas Conference on Computing Systems*, pp. 66-77, Austin, Texas, 1976.

[RaEs 80] Razouk, R.R., and G. Estrin, "Validation of the X-21 interface specification using SARA," *Proceedings of the 1980 Trends and Applications Symposium: Computer Network Protocols*, pp. 155-167, National Bureau of Standards, Gaithersburg, Maryland, May 1980.

[Schw 81-1] Schwabe, D., *Formal Techniques for Specification and Verification of Protocols*, Ph.D. Thesis, University of California, Los Angeles, Computer Science Department, March 1981.

[Schw 81-2] Schwabe, D., *Formal Specification of a Connection-Establishment Protocol*, USC/Information Sciences Institute, RR-81-91, March 1981.

[Sten 76] Stenning, N.V., "A data transfer protocol," *Computer Networks* 1, 1976, 99-100.

[Suns 81] Sunshine, C.A., *Formal Modeling of Communication Protocols*, USC/Information Sciences Institute RR-81-89, March 1981.

[TGELB 81] Thompson, D.H., et al. (eds.), *The Affirm Reference Library*, USC/Information Sciences Institute, 1981. (Five volumes: Reference Manual, User's Guide, Types Library, Annotated Transcripts, and Collected Papers, 450 pages.)

[TSEGS 81] Thompson, D.H., et al., *Specification and Verification of Communication Protocols in Affirm Using State Transition Models.* USC/Information Sciences Institute RR-81-88, March 1981.

[Vale 77] Valette R., *Sur la description, l'analyse et la validation des systemes de commande paralleles,* These d'etat, Toulouse, 1977.