A GRAMMAR-BASED METHODOLOGY FOR PROTOCOL SPECIFICATION AND IMPLEMENTATION

by

David P. Anderson

Computer Sciences Technical Report #612

September 1985

A GRAMMAR-BASED METHODOLOGY FOR PROTOCOL SPECIFICATION AND IMPLEMENTATION

by

David P. Anderson

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1985

ABSTRACT

A new methodology for specifying and implementing communication protocols is presented. This methodology is based on a formalism called "Real-Time Asynchronous Grammars" (RTAG), which uses a syntax similar to that of attribute grammars to specify allowable message sequences. In addition RTAG provides mechanisms for specifying data-dependent protocol activities, real-time constraints, and concurrent activities within a protocol entity. RTAG encourages a top-down approach to protocol design that can be of significant benefit in expressing and reasoning about highly complex protocols. As an example, an RTAG specification is given for part of the Class 4 NBS Transport Protocol (TP-4).

Because RTAG allows protocols to be specified at a highly detailed level, major parts of an implementation can be automatically generated from a specification. An *RTAG parser* can be written which, when combined with an RTAG specification of a protocol and a set of interface and utility routines, constitutes an implementation of the protocol. To demonstrate the viability of RTAG for implementation generation, an RTAG parser has been integrated into the kernel of the 4.2 BSD UNIX operating system, and has been used in conjunction with the RTAG TP-4 specification to obtain a working TP-4 implementation in the DOD Internet environment.

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1. Background

The usage of *protocol* in Computer Science stems from its usual meaning as a mutually agreed-upon set of rules for interaction. The difference is that the interacting parties are computer programs rather than people. *Data communication* protocols are those that manage the transfer of data between entities on distinct computer systems. Such protocols serve to enhance the flexibility and reliability of lower-level communication facilities.

General-purpose data communications systems may involve protocols operating at several different logical and hardware levels ([Cyp78], [Tan81]). As an organizational model for such systems, the International Standards Organization (ISO) developed the Open System Interconnection (OSI) Seven-Layer Reference Model ([Zim80], [Day83], [ISO83]). In this model, the protocols at a particular layer are defined in terms of the services they provide to the immediately upper layer, and the services they expect of the immediately lower layer.

A list of the services typically provided by each layer of the OSI model is given in [Bow83]. For example, a typical protocol in the transport layer might provide reliable connection management and ordered, reliable dual-priority data

transfer between processes on different hosts, in the presence of packet loss, delay, reordering and garbling by the intervening network and/or the lower protocol levels, and in the presence of host machine failures (see [Stu83]).

The services provided by data communications protocols can be realized using a variety of mechanisms ([Sun78], [Fle78], [Pou78]). Connection management may involve three-way handshaking, reference numbers, and timer-based mechanisms. Ordering, reliability, and flow control of data transfer can be achieved by mechanisms using sequence numbers, with error recovery based on positive or negative acknowledgment strategies, usually with timers. Sliding send and receive windows are used to increase throughput.

The ISO, in cooperation with other organizations, is developing standard protocols corresponding to the various layers of its reference model. Earlier work, such as the development of the Internet by the U.S. Department of Defense [Cer83] as well as systems designed by various computer companies (e.g., IBM's SNA [Gra79], DEC's DECNET [Wec80], and Xerox's Pup [Bog80]), has resulted in protocol architectures that are functionally similar to, but incompatible with, each other and the proposed ISO protocols.

There is a need to specify protocols in a machine-independent way, particularly in the data communications domain where many different computer systems may need to use a particular protocol. This has encouraged the development of many *formal description techniques* (FDT's) for specifying protocols.

The role of FDT's is discussed in [Vis83], and a survey is given in [Sun81]. Most FDT's in current use are based on finite state automata (FSA). Pure FSA are amenable to automated protocol verification, but can express only simple or idealized protocols. Other FDT's augment the automata with high-level language (HLL) code to express the details of real-world protocols.

1.1. Summary of Results

We have developed a new FDT for specifying and implementing data communications protocols. This FDT is called "Real-Time Asynchronous Grammars" (RTAG), and is based on a context-free grammar (CFG) notation in which ordinary terminal symbols correspond to messages sent and received. RTAG also provides convenient mechanisms for specifying concurrent protocol activities and real-time constraints.

RTAG has significant advantages over FSA-based FDT's. It encourages a top-down approach to designing and specifying protocols, in which a protocol is decomposed into (possibly concurrent) *subprotocols* which can be specified separately. In addition, RTAG allows more protocol mechanisms to be expressed without resort to HLL code; this is in part due to its more powerful underlying formalism (CFG versus FSA).

A software system based on an *RTAG compiler* and an *RTAG parser* has been developed. Applied to an RTAG specification of a protocol, these programs provide a major part of an implementation of the protocol; essentially only packet

assembly/disassembly and interface routines need be added. The RTAG parser responds to incoming messages by attempting to "derive" them according to the grammar. As a side-effect it may generate outgoing messages. Special terminal symbols represent certain amounts of real time, and the parser attempts to derive these symbols when the time intervals elapse.

If a set of protocols have been specified with RTAG, implementations of these protocols on a computer system can be obtained by writing an RTAG parser and interface routines on that system. Conversely, changes to a protocol running on a network of (possibly dissimilar) systems all running the RTAG parser can be effected by changing the RTAG specification, rather than by rewriting many hand-coded protocol implementations.

We have implemented an RTAG compiler and parser under 4.2 BSD UNIX [UNI83] and have written an RTAG specification of the NBS Class 4 Transport Protocol, TP-4 [NBS83]. The RTAG parser has been installed in the UNIX kernel and interfaced with its networking code, yielding an RTAG-based implementation of TP-4. This implementation has successfully communicated with other TP-4 implementations over the DOD Internet.

1.2. Organization

This thesis is organized as follows: Chapter 2 provides an overview of previous work in protocol specification. Chapter 3 gives a brief overview of RTAG, and Chapter 4 describes its syntax and informal semantics in more detail.

Chapter 5 explains the TP-4 specification. Chapter 6 gives the design of an RTAG parser, and Chapter 7 discusses RTAG software tools and their integration into debugging, prototyping, and production environments. Chapter 8 gives performance figures for the TP-4 implementation and proposes techniques for increasing performance. Chapter 9 contrasts RTAG with related work, and Chapter 10 summarizes RTAG and suggests areas for further work.

CHAPTER 2

PROTOCOL SPECIFICATION

2.1. Limiting the Area of Interest

This thesis is concerned with an FDT suitable for specification of data communications protocols above the level of packet formatting, with an emphasis on automated implementation. We do not address protocol design issues, except to discuss whether (and how) particular FDT's can express common protocol mechanisms. Furthermore, we are concerned with single protocol entities in a layered protocol system, which directly interact only with adjacent entities rather than with peer entities (see Figure 1).

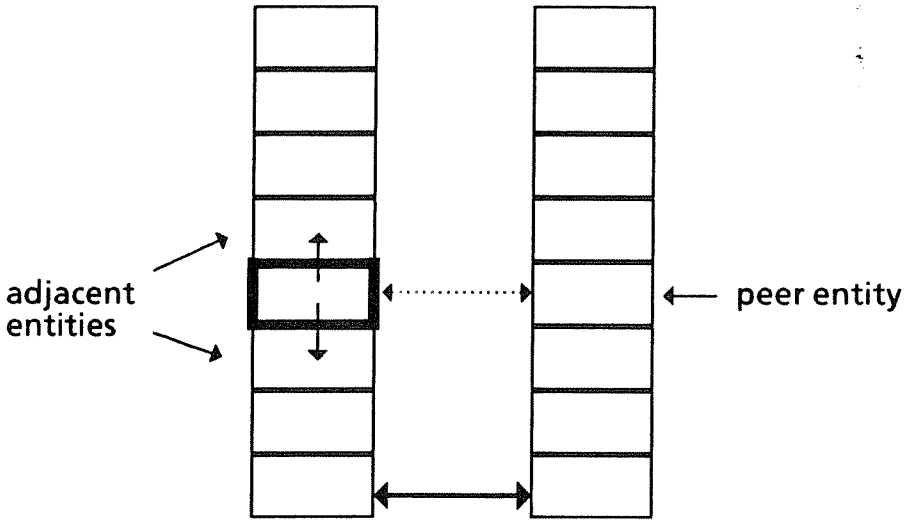

Figure 1: Entity relationships

This section motivates formal specification of standard protocols, and lists types of semantic information which FDT's may express (a more detailed discussion can be found in [Boc80]). A survey of previous work in the area is then given.

2.2. Goals of Formal Specification

One goal in specifying a standard protocol is to allow system programmers to create compatible implementations of the protocol. Since there is a wide variation in existing programming language and operating system environments, an FDT should not depend on a specific language or environment. One way to achieve the goal is to provide a means for automated implementation generation

which is portable to different environments.

A second goal of a protocol specification methodology is to support some form of protocol verification: i.e. to provide a logical framework for proving that a protocol fulfills its "service specification" given assumptions about the underlying communications medium, or proving that the protocol is free from specific problems such as undefined transitions, deadlock, livelock, or unbounded outstanding messages.

A third goal is to support study of the performance of protocol systems by simulation or analytical methods. These are actually two very different goals, since realistic simulation requires a detailed "executable" specification, whereas analytical methods require a simple FDT (such as bounded Petri nets or FSA) amenable to Markov chain analysis or other analytical techniques.

Protocol specification can be seen as an extension of program specification [Jon80]. However, conventional program specification methodologies do not necessarily provide a good basis for protocol specification because 1) they cannot express real-time semantics, and 2) their emphasis is for the most part on computation rather than communication, and the complexity of protocols lies largely in the latter.

2.3. Models of Protocol Activity

We now examine in detail the idea of "event" which underlies protocol specification semantics, and classify the semantic features of existing FDT's.

2.3.1. Semantics of Events

The underlying model for virtually all FDT's (including RTAG) is that of atomic message-passing *events* between adjacent entities. An *output event* is the sending of a message by the entity being specified, and an *input event* is the receipt of a message. Each message is assumed to have a *type* or *name* from some fixed set, and may carry other information as well.

Events involve pairs of entities that, because we are considering layered protocols, exist on the same machine. Many different semantics for message-passing (i.e. for events) exist. Scott [Sco85] gives a useful taxonomy for these semantics. A sender may block until receipt, block until reply, or not block. Receipt may be explicit or implicit. Different delivery mechanisms, both reliable and unreliable, may be used. And, in the single-processor case, either the sender or the receiver may resume running after a non-blocking send.

Most or all of these possibilities are realized in real systems. For example, in the 4.2 BSD UNIX networking system code, four distinct event semantics exist:

(1) A message between two protocol layers (say TCP and IP) is a procedure call, so delivery is reliable and the sender blocks until reply.

(2) A message from a bottom-level protocol (e.g. IP) to a network interface driver is delivered by a non-blocking append to a bounded-length queue, followed by a call to the driver. Delivery is unreliable since, if the queue is

full, the message is discarded without notification.

(3) Message delivery from a network interface driver to IP is also unreliable because it uses a bounded-length queue. After sending, the driver continues to run, and IP is scheduled for later execution by a "software interrupt" mechanism.

(4) A message from a top-level protocol to a user program is reliable, assuming that the protocol does flow control based on available buffer space. The user process is notified using the *wakeup* mechanism, which allows the sender (i.e. the protocol) to continue to run.

The correct operation of a protocol may depend on a particular event semantics: for example, block-until-reply semantics may be necessary to provide inter-layer flow control. In addition, the possibility of unreliable message passing must be reflected in the design of the protocol. In spite of these factors, existing FDT's (including RTAG) leave event semantics unspecified. One justification for this is that the semantics are usually provided by the existing operating system rather than by the protocol entity (this is true in 4.2 BSD UNIX).

2.3.2. Components of Protocol Specifications Semantics

There are several levels of detail which an FDT may express. Different aspects of a protocol may be relevant in different applications. For example, a specification to be used in performance analysis might not specify how data to and from higher levels is divided, concatenated, or buffered within the protocol

entity. This specification would be crucial, however, for implementation.

The semantic models of existing FDT's are based on the following components, and can be classified according to which components they include:

Event order constraints

All protocol specifications impose some constraint on the temporal ordering of event occurrences. In the simplest model, only event names are considered and the specification thus defines a language over the alphabet of event names, i.e. those event sequences that satisfy the ordering constraint.

Real-time constraints

A protocol may constrain the precise real time intervals between certain event occurrences. Examples include timeout-driven retransmission, "reference waits" before re-using reference numbers, and the "quiet time" observed by a host after it recovers from a crash.

Data dependencies

In real protocols, messages often contain data such as integers or variable-size byte strings. This additional information may be destined for an upper layer client, or it may be part of a protocol mechanism (such as the sequence number in an acknowledgement message). Formal models for such protocols must be based on parameterized events, and their ordering and real-time constraints may involve these parameters.

Packet formatting

A complete specification of a real-world protocol must include details such as the format of packets, ordering of bytes within a word-size integer, checksum algorithms, etc.

Underdetermined Specification

A protocol specification need not give a complete description of a deterministic machine that implements the protocol. Instead, it may allow a range of entity behaviors. For example, existing specifications of transport protocols allow a range of acknowledgement strategies. Such a specification may impose as well a probabilistic or "fairness" constraint on allowable behaviors ([Mol82], [Par84]).

2.4. Previous Work in Protocol Specification

2.4.1. Finite-state automata

Event-order constraints are often modeled by finite-state automata ([Bjo70], [Boc78], [Dan80]). Each automaton transition is triggered by an input event, and may generate one or more output events. The effect of timeouts on event order (but not the actual timeout intervals) can be simulated by ϵ transitions. Figure 2 shows an FSA description of both ends of an alternating bit protocol.

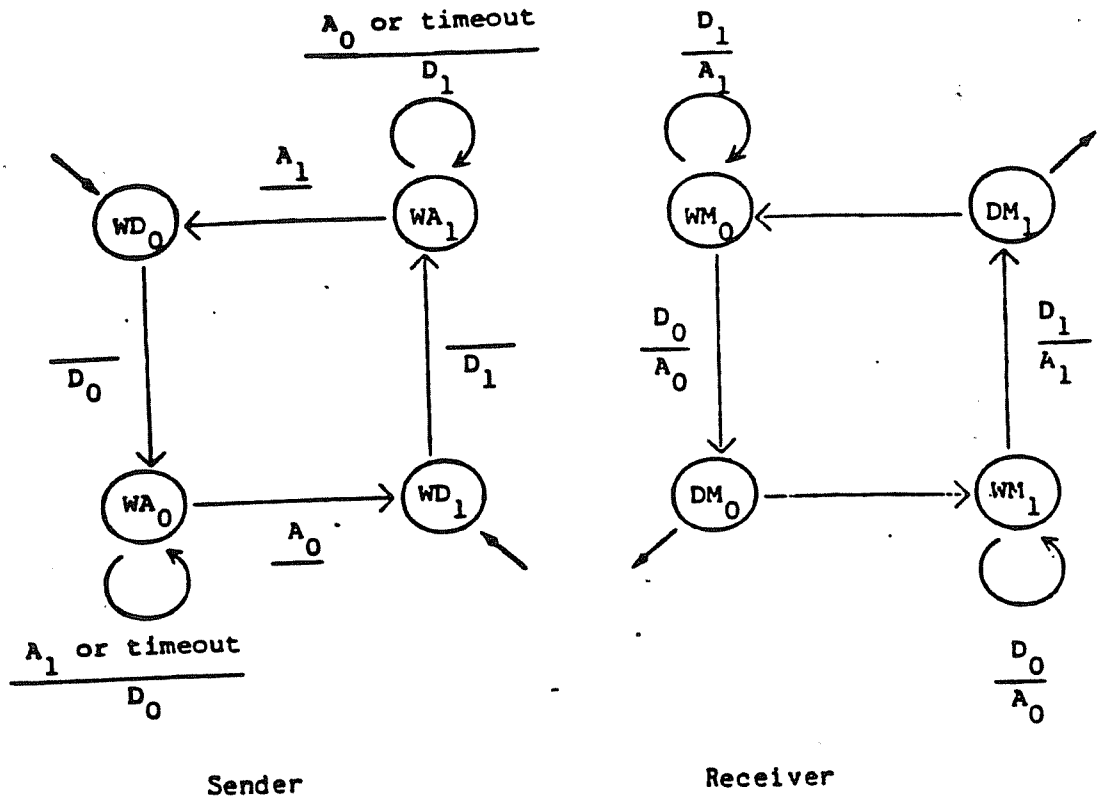

Figure 2: FSA description of
an alternating bit protocol
(from Yemini and Nounou [Yem83])

A more complex example, shown in Figure 3, is the FSA for the connection and disconnection phases of one TP-4 transport connection, taken from the National Bureau of Standards (NBS) TP-4 specification [NBS83].

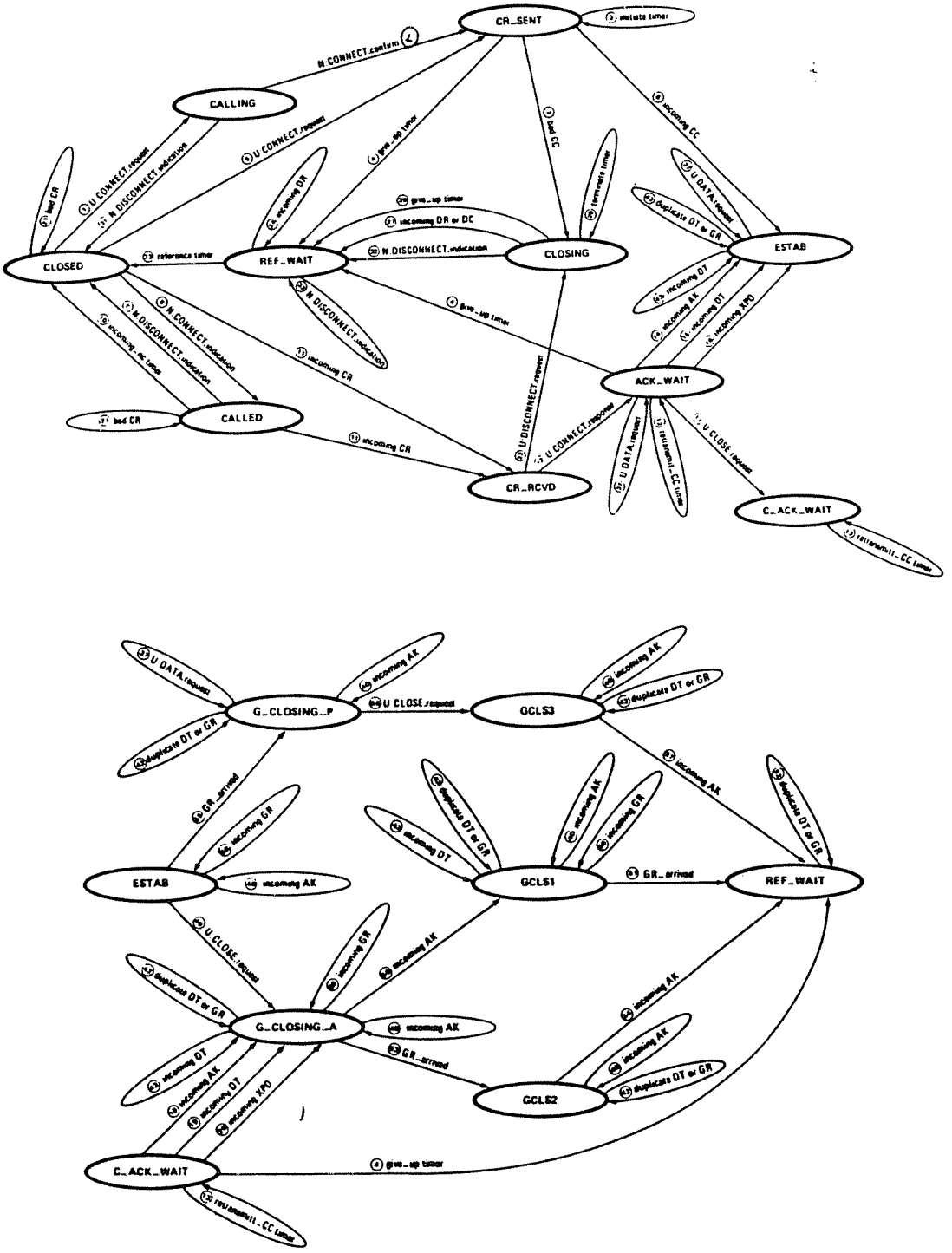

Figure 3: FSA for TP-4
(from NBS specification [NBS83])

Pure FSA cannot express data-dependent protocol mechanisms; for example, the FSA for TP-4 does not specify the sliding window mechanism. To deal with this problem, FSA can be extended so that the "state" includes a set of variables, and each transition includes a HLL guard condition and a HLL code segment to be performed when the transition is performed, both of which may refer to state variables and event parameters. This approach is called "augmented finite state machines" (AFSM).

AFSM-based FDT's generally use explicit timers to specify real-time protocol mechanisms such as timeouts. HLL code segments can start and cancel named timers, and timeouts are treated as input events which trigger "timeout" transitions.

Pure FSA allows verification by various state-space exploration techniques, often on a "product" FSA representing two entities ([Rub82], [Sid83b]). Logic programming systems such as Prolog can be used for this purpose ([Sid83a], [Log84]). West and Zafiropulo [Wes78] use FSA to model and verify certain aspects of the CCITT X.21 protocol. State-space exploration can be applied to data-independent parts of an AFSM specification, but other techniques are needed in general ([ScF80], [Hai80]).

Blumer and Tenney [Blu82] describe an AFSM-based specification project whose goals include automatic implementation and verification. Pascal is used as the embedded HLL. They describe software for converting an AFSM

description to a C program that implements the protocol; this program consists of C code simulating the FSA, intermixed with Pascal-to-C translation of the embedded HLL code. They have used this system to specify and implement the TP-4 transport protocol, and report that approximately 40% of the final C code was generated automatically, with execution speed comparable to manual implementations.

A similar project undertaken by IBM is described in several papers ([Sch80], [Poz82], [Nas83]). The project involves the Format and Protocol Language (FAPL), which has been used to create an "executable specification" of IBM's Systems Network Architecture (SNA).

FAPL is an extension of PL/1. It can be used to specify multiple protocol layers, and its runtime system handles scheduling and communication between layers. Within a layer, FAPL's capabilities are equivalent to AFSM, except that it does not include HLL enabling conditions. Instead, the HLL code associated with a transition can decide to "back out" of the transition, or change the target state. FSA specification is built directly into the programming language instead of being viewed as an exterior structure. An FAPL compiler translates FAPL source into a sequential target language (PL/1 or related languages). Implementation-dependent functions and interface routines are coded directly in the target language. The papers mention using the resulting code for simulation (both for verifying the SNA protocols and for testing applications that use SNA)

and for production use.

2.4.2. Grammars-based systems

Context-free grammars (CFG's) can be used to specify valid event orders. Harangozo ([Har77], [Har78]) uses a CFG-based FDT to specify a portion of the HDLC data-link protocol. An attribute-like notation is used to represent sequence numbers. Symbols and productions are parameterized by variable subscripts, and each production denotes the family of productions obtained by instantiating the variables with all values in their domains. There is not provision for real-time constraints.

Teng and Liu [Ten78] use CFG's (without attributes or real-time constraints) to specify some simple protocols. They propose using these specifications for implementation and limited verification, but no results are given.

2.4.3. Algebraic systems

The "Calculus of Communicating Systems" (CCS), developed by R. Milner [Mil80], is an algebraic system whose basic elements represent input/output events. Event-ordering constraints on entities or sets of entities are represented by "behavior expressions" built up from certain operators, or by systems of equations involving these expressions. Milner defines a notion of "observational equivalence" relevant to nondeterministic entities, and gives a system of algebraic rules allowing equivalences between behavior expressions to be proved.

Yemini and Nounou [Yem83] discuss a "protocol development environment" where several FDT's (FSA, Petri nets, high-level language) are available for protocol specification. They propose the development of tools to convert these forms into a "canonical semantic model" based on CCS, that can then be analyzed for correctness and performance by formal methods.

CCS also serves as the basis for LOTOS, an FDT developed by an ISO subgroup for the purpose of specifying the ISO protocols [Bri85]. To express data dependencies, LOTOS augments CCS with an abstract data type mechanism. LOTOS has no real-time constraint mechanism. It has been used to give a service specification of the ISO transport layer [Bri84]. No work involving LOTOS-based automatic implementation has been reported.

2.4.4. Petri nets

Petri nets [Pet77] are useful for modeling concurrency and synchronization in communication systems, and provide a basis for correctness and performance analysis ([Mer79], [Dan77]). Figure 4 shows a Petri net used by Molloy [Mol82] to analyze the performance of an alternating bit protocol.

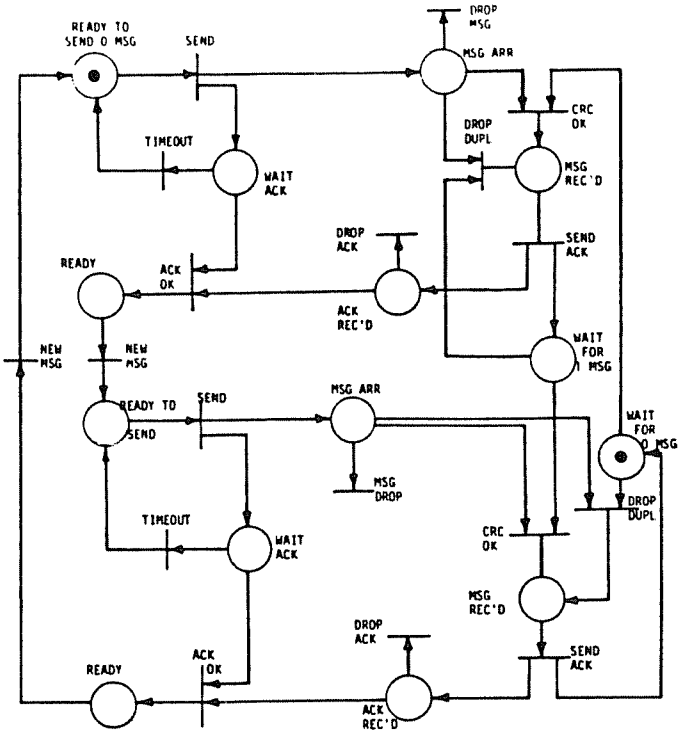

Figure 4: Petri net for an alternating bit protocol (from Molloy [Mol82])

The Petri net formalism can be extended in several ways to express real-time and data-dependent constraints. Ozsu and Weide [Ozs82], for an application involving simulation-based performance study of concurrency control mechanisms, use an extended Petri net formalism in which tokens may have attributes. The transitions in the net have associated HLL code which is used to generate attribute values for tokens produced by the transition and to decide the routing of these tokens. A similar FDT based on "predicate/transition nets" has been used to model simplified versions of OSI protocols [Bur84].

2.5. Formal Specification of Packet Formats

This thesis does not address the problem of specifying packet formats, or of automating packet assembly based on such specifications. However, this is an important goal, particularly for protocols (such as presentation-level protocols) where packet structure is complex or the mapping between host system format and network format is nontrivial. Work in this area is being done: a standard formalism for packet format is embodied in the CCITT Draft Recommendation X.409, and an automated software system for packet assembly/disassembly has been developed [Pop85].

CHAPTER 3

RTAG OVERVIEW

The successful automated-implementation projects to date have used AFSM. After using an AFSM-based specifications in an implementation project, we concluded that this approach has two major drawbacks: 1) it forces the description of independent protocol mechanisms to be intertwined, and 2) because of the limited power of FSA, too much detail has to be put in HLL. We feel that by using a more powerful underlying formal component, such as attribute grammars, these problems can be avoided.

3.1. Goals

RTAG, (introduced in [And84]), is a formal description technique for protocols developed with the following goals:

(1) To allow detailed description of real-world communications protocols, while avoiding, as much as possible, the complexity of a general-purpose programming language.

(2) To support structured protocol design. Specifically, to allow a protocol to be recursively decomposed into serial and concurrent subprotocols that are specified separately.

(3) To facilitate the development of software for generating usable protocol implementations based on formal specifications.

Of the five semantic components described in Section 2.3.2, RTAG addresses the first three: event order constraints, real-time constraints, and data-dependence. It does not address the specification of packet formatting, or the specification of complex data types for messages: messages are restricted to having a fixed set of data fields, each of which has a simple type. RTAG is not intended for expressing underdetermined specifications, although it is possible to write underdetermined (ambiguous) RTAG specifications.

3.2. Protocol Specification by Attribute Grammars

An RTAG specification is based on an underlying context-free grammar in which most terminal symbols correspond to message events. These symbols are called *input* or *output* symbols depending on whether the message is received or sent by the protocol entity being specified.

RTAG grammars generate event sequences allowed by the protocol. For example, the production

<goal> : [net→data] [user←data] <goal>.

might be part of a trivial protocol that responds to messages from the network layer (the **[net→data]** input event) by sending a message to the upper layer (the **[user←data]** output event).

Each RTAG symbol has an associated set of *attributes,* which in each symbol instance are instantiated with data values. The attributes of an input or output symbol correspond to the fields of the associated message. A production may have a boolean-valued *enabling condition* involving attributes values, which must evaluate to **true** for the production to be applied. Each production can also have an associated set of *attribute assignments* which specify its side-effects on attributes, and which are performed when the production is applied.

Each grammar symbol can be thought of as representing a "subprotocol" consisting of the events sequences it can generate. Productions can compose subprotocols in sequence:

<x> : <y> <z>.

or in parallel:

<x> : {<y> <z>}.

In the first case, the events derived from <y> must precede (in real time) the events derived from <z>, whereas in the second case the event sets may be interleaved.

Real-time constraints, such as timeouts, can be represented in RTAG using a special terminal symbol **/timer/**, which has an attribute representing the length, in clock ticks, of an idle time interval.

3.3. Implementations Based on RTAG Specifications

An *RTAG parser* is a program which, given an RTAG specification, attempts to enforce the specification, and thus to implement the protocol. It processes input events and handles timers. Output symbols can be thought of as "action routines" which are performed as a consequence of applying productions. An algorithm for an RTAG parser will be given in Chapter 6. There are major differences between an RTAG parser and a parser for a CFG with action symbols. This distinction will be explored in Section 9.1.

CHAPTER 4

RTAG SYNTAX AND SEMANTICS

This section gives an informal summary of the syntax and semantics of RTAG. A precise syntax, in the form of *Lex* and *YACC* files for RTAG, is given in Appendix II. A more precise semantics could be given, perhaps by showing a semantics-preserving transformation from RTAG to some version of CSP, the semantics of which have been well studied [Hoa78].

4.1. Alternative Models for RTAG Semantics

An RTAG specification defines the behavior of a protocol entity. This entity must respond to input messages (and to the passage of real time) by generating output messages. The mapping from an RTAG specification to the set of entity behaviors which are allowed by the specification constitutes the *semantics* of RTAG.

In spite of the informality of our treatment of RTAG semantics, we nonetheless need a solid conceptual model on which to base it. There are several possibilities:

(1) The *denotational* approach, in which a mathematical object S_X is assigned to each grammar symbol X, by rules that recurse on productions. S_X must fully describe the "subprotocol" represented by X, in terms of its

interactions both with adjacent entities and with other subprotocols within the same entity. We have tried this approach, using as the structure of S a predicate over event sequences and time-dependent attribute values, but have found that it leads to considerable complexity, with no apparent theoretical or pedagogical benefit.

(2) The *parser-based* model: we give in Chapter 6 the design of an RTAG parser, a real-time program that maintains an attributed parse tree, and responds to input events by attempting to derive them from leaf nonterminal symbols. The semantics of RTAG could be defined as the actions of this parser.

(3) The *process-based* model in which each grammar symbol X is associated with a process definition P_X. Event symbol processes read or write messages, and each production defines a process as the sequential or parallel composition of subprocesses. For example, the production

<x> : <y> <z>.;

defines $P_{<x>}$ as a process that invokes processes $P_{<y>}$ and $P_{<z>}$ in sequence. The attributes of X correspond to per-process variables of P_X. The protocol defined by an entire RTAG specification is realized by executing the process associated with the goal symbol.

The parser- and process-based models are connected in that the parse tree of the former will always match the process ancestry tree of the latter, modulo

design decisions on precisely when records are created or deleted. In the process-based model, RTAG is seen as a specialized concurrent programming language; this is related to the parser-based approach by viewing the RTAG parser as an interpreter and process scheduler for this language. Conversely, the process-based model can be seen as defining a type of recursive-descent parser, in which multiple branches can be simultaneously active.

As a basis for describing informal RTAG semantics, the process-based model has these advantages over the parser-based model: 1) it is higher-level (i.e. it hides details of scheduling and interpretation) and offers a conceptually simpler view to the RTAG user; 2) because the process-based model says less about implementation details, it is more amenable to interpretation in different programming models such as object-based or logic programming.

For the above reasons, our informal RTAG semantics will use the process-based model, and RTAG features will be described as components of process definitions. However, it seems to be easier to talk about implementation details in parser-based terminology, and so Chapter 6 describes algorithms for an RTAG parser.

4.2. Details of the Process Model for RTAG

We now discuss the process-based model in more detail. First, it should be noted that the processes are very lightweight: they are essentially restricted to executing in-line code. and their memory usage, outside of subprocess

invocation, consists of a fixed set of per-process variables. RTAG processes would probably not be implemented as distinct operating-system-level processes.

In an RTAG protocol entity there is at any point an ancestral tree of process *instances*; initially there is exactly one instance of the *goal* process, i.e. that corresponding to the goal symbol. A process is first *instantiated*, creating its per-process variables; it is later *started*, i.e. executed. It becomes *finished* when it reaches the end of its definition or because of intervention by another process. The per-process variables continue to exist until the process is *removed*. Processes can *block* waiting for input events to happen, time to elapse, or expressions to become true.

Assuming, as we are, that the RTAG protocol entity resides on a single machine, the scheduling of its processes must be considered. When a message arrives, for example, there may be a choice of newly-unblocked processes to execute. The execution of the chosen process may unblock still other processes. We make the following restrictions on RTAG (not operating system) process scheduling:

(1) It is non-preemptive; context switches are allowed only at the point of starting a subprocess, and when a process blocks. All processing resulting from a particular event (message arrival or timeout) is done atomically.

(2) When an input event is being handled, processes capable of reading it have priority over other processes.

(3) Newly-started processes are prioritized by the order of declaration of their symbol (earlier declaration gives higher priority), and the highest-priority process is always executed first.

For the purpose of discussing RTAG's real-time semantics, we assume that the CPU time used by RTAG processes is negligible relative to real time. Without some assumption of this sort, the real-time constraints expressed in an RTAG specification would be meaningless. The assumption is not unreasonable in applications where the protocol runs at high priority in a memory-resident kernel, and does little computation.

The remainder of this section lists RTAG's features. The description of each feature gives the syntax, and discusses the semantics in terms of the mapping from a symbol X to the associated process definition P_X.

4.3. Symbols and Attributes

RTAG has the following symbol classes:

(1) *Nonterminal symbols* are delimited by matched angle brackets (<, >).

(2) *Event symbols* (both input and output) are delimited by matched square brackets ([,]). Names should suggest whether the symbol is an input or output symbol, and what other entity it involves. For example, in our TP-4 specification, **[U→CR]** represents a connection request received from the upper layer, and **[N←DT]** represents a data packet sent to the network layer.

(3) *Special terminal symbols* represent internal actions of the protocol entity; their names and are delimited with slashes. There are two special terminal symbols: **/timer/** and **/remove/**.

Each symbol has an associated set of typed attributes, the names of which must be valid C identifiers [Ker78]. The following attribute data types are used: **integer, boolean, dataptr** (pointer to a variable-size byte string), and **symbol-ref** (pointer to a symbol or process instance; used by the **/remove/** special terminal symbol, see below).

Here is an example of a symbol declaration section:

```
goal     <connection>          /* declare "goal" nonterminal symbol */

nonterm <send>                 /* declare another nonterminal */
         int      seqno        /* with one integer attribute */

input    [U→CR]                /* declare an input terminal symbol */
         int      refno

output   [N←DT]                /* declare an output terminal symbol */
         int      refno
         dataptr  data
```

The semantics of terminal symbols are as follows:

- If X is an output symbol, P_X sends the associated message, using the attribute values of X as values for the message fields. In practice, this sending is done by calling an *event performance routine*, passing the attribute values of X as parameters to the routine. Output symbols can therefore be thought of as "action symbols".

- If X is an input symbol, P_X blocks until an event occurs consisting of the arrival of the message represented by X. P_X is then finished, with its per-process variables set to the values contained in the message fields, and is said to have *accepted* the message.
- **/timer/** has an integer attribute *interval*. $P_{/timer/}$ sleeps for the amount of time given by the value of this attribute.
- **/remove/** has an attribute *where* of type *symbol-ref*, which points to a symbol (process) instance X. When an instance of $P_{/remove/}$ is executed, X and all its descendants in the process tree are flagged as finished. This is used, for example, in handling abnormal closure of connections.

4.4. Attribute References and Expressions

Expressions associated with a production P can refer to symbols and attributes of symbols either in P, or at a relative position in the process tree. Symbol references are called *local* and *non-local* accordingly. A local reference is of the form **$n**, and refers to the nth symbol of P; **$0** is its parent (LHS) symbol and **$1** is the first symbol of the RHS. A nonlocal reference is written as **sym1/sym2/ ... /symn**, the semantics of which are as follows: find the closest ancestor named *sym1* of the parent of P; find the leftmost of its children named *sym2*, the leftmost of that process's children named *sym3*, and so forth. A reference to a nonexistent process (which can be detected at runtime) is considered a fatal error.

Attribute references are of the form **symbol-reference.attrname** and refer to the named attribute of the process instance specified by *symbol-reference*.

Expressions in RTAG are formed from constants (**empty** is a *dataptr* constant denoting the empty string), symbol and attribute references, integer operators (+, −, mod), logical and relational operators (written as in C [Ker78]), a *dataptr* operator (catenation, written as **cat**), externally defined functions (Section 4.9) and parentheses.

4.5. Simple Productions

A simple production (i.e. the unique production rule for its LHS symbol) is written as:

```
<x>  :  α.
;
```

where **<x>** is any nonterminal and α is a string of grammar symbols and optional curly brackets.

If **<x>** has no other productions, $P_{<x>}$ is the composition of the processes of the grammar symbols in α (with possible modifications due to the presence of "attribute assignments" or an "enabling condition" in the production, see Sections 4.6 and 4.7). α may be empty, in which case $P_{<x>}$ is considered to consist of an invocation of an *epsilon process* which is immediately finished.

When $P_{<x>}$ is started, its subprocesses are instantiated but not started. In the absence of curly brackets, the process composition is sequential, i.e. each subprocess is started when its left neighbor finishes. Hence left-to-right syntactic order corresponds to real-time order. For example, in the process defined by

```
<x>  :  <y> <z>.
;
```

execution of $P_{<y>}$ will strictly precede that of $P_{<z>}$.

Processes may also be composed in parallel, allowing their events to overlap in time. A group of RHS symbols may be surrounded by curly brackets, forming a *concurrent group*; the associated process executes the component processes in parallel, and waits for them all to finish. Concurrent groups may contain only nonterminal symbols; this restriction simplifies the semantics of attribute-related constructs (Sections 4.6 and 4.7).

As an example, in

```
<x>  :  {<a> <b>}  <c>  {<d> <e>}.
;
```

the processes $P_{<a>}$ and $P_{<b>}$ start together and are executed in parallel; when both are finished $P_{<c>}$ starts, and when it is finished $P_{<d>}$ and $P_{<e>}$ start together; when both are finished $P_{<x>}$ is finished.

4.6. Attribute Assignments

A production can have zero or more "attribute assignments", each of the form **attribute-reference = expression**. Attribute assignments are used to transfer information between process instances. *Performing* an attribute assignment consists of evaluating the expression and storing the value in the referenced attribute.

The order of subprocess instantiation and attribute assignment performance is crucial to RTAG semantics. When a process starts, its subprocesses are all instantiated and their attributes are initially undefined. In the general case, attribute assignments are performed prior to starting the first subprocess. Assignments are performed in the order in which they appear in the specification.

There are two special cases for attribute assignments semantics. First, in a production of the form

```
<x>  :  /timer/ α.
        $1.interval = expression
        (other attribute assignments)
;
```

the assignment of **/timer/**.interval is performed when $P_{<x>}$ starts. When $P_{/timer/}$ finishes, the subprocesses in α are instantiated, and the other attribute assignments are performed.

Second, in a production of the form

```
<x>  :  [y] α.
        (attribute assignments)
;
```

where [y] is an input event, $P_{<x>}$ initially invokes $P_{[y]}$. The subprocesses in α are instantiated, and the attribute assignments are performed, only after [y] is finished, i.e. after an input event has occurred and been accepted.

4.7. Enabling Conditions

A production of a symbol X can have a boolean-valued *enabling condition*. P_X blocks, without instantiating subprocesses or performing attribute assignments, until the value of the enabling condition is **true**, at which point P_X is unblocked. If the condition is **true** but other processes cause it to revert to **false** before P_X is scheduled for execution, then P_X remains blocked.

For example, in

```
<x>  :  [N-DT] <z>.
        if $0.seq > <y>.windowstart
;
```

$P_{<x>}$ blocks until the *seq* attribute of **<x>** is larger than the *windowstart* attribute of the closest **<y>** ancestor, then instantiates **[N-DT]** and **<z>** and starts **[N-DT]**. In the general case, enabling conditions can refer only to attributes of processes which exist when **<x>** is started (in particular, attributes of **<x>** and its ancestors).

If the first symbol of the production RHS is an input symbol [y], the semantics are also slightly different. The enabling condition may depend on the attributes of [y] as well as those of existing processes. $P_{<x>}$ blocks until an event occurs consisting of the arrival of a message of type [y] whose attribute values satisfy the enabling condition. At this time subprocesses are instantiated and attribute assignments are performed.

4.8. Alternative Productions

A nonterminal <x> may serve as the parent of several "alternative" productions, denoted as follows:

```
<x> :    α1.
         (enabling condition)
         (attribute assignments)
         |    α2.
         ...
;
```

The alternative productions can be thought of as alternative definitions of $P_{<x>}$. Intuitively, the process $P_{<x>}$ blocks until some alternative can make "significant progress", at which point that process definition is executed, and the other alternatives are discarded.

In the context of a process instance <x> which is defined by alternative productions, we will say that an alternative P *yields* a symbol Y if execution of P leads, without blocking and without starting more than one process per concurrent group, to instantiating and starting P_Y. In grammar terminology, this

means that Y can be left-derived from **<x>** by a sequence of enabled productions.

A process $P_{<x>}$, defined by a set of alternative productions, blocks until an alternative is *selected*, at which point that alternative is executed, and the others discarded. If more than one alternative is selected, the choice is nondeterministic. An alternative P is said to be *selected* when either

(1) an input event **[y]** occurs such that P yields an instance of **[y]** which accepts the message;

(2) P yields the epsilon process, an output symbol, or **/remove/**; or

(3) P is defined by a production of the form

```
<x> :     /timer/ α.
      (enabling condition)
      $1.interval = expression
      ...
```

and an instance of P, started when **<x>** is started, finishes **/timer/**.

As an example of alternative productions, consider the following:

```
<get ack>  :     [N→AK].

           |     .
           if <connection>.closed

           |     /timer/ <timed out>.
           $1.interval = 10
;
```

The semantics of $P_{<get\ ack>}$ are as follows: when an instance of **<get ack>** is

started,

a) if the [N→AK] event occurs, the first production is selected;

b) if the *closed* attribute of the nearest **<connection>** ancestor becomes true, the second (epsilon) production is selected;

c) if 10 time units elapse without either a) or b) taking place, then the third production is selected and a **<timed out>** process starts.

Hence there are three alternate definitions of the **<get ack>** process, and the choice of which to execute depends on what happens first (in terms of events, attribute value changes, and the passage of real time).

In practice, the only way to detect when an alternative process is selected according the rules 1) and 2) above, is to execute it when something (such as an attribute value change or message arrival) has happened that might cause it to become selected. All possible execution paths by which the types of processes listed in rules 1) and 2) could be reached must be tested. This execution must be "tentative" because if it fails to yield the necessary symbol (due to being blocked by an enabling condition) all resulting process instances must be removed, and attribute changes in other parts of the process tree must be undone.

For RTAG specifications to be useful, the "alternative selection" task must be effectively computable. Switching for a moment to grammar terminology, we point out that the production sequences of interest in rules 1) and 2) above are

left-derivations (in a modified sense to be discussed in Section 6.3). In order to ensure that the "alternative selection" task is effective, it suffices to prohibit left recursion (also in the sense of Section 6.3), since then the number of production sequences of interest is bounded and, since each sequence can involve no more productions than there are grammar symbols, the time needed to check a sequence is bounded by a constant. The theoretical maximum for this constant is large (exponential in the number of grammar symbols) but is not approached in practice.

For this reason, and because it did not seem to be useful as a specification mechanism, left recursion is prohibited in RTAG (this means that no process may yield itself, even in the underlying CFG). The loss of generality is that the protocol cannot perform a calculation of unbounded length to yield an input event.

4.9. Externally Defined Functions

External functions can be used in attribute assignments and enabling conditions as an "escape" into HLL code. They might be used for calculations that occur often, that are not easily expressed within RTAG, or that are installation-dependent. For instance, the TP-4 specification (contained in Appendix I) uses an external function for the cyclic "between" relation on sequence numbers. Function names are delimited with number signs; name, value type and parameter types must be declared. For example, the "between" function is declared by:

```
extern boolean #between#(int, int, int)
```

4.10. Multiple Acceptance of Input Symbols

Since an RTAG entity may contain many concurrent processes, message receipt semantics could be single-receiver (an instance of a message can be accepted by at most one process) or broadcast (all processes which can accept a message do so). Our experience has suggested that broadcast semantics for all input messages is sufficient for typical applications. Broadcast semantics are useful because, for certain natural subdivisions of complex protocols like TP-4, a single input message may be relevant to several subprotocol processes. For example, in our specification for the sending component of TP-4, the acknowledged delivery of each packet is handled in a separate process. A single acknowledgement event may serve to acknowledge several packets, and hence be relevant to several processes.

RTAG supports only broadcast semantics. It should be added that, as a result of the acceptance of a message X by a process, other processes may be instantiated and started that could accept a message of the same type as X. However, the next-generation processes cannot accept X since they start after X occurs.

If an input message is not accepted by any process, it is discarded and ignored. This is a reasonable default since most standard protocols must deal with packet duplication across networks, and therefore they ignore most types of

duplicates. In other applications it might be preferable to regard unaccepted messages as fatal errors and to abort the protocol, or to log them for debugging purposes.

4.11. Key Attributes

In our RTAG-based protocol implementations, the work done in processing an input event consists partly of computing a *candidate set*, i.e. a set of processes instances that are currently blocked in an input process or in alternative selection, but which may be able to accept the new message and proceed. Since not all candidates may in fact accept the message, and each test may be expensive, we seek an efficient way of computing a reasonably small candidate set.

A candidate set computed only on the basis of CFG information will be excessively large. For example, in a protocol handling multiple connections, the candidate set for an input message relevant to only one connection will include (if only CFG information is used) candidates from many other connections. Most protocols that handle multiple connections have a "reference number" mechanism for associating input messages with connections. While this could be enforced at the leaf level by having a reference number equality clause in each enabling condition, considerable overhead would be incurred.

RTAG includes a mechanism that can express the semantics of reference numbers, and that provides a means for calculating smaller candidate sets. This mechanism is as follows: an attribute name can be declared as **key**. No two

nonterminal process instances may simultaneously have the same value for the *key* attribute. If an input symbol instance X has the key attribute with value k, and there is a nonterminal instance Y with key attribute value k, X can be accepted only by descendants of Y. Otherwise (if no nonterminal instance has key attribute value k) X can be accepted only by processes of which no ancestor has the key attribute.

The key attribute mechanism is sufficient for transport protocols, where there is only one level of reference-numbering. An extension of the mechanism may be required for other applications.

4.12. Removal of Finished Processes

To achieve memory conservation, process instances must be eventually removed. Process removal is done in the following cases:

(1) When a nonterminal process finishes, the instances of its subprocesses (if any) are removed.

(2) When, because of direct or indirect recursion, a process instance X_1 has an ancestor X_2 with the same symbol name, and all of X_1's ancestors except those which are ancestors of X_2 are finished, then the process subtree rooted at X_2 is "grafted" in place of X_1, and the remainder of X_1's ancestors are removed (see Figure 5).

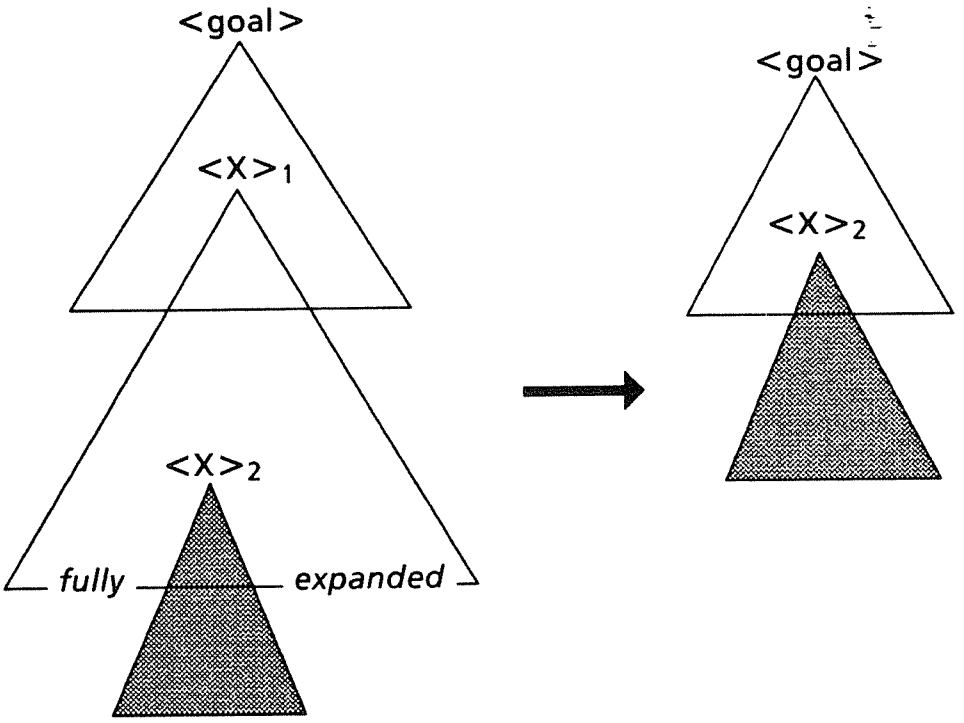

Figure 5: Removal of recursive processes

For typical recursive constructs, procedure 2) prevents unbounded memory usage. Process execution is not affected by either procedure. However, storing shared data in the attributes of a finished process, or in a process which is part of a recursive sequence, must be done with caution since such processes can be removed.

4.13. A Short Example: An Alternating Bit Protocol

An RTAG specification of the sending end of an alternating-bit protocol is given below as a small example. The declaration section is omitted for brevity.

```
<goal> : <packet tail>.
        $1.seqno = 0
;

<packet tail> : <packet> <packet tail>.
        $2.seqno = ($0. seqno + 1) mod 2;

        | [U-FINISHED] .
;

<packet> : [U-DATA] [N-DATA] <retransmit>
        $0.data = $1.data
        $2.data = $1.data
        $2.seqno = <packet tail>.seqno
;

<retransmit> : [N-ACK] [U-ACK].
        if $1.seqno == <packet tail>.seqno

        | [N-ACK] [N-DATA] <retransmit>
        if $1.seqno != <packet tail>.seqno
        $2.data = <packet>.data
        $2.seqno = <packet tail>.seqno

        | /timer/ [N-DATA] <retransmit>
        $1.interval = 100
        $2.data = <packet>.data
        $2.seqno = <packet tail>.seqno
;
```

CHAPTER 5

AN EXAMPLE: THE TP-4 TRANSPORT PROTOCOL

5.1. Design Principles

As an example and test case for RTAG, we have written a specification of an almost complete subset of the NBS class 4 transport protocol (TP-4), working from an AFSM specification of the protocol [NBS83]. The entire specification is given in Appendix I; in this section we explain the mechanisms in detail.

The TP-4 example illustrates the following principles for protocol design using RTAG:

(1) Logically distinct parts of the protocol, which we call *subprotocols*, are put in different processes (i.e., subtrees). RTAG provides a means for abstraction of subprotocols (i.e. the external interface of a subprotocol can hide its internal mechanisms) and encourages a top-down design approach.

(2) Information relevant only to a particular subprotocol is stored in the attributes of the process (i.e. symbol) serving as the root for that subprotocol, rather than higher in the tree.

5.2. Summary of TP-4

TP-4 was chosen as a test case because it incorporates many common protocol mechanisms and offers a significant level of complexity. TP-4 provides multiple connections, reliable connection management, reliable sequenced message transfer with end-to-end flow control, a separate logical channel for "expedited" (high-priority) data, and sliding send and receive windows for increased throughput. Messages between TP-4 peer entities are called *transport protocol data units* (TPDU's). There are a dozen or so types of TPDU's, including connection request (CR), connection confirmation (CC), data (DT), acknowledgement (AK), and graceful close (GR). Data messages passed between TP-4 and upper layer clients are called *transport service data units* (TSDU's).

The TP-4 subset we have specified comprises a complete and functional protocol, and can communicate with implementations of the full protocol. Certain nonessential features have been omitted for simplicity. These simplifications are as follows:

- Subsequence numbers for AK TPDU's are not used, and hence spontaneous shrinking of the peer receive window is not handled correctly.
- Timeout intervals are fixed.
- Only 7-bit sequence numbers are used.

In the 4.2 BSD UNIX protocol system, the adjacent entities are IP below and the UNIX socket system (which provides buffering and process synchronization) above. The interfaces (i.e. message sets) used by our RTAG TP-4 specification are chosen to reflect these entities, and deviate from those in the NBS TP-4 specification. For example:

- IP provides a datagram services, while the NBS specification assumes a virtual circuit network service.
- The NBS specification uses a model in which data is passed from the upper layer via "descriptors" which are read asynchronously, whereas the socket system keeps all buffers in kernel memory.
- The socket system uses a "listening port" model of connection establishment which differs from the NBS model.

The fact that the specification had to be "tailored" to our host environment is not completely satisfactory. However, the differences are to some extent unavoidable, and in any case they are small and fairly well localized in the RTAG specification.

5.3. Naming of Event Symbols

A mnemonic naming convention for terminal symbols is used. **[U→CR]**, for example, is a connection request message from the upper layer. The direction of the arrow distinguishes input and output events, and the first letter indi-

cates the other entity involved (N for network, U for upper layer). Each type of TPDU has event symbols for its sending and receipt (**[N←CR]** is the arrival of a CR TPDU from the network layer, **[N←DT]** is the sending of a data TPDU to the network layer, etc.). The declaration portion of the specification (Appendix I) gives the complete set of event symbols and their attributes.

5.4. Multiple Transport Connections

A timer for the production of **<goal>** provides an initial inactive period which protects the integrity of connections. The first production of **<TC tail>** allows many TP-4 transport connections to exist concurrently:

```
<goal>          :       /timer/ <TC tail>.
        $1.interval = QUIET_TIME
;

<TC tail>       :       {<TC> <TC tail>}.
                |       [U←FIN] .
;
```

This produces the structure shown in Figure 6.

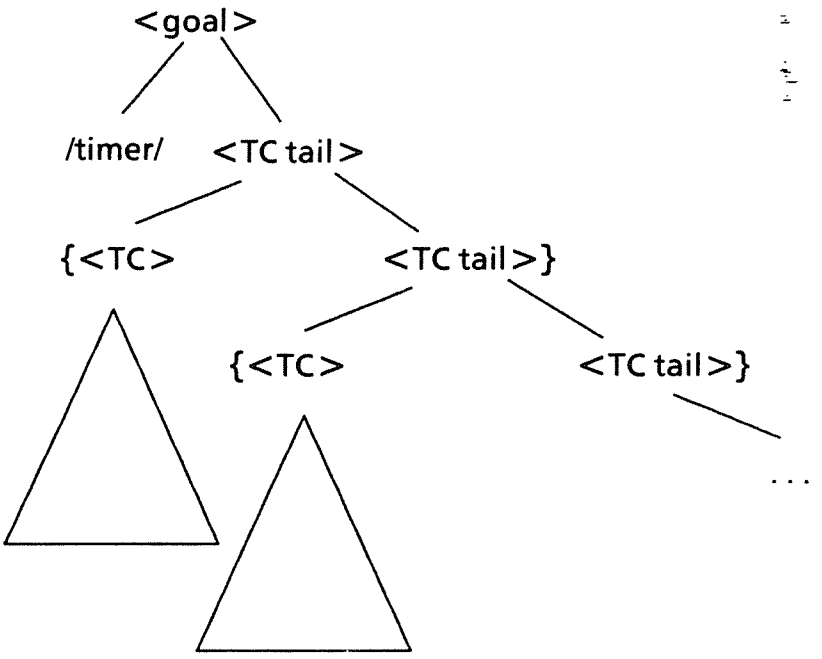

Figure 6: Multiple transport connections

The alternative productions of **<TC tail>** demonstrate a precaution that must be taken with recursive productions. In the absence of its second production, the **<TC tail>** process never blocks, and simply continues to spawn **<TC>** processes until memory is exhausted. **[U→FIN]** represents a command from the upper layer that no new connections are to be started. Even if this message never occurs (e.g., in UNIX there is no provision for stopping a protocol), the second production of **<TC tail>** provides an alternative production of **<TC tail>**, so that an instance of $P_{<TC\ tail>}$ blocks until it receives an event such as **[U→CR]** or **[N→CR]** which selects the first production.

Each connection (i.e. <TC> process) consists of three concurrent sub-protocols: connection establishment, transaction, and disconnection (see Figure 7). The attributes of <TC> are data, such as local and foreign addresses and reference numbers, common to more than one of these subprotocols.

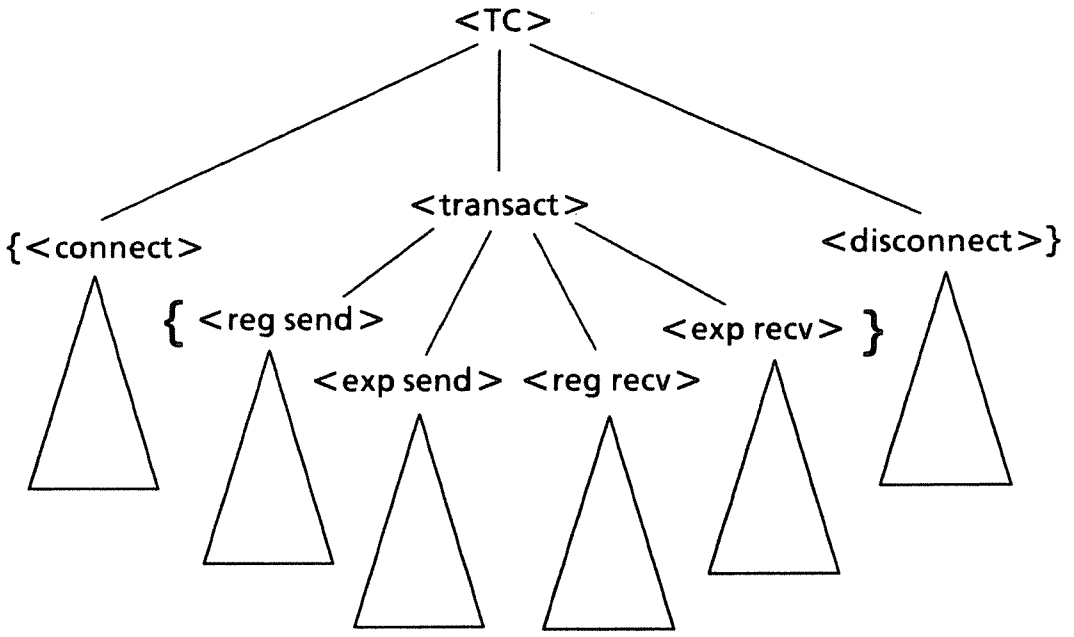

Figure 7: Subprotocols of a connection

5.5. Connection Establishment

Connection establishment involves an active and a passive end, which engage in a "three-way handshake" consisting of a CR, a CC and an AK TPDU. The active end is initiated by the receipt of a [U→CR] message, and

attempts to deliver a CR TPDU to the passive peer:

```
<connect>       :       <active open>.
                |       <passive open>.
;

<active open>   :       [U-CR] [N-CR] <retransmit CR>.
                (...)
                $3.count = RETRANS_COUNT

;

<retransmit CR>         :       [N-CC] [U-ACC] [N-AK].
                (...)

                | /timer/ [N-CR] <retransmit CR>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                (...)
                $3.count = $0.count - 1

                | /timer/ .
                if $0.count = = 0
                $1.interval = GIVEUP_TIME
                <TC>.connfailed = true

;
```

These productions show the general technique for timed retransmission of a TPDU: the initial sending is followed by a "retransmission symbol" (**<retransmit CR>**) whose attributes include a retransmission count. The retransmission symbol has three alternative productions: the arrival of a TPDU acknowledging the sent TPDU (in this case a CC TPDU), a timed production for retransmission, and a timed production (enabled when the retransmission count reaches zero) for stopping retransmission and waiting for an additional period before "giving up" and declaring the connection dead.

The concurrency of the connection and transaction subprotocols in the RHS of the **<TC>** production may be puzzling, since one thinks of connection establishment as strictly preceding data transfer. However, this concurrency is necessary: the passive end of a connection must be prepared to receive regular and expedited data after it sends a CC TPDU and while it is waiting for the completion of the three-way handshake, since the arrival of a valid data TPDU may take the place of a (possibly lost) AK TPDU as the third part of the handshake. On the other hand, some sequentiality must be enforced since data cannot be sent by either end until connection establishment is complete.

The necessary synchronization is enforced by the enabling conditions on the productions of **<transact>**, **<reg send>** and **<exp send>**, which depend on the *start_send* and *start_receive* attributes of **<transact>**. The passive open subprotocol enables the receive subprotocol (by setting **<transact>**.start_receive) as soon as it sends the CC TPDU, and enables the send subprotocol only when a valid AK or data TPDU has been received. The active open subprotocol enables both the send and receive subprotocols following receipt of the CC TPDU. This illustrates the way in which attributes and enabling conditions can be used to synchronize concurrent subprotocols.

5.6. The Transaction Subprotocol

The transaction subprotocol is further decomposed into four concurrent subprotocols: regular send and receive, and expedited send and receive.

5.6.1. Sending

For sending regular-priority data, TP-4 uses a sliding window whose size cannot exceed to the receive window size (or "credit") supplied by the peer. Acknowledgement TPDU's are used to 1) acknowledge new data, 2) report increases in the receive window size, and 3) provide a connection "heartbeat" allowing failures to be detected early.

The upper layer is assumed to manage a "send buffer" (a superset of the send window) such that a writer attempting to generate a **[U→DT]** message is blocked until there is room for the data. The **[U←AK]** message is used to inform the buffer manager of new space in the send buffer resulting from acknowledgement of data.

The regular send subprotocol is divided into subprotocols for sending data and for handling acknowledgements.

```
<reg send>    :       {<send msg tail> <recv acks>}.
              if <transact>.start_send
              $0.nextseq = 0
              $0.windowend = <transact>.initial_credit
              $1.ready = true
;

<recv acks>   :       /timer/.
              $1.interval = INACTIVITY_TIME
              <TC>.transerror = true

              |       [N→AK] <recv acks>.
              <reg send>.windowend = ($1.seqno + $1.credit) mod MAXSEQ
;
```

The send-data subprotocol is further divided into subprotocols for TSDU's (**<send msg>**) and each of these is further divided into subprotocols for each

packet within the TSDU (**<send packet>**).

The recursive production of **<send msg tail>** provides a queue for TSDU's which are in the send buffer but not necessarily in the send window.

```
<send msg tail> :       {<send msg> <send msg tail>}.
                $2.ready = false

                |       [U→GR] <transmit GR>.
;

<send msg>      :       [U→DT] <transmit msg>.
                $2.data = $1.data
;
```

Figure 8 shows the queueing and transmission of TSDU's.

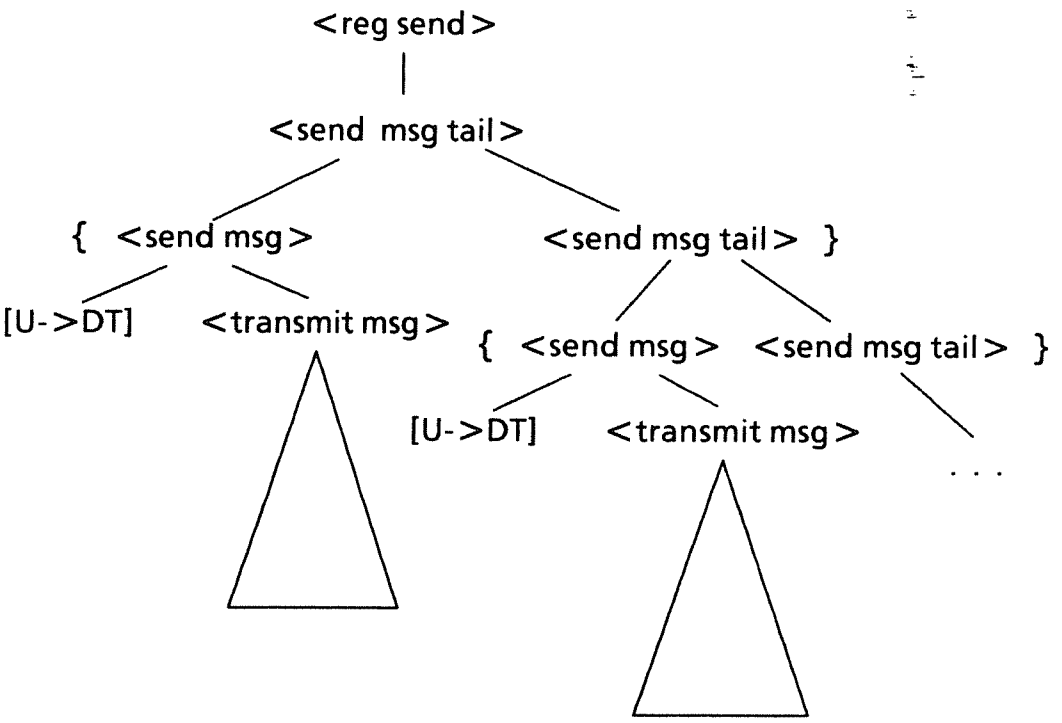

Figure 8: TSDU sending

The delivery of each TSDU involves a recursive process **<send packet tail>** which splits the TSDU's data into packets, and invokes a **<send packet>** process to handle the delivery of each one. The concurrency of these processes is needed to correctly implement the sliding window protocol.

The subprotocol for delivering a TSDU is defined below. It uses the following external functions: **#extract#**(d) removes a packet-sized substring from the head of the byte string referenced by *d*, and returns a reference to the removed data; **#eot#**(d) returns **true** if the given byte string is empty.

```
<transmit msg> :        <send packet tail>.
                if <send msg tail>.ready
                $1.eot = #eot#($0.data)
;

<send packet tail> :    {<send packet> <send packet tail>}.
                if not $0.eot
                $1.data = #extract#(<transmit msg>.data)
                $1.eot = #eot#(<transmit msg>.data)
                $1.seqno = <reg send>.nextseq
                <reg send>.nextseq = (<reg send>.nextseq + 1) mod MAXSEQ
                $2.eot = $1.eot

                |       /freedata/.
                if $0.eot
                $1.data = <transmit msg>.data
                <send msg tail>/<send msg tail>.ready = true
;
```

A boolean attribute *ready* of **<send msg tail>** is used to prevent a queued TSDU from beginning transmission until the last packet of the previous message has been sent. It is set by the last production of **<send packet tail>**.

The acknowledged delivery of a packet is handled by an instance of the **<send packet>** process, defined below. The following external functions are used: **#between1#**(x,y,z) returns **true** iff $x<y\leq z$ in cyclic order; **#copy#**(d) returns a new reference to the given byte string (this could involve a reference count mechanism rather than physically copying the data). Figure 9 shows the delivery of packets within a TSDU.

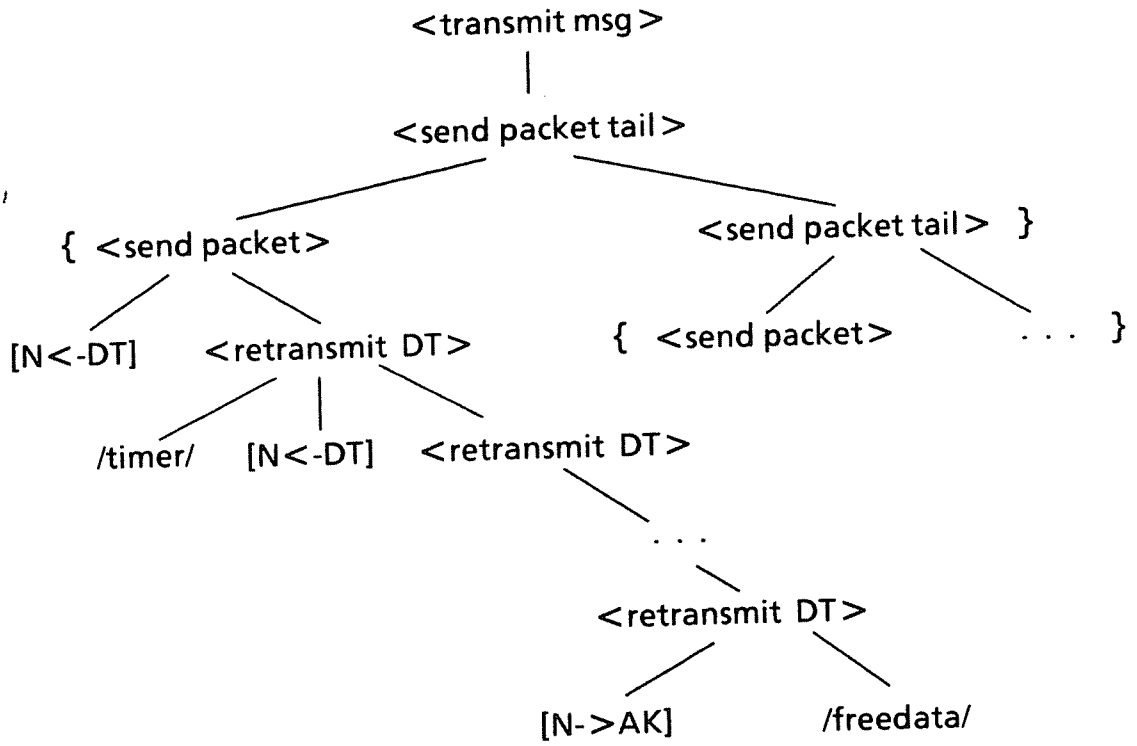

Figure 9: Packet sending

```
<send packet> :        [N-DT] <retransmit DT>.
                if (<reg send>.nextseq != <reg send>.windowend)
                        && (<transact>.nxoutstanding == 0)
                $1.src_ref = <TC>.refno
                $1.dst_ref = <TC>.foreign_refno
                $1.eot = $0.eot
                $1.seqno = $0.seqno
                $1.data = #copy#($0.data)
                $2.count = RETRANS_COUNT
;

<retransmit DT> :      /timer/ [N-DT] <retransmit DT>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                $2.src_ref = <TC>.refno
                $2.dst_ref = <TC>.foreign_refno
                $2.eot = <send packet>.eot
                $2.seqno = <send packet>.seqno
                $2.data = #copy#(<send packet>.data)
                $3.count = $0.count - 1

                |       [N-AK] [U-AK].
                if #between1#(<send packet>.seqno, $1.seqno, <reg send>.nextseq)
                $2.refno = <TC>.refno
                $2.data = <send packet>.data

                |       /timer/.
                if $0.count == 0
                $1.interval = GIVEUP_TIME
                <TC>.transerror = true
;
```

There is an enabling condition on the production of **<send packet>** which delays transmission of a data packet until it is in the send window and there are no outstanding expedited data packets (this correctly implement the semantics of expedited send: new regular data packets cannot be sent while there are outstanding expedited packets, but retransmissions are unaffected). The delivery of a packet ends when either it is acknowledged or when the give-up interval has elapsed after the last retransmission. In the latter case the *trans_error* attribute of **<TC>** is set to **true**, enabling a production of **<disconnect>** that ter-

minates the connection (see Section 5.7).

Expedited sending, which uses a stop-and-wait protocol and has its own sequence numbering, takes place in a separate subprotocol, **<exp send>**.

The sending portion of the grammar also handles the sending of a GR, since the GR TPDU has a sequence number in the regular data TPDU space and must be sent in sequence. Graceful close is discussed in Section 5.6.3.

5.6.2. Receiving

The regular receive subprotocol (**<reg recv>**) is interfaced to an upper-layer buffer manager with which it communicates via an input event **[U→AK]**, sent when there is new space in the buffer, and an external function *#bufslots#* which returns the number of receive window "slots" available in the buffer. The receive subprotocol is composed of two concurrent subprotocols: the receiving and transfer of data (**<recv packet tail>**) and the sending of acknowledgments (**<send acks>**):

```
<reg recv>      :       {<recv packet tail> <send acks>}.
                $0.recv_next = 0
                $0.window_end = #bufslots#(<TC>.refno)
                $1.data = empty
                $1.seqno = 0
;
```

The receive window is implemented as a dynamic set of concurrent processes, each of which waits to receive a packet with a particular sequence number (see Figure 10).

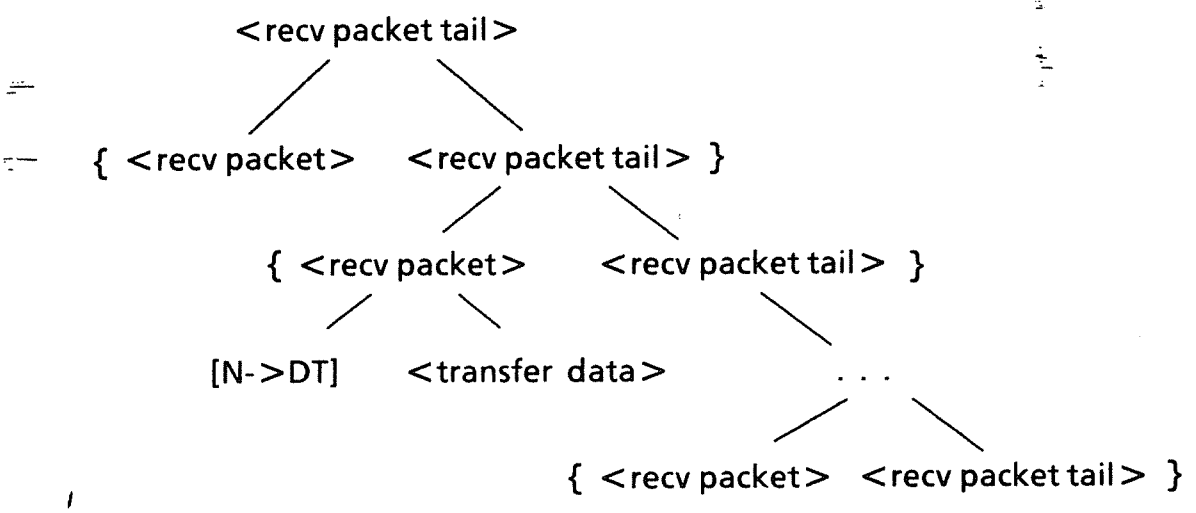

Figure 10: Receive window

This is realized by the following productions (*#between2#*(x,y,z) is **true** iff $x \leq y < z$ in cyclic order):

```
<recv packet tail> :    {<recv packet> <recv packet tail>}.
                        if #between2#(<reg recv>.recv_next, $0.seqno, <reg recv>.window_end)
                        $1.seqno = $0.seqno
                        $2.seqno = ($0.seqno + 1) mod MAXSEQ
;

<recv packet> : [N-DT] <transfer data>.
                        if $0.seqno = = $1.seqno
                        $2.data = $1.data
                        $2.eot = $1.eot
                        $2.seqno = $0.seqno

                        ...
;
```

The **<transfer data>** subprotocol is responsible for delivering the data to the

upper layer:

```
    <transfer data>:
                    if (not $0.eot) && ($0.seqno == <reg recv>.recv_next)
                    <reg recv>.recv_next = ($0.seqno + 1) mod MAXSEQ
                    <recv packet tail>/<recv packet tail>.data =
                            <recv packet tail>.data cat $0.data

                    |       [U-DT].
                    if $0.eot && ($0.seqno == <reg recv>.recv_next)
                    <reg recv>.recv_next = ($0.seqno + 1) mod MAXSEQ
                    $1.refno = <TC>.refno
                    $1.data = <recv packet tail>.data cat $0.data
                    <recv packet tail>/<recv packet tail>.data = empty
    ;
```

The process blocks until its packet is the first packet in the receive window. Then, if the packet's end-of-message flag (*eot*) is not set, the data is appended to the earlier portion of the message, and passed on to next process in the receive window. If the end-of-message flag is set, the packet is again appended, and the resulting string (which is a complete TSDU) is sent to the upper layer.

Because data packets can arrive out of order and do not include a TSDU number, it is impossible to write a specification in which each TSDU is received in a different subprotocol.

The **<send acks>** subprotocol is responsible for sending acknowledgements. In the acknowledgement strategy specified, AK TPDU's are sent when a data or graceful close TPDU is received, when a **[U-AK]** is received, and when no AK has been sent in a certain amount of time (this provides a guaranteed level of background subprotocol, the absence of which indicates a dead connection). The subprotocol is specified as follows:

```
<send acks>   :       /timer/ <send ack> <send acks>.
                      $1.interval = WINDOW_TIME

              |       [N-DT] <send ack> <send acks>.

              |       [N-GR] <send ack> <send acks>.

              |       [U-AK] <send ack> <send acks>.
                      <reg recv>.window_end =
                        (<reg recv>.recv_next + #bufslots#(<TC>.refno)) mod MAXSEQ
;

<send ack>    :       [N-AK].
                      $1.src_ref = <TC>.refno
                      $1.dst_ref = <TC>.foreign_refno
                      $1.seqno = <reg recv>.recv_next
                      $1.credit = #bufslots#(<TC>.refno)
;
```

5.6.3. Graceful Close

TP-4's "graceful close" mechanism allows connections to be closed by a three-way handshake, ensuring that no data is lost. The upper layer sends a "graceful close request" to the transport layer to indicate that it has no more data to send. This is sent to the peer entity as a GR TPDU, which is numbered in the regular data sequence space so that peer can know the sequence number of the last data packet. In the peer, when all data has been received (i.e., when the GR is first in the receive window), the upper layer is notified. After this point, as soon as the upper layer issues its own graceful close request (which it may already have done) the connection is closed (except for the "reference wait", see Section 5.7).

Sending and receiving of GR TPDU's is handled in the regular data send and receive subprotocols. Attributes *GRsent* and *GRarrived* of **<transact>**

signify respectively that a GR TPDU has been sent and acknowledged, and that a GR TPDU has arrived and is first in the receive window. When these are both **true**, the **<disconnect>** process takes over and terminates the connection (see Section 5.7).

5.7. Disconnection

The **<disconnect>** subprotocol of a connection handles messages from the upper or lower layers which abruptly close the connection, "cleans up" a connection which has failed in either the connection or transaction phases, and provides the "reference wait" (a period during which an old reference number cannot be reused) following a graceful close. The disconnect subprotocol is defined as follows:

```
<disconnect>  :       [U-DR] /remove/ /remove/ <deliver DR> <ref wait>.
              if <TC>.transerror || <TC>.connfailed
              $1.refno = <TC>.refno
              $2.where = <TC>/<transact>
              $3.where = <TC>/<connect>

              |       /remove/ /remove/ <ref wait>.
              if <TC>/<transact>.GRsent && <TC>/<transact>.GRarrived
              $2.where = <TC>/<transact>
              $3.where = <TC>/<connect>

              |       [U-DR] /remove/ /remove/ <deliver DR> <ref wait>.
              $2.where = <TC>/<transact>
              $3.where = <TC>/<connect>

              |       [N-DR] /remove/ /remove/ [U-DR] [N-DC] <ref wait>.
              $2.where = <TC>/<transact>
              $3.where = <TC>/<connect>
              $4.refno = <TC>.refno
              $5.src_ref = <TC>.refno
              $5.dst_ref = <TC>.foreign_refno

;
```

The *connfailed* and *transerror* attributes of **<TC>** are used by **<connect>** and **<transact>**, respectively, to communicate to **<disconnect>** that a fatal error has occurred.

<disconnect> uses **/remove/** to remove all processes in the connect and transact subprotocols. This guarantees that no loose ends will remain in the process tree after a connection is finished.

CHAPTER 6

DESIGN OF AN RTAG PARSER

6.1. The RTAG Parsing Task

The discussion of RTAG semantics in Section 4 mentions that an algorithm exists which supports those semantics, i.e. which given an RTAG specification will "interpret" it correctly within the limits of CPU speed. The main tasks of this algorithm are to provide multiplexed process execution, handle timers, and correctly select among alternative process definitions.

In this section we sketch an algorithm which performs these tasks reasonably efficiently, in the context of a conventional interrupt-driven sequential architecture. It seems easier to discuss the algorithm using parser-based terminology, perhaps because its connection to concrete data structures is clearer. For this reason, the implementation of the algorithm will be called an *RTAG parser*; it is a real-time program that, given an RTAG specification, processes input messages, handles timers, and generates output events in such a way that the protocol defined by the specification is obeyed. It does so by maintaining a "parse tree" of attributed symbol instances, and responding to input events by attempting to "left-derive" them by applying productions to leaf nonterminal symbols. It is essentially a top-down parser with backtracking.

An RTAG parser based on this design has been written in C under UNIX. Chapter 8 gives performance figures for a TP-4 implementation based on this parser, and discusses how the efficiency of the parser might be increased.

6.2. Restrictions Imposed by the RTAG Parser

The RTAG parser requires that input symbols and /**timer**/ occur only as the first symbol in a production RHS. This restriction simplifies the parser design. Since communication protocols using an unreliable lower level cannot rely on receiving a particular message, their RTAG specifications must always leave an alternative, meaning that input symbols must be leftmost. Therefore the restriction is a natural one in this setting.

To simplify the design of the parser, the semantics it provides differ in the following ways from those described in Chapter 4:

(1) Recall from Section 4.7 that an alternative process definition is *selected* if it can yield (without blocking) epsilon, an output symbol, or /**remove**/. The parser does not check each of the many possible production sequences to its end. Rather, if the first production in a sequence is enabled, it is applied irrevocably, regardless of whether productions later in the sequence are disabled.

(2) The parser maintains at most one timer per process.

(3) Process removal because of recursion (Section 4.12) is done only for right-recursive productions, rather than for arbitrary recursion.

For the RTAG specifications that we have written (including the TP-4 specification) these differences have no effect.

6.3. Definitions

A revised notion of "left-derivation" is needed in discussing selection of alternative process definitions. The intended semantics of the curly bracket notation (Section 4.4) are that the ordering of symbols within a concurrent group is irrelevant. We must therefore modify the notions of "leftmost" and "left-derivation". If a production RHS begins with a concurrent group, all the symbols in that group are *weakly leftmost* in the production; otherwise only the leftmost RHS symbol is weakly leftmost. A symbol <x> *immediately weakly left-derives* <y> if there is a production of the form

<x> : α.

in which <y> is weakly leftmost. *Weakly left-derives* is the transitive closure of "immediately weakly left-derives". The "weakly" prefix will henceforth be omitted.

The notions of "derives" and "left-derives" involve only the underlying CFG. If *X* is a nonterminal leaf in a particular parse tree and *Y* is another symbol instance or epsilon, *X yields Y* means that *Y* can be left-derived from *X* by a

sequence of productions which, when applied top-down from X and with attribute assignments performed, are all enabled.

6.4. Immediate Productions

Some productions are applied only in the course of deriving input events. The parser must distinguish these from productions that may have to be applied for reasons other than message arrival. A production P is *immediate* if either

(1) P is the only production of its parent symbol and the RHS starts with a nonterminal;

(2) P is part of a production sequence which left-derives epsilon, an output symbol, or **/remove/**;

(3) the RHS of P starts with **/timer/**.

6.5. Status of Symbol Instances

A symbol instance is *expanded* when a production has been applied to it, and *active* if it, or one of its descendants, is eligible for expansion. A symbol instance is initially inactive, and becomes active when all of its left siblings are finished (or, if part of a concurrent group, when all siblings to the left of the group are finished).

Symbol status can be mapped roughly to process status: activating a symbol corresponds to starting a process, expanding the symbol corresponds to selecting an alternative or proceeding after an initial blockage, and the symbol instance is

deactivated when the process finishes.

6.6. Static Data Structures

The parser uses the following static structures, which depend on the RTAG specification, and are computed in advance:

- **Symbol Descriptors**

Each symbol X is described by record giving its class (input, output, nonterminal, special) and the number of attributes.

If X is an input symbol, the descriptor also contains 1) a list of nonterminals that derive X in the underlying CFG; 2) a list of nonterminals that left-derive X (in the sense of Section 6.3); and 3) for each nonterminal Y which left-derives X, a list of the production sequences by which X can be left-derived from Y.

If X is an output symbol, the descriptor contains a pointer to the corresponding event performance routine.

If X is a nonterminal, the descriptor contains a list of its immediate productions.

- **Production Descriptors**

Each production in the RTAG specification is described by a record containing: 1) the enabling condition (in the current version, expressions are encoded in an intermediate form that is interpreted by the parser); 2) a list of

encoded attribute assignments; 3) pointers to the parent symbol's descriptor, and to those of each RHS symbol; and 4) information describing the concurrent grouping of the RHS symbols.

6.7. Dynamic Data Structures

The parser maintains the following dynamic (time-varying) data structures:

- **Symbol Instance Descriptors:**

Each symbol instance is described by a record containing 1) pointers to its parent and children instances; 2) the status (Section 6.4) of the symbol; 3) if expanded, a pointer to the production descriptor; 4) pointers to attribute instance descriptors for this symbol; and 5) a pointer (possibly null) to a timer descriptor. The parse tree is rooted by an instance (called *root*) of the goal symbol of the grammar.

- **Attribute Instance Descriptors:**

Each attribute instance is described by a record containing permanent and tentative values, and a set of *immediate links*, each of which points to an active symbol instance having an immediate production whose enabling condition depends on this attribute. Immediate links are also chained to the symbol instance.

- **Timer Descriptors**

Each timer is described by a record containing pointers to a symbol instance and to the descriptor of a production whose RHS starts with /timer/, and an integer delay. The parser maintains an incremental delay queue of timers descriptors.

The parser maintains the following queues, which can be thought of as "ready queues" of processes waiting to run:

(1) *Newly_active_queue*, a queue of symbol instances that have just become active and are awaiting processing. It is sorted by priority. A symbol is added to this queue when the left siblings of its concurrent group are finished.

(2) *Immed_queue*, a queue of active nonterminal instances that may be eligible for expansion by an immediate production. A symbol is added to this queue when an attribute value, on which the enabling condition of one of its immediate production depends, is changed.

Figure 11 illustrates the parser's data structures.

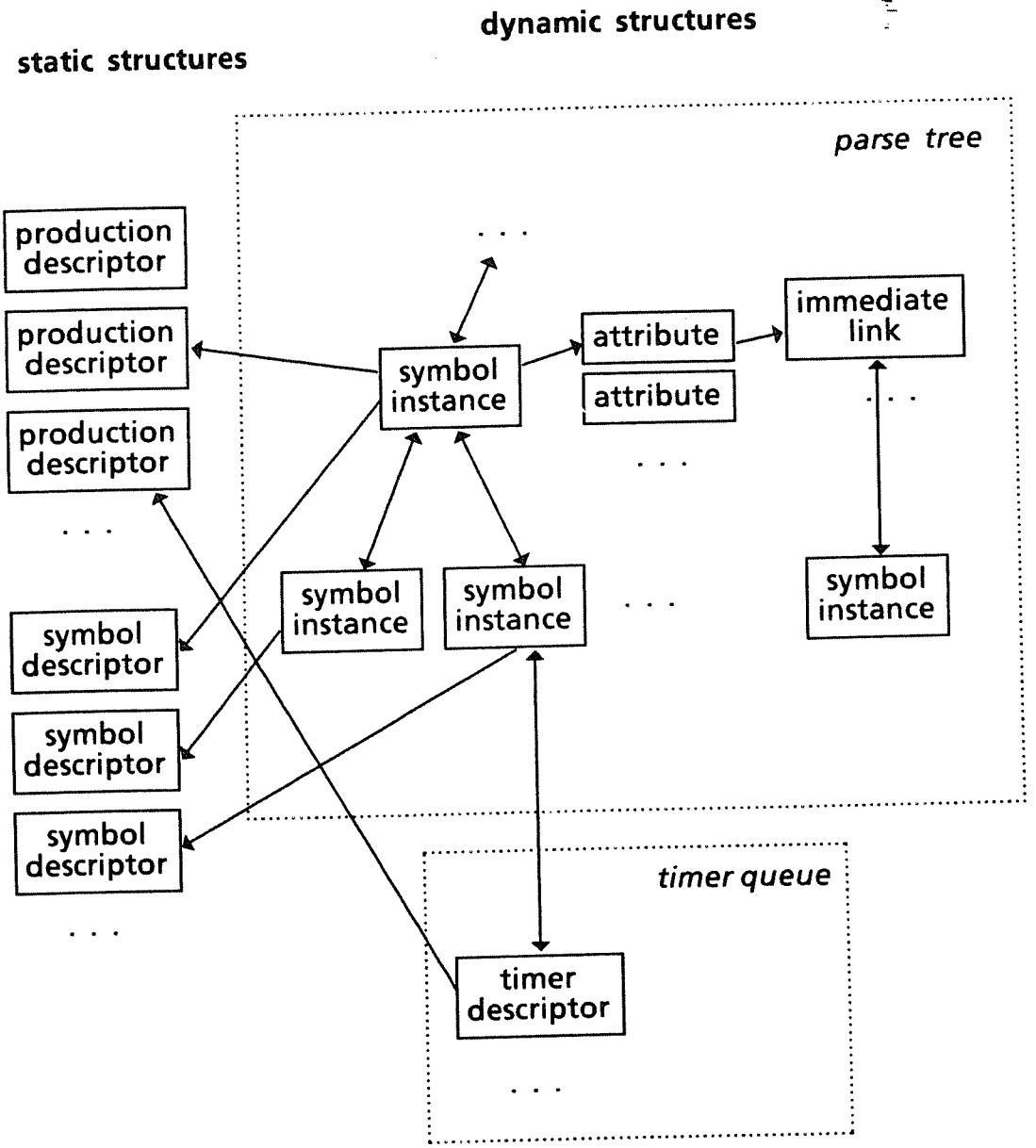

Figure 11: RTAG parser data structures

6.8. Algorithms

We present the parser design in roughly top-down order, starting with the algorithms for processing input events and timeouts, and recursively working down. The algorithms are expressed in C-like pseudocode.

The initialization of the parser consists of creating a goal symbol descriptor to serve as the parse tree root, and setting all queues to empty. Its actions thereafter are initiated only by input events and timeouts.

6.8.1. Processing Input Events

The algorithm for processing an input event is as follows:

```
/* process the message input_sym */

process_input(input_sym) {
   build_candidate_set(input_sym);
   for each element nonterm_sym of the candidate set {
      yield_input(nonterm_sym, input_sym);
   }
   do_consequences();
}
```

The steps are as follows:

(1) compute a "candidate" set of leaf nonterminals which might yield the input symbol;

(2) attempt to yield the input symbol from each candidate in turn, and

(3) perform the consequences of applying these productions (in the process model, this corresponds to executing newly-created or newly-unblocked processes).

In the context of a particular parse tree, the "candidate set" of an input symbol instance [x] is the set of nonterminal leaves that left-derive [x] in the underlying CFG, and that satisfy the "key attribute" restriction. It is computed by the following routine:

```
/* Find candidate set for input_sym */

build_candidate_set(input_sym) {
   candidate set = empty;
   if input_sym has the key attribute with value x {
      if some nonterminal P has the key attribute with value x
         search_subtree(input_sym, P, false);
      else
         search_subtree(input_sym, root, true);
   } else
      search_subtree(input_sym, root, false);
}

/* Do depth-first search for candidates in subtree below nonterm_sym.
  If check_key is true, prune subtrees having key attribute. */

search_subtree(input_sym, nonterm_sym, check_key) {
   if check_key {
      if nonterm_sym has different key attribute value than input_sym, return;
   }
   if nonterm_sym is expanded {
      for each nonterminal child X of nonterm_sym {
         if X is active and derives input_sym
            search_subtree(input_sym, X);
      }
   } else {
      if nonterm_sym left-derives input_sym
         add nonterm_sym to candidate set;
   }
}
```

Having computed the candidate set, the parser calls **yield_input** to attempt to yield the input symbol from each candidate.

The algorithm for **yield_input** (given below) refers to various other procedures. Of these, the following will be explained in more detail in later

sections: **prod_assigns** performs the attribute assignments of a production; **undo_changes** undoes tentative assignments; **make_permanent** makes tentative assignments permanent, and **finished** processes a newly finished symbol.

The remaining procedures called by **yield_input** are the following:

activate(sym): mark *sym* as active and append it to *newly_active_queue*. Repeat this for siblings of *sym* that are in its concurrent group.

prod_enabled(production, sym): return **true** iff *production* has no enabling condition, or has an enabling condition whose value, relative to *sym*, is **true**.

remove_immed_links(sym): remove the immediate links to *sym* from attributes of other symbols.

cancel_timer(sym): cancel the timer associated with *sym*, if any.

add_production(sym, production): create symbol instances as given by the RHS of *production*, and link them into the parse tree as the children of *sym*.

remove_ancestors(sym): remove the ancestors of *sym* from the parse tree.

We now return to **yield_input**, whose goal is to yield an input symbol [x] from a candidate **<a>**. Its logic is as follows: an outer for-loop ranges over the set of production sequences that left-derive **[x]** from **<a>** in the underlying CFG. These sequences are tested in turn until one of them succeeds or the set is exhausted. The task of testing a sequence is divided into cases depending on

whether the sequence has more than one production. If so, the productions, and their attribute assignments, are applied tentatively, and are undone if some production in the sequence is not enabled.

```
/* attempt to yield input_sym from nonterm_sym */

yield_input(input_sym, nonterm_sym) {

/* loop over possible production sequences */

   for each prod. seq. P1, . . . , Pn by which nonterm_sym left-derives input_sym {
      if n == 1 (i.e., the sequence is a single production) {

/* if single-production sequence, just check enabling condition */

         if prod_enabled(P1, nonterm_sym) {
            remove_immed_links(nonterm_sym);
            cancel_timer(nonterm_sym);
            add_production(nonterm_sym, P1);
            copy attribute values from input_sym to leftmost child of nonterm_sym;
            prod_assigns(nonterm_sym, P1, false);
            break;
         }
      } else {

/* if multiple-production sequence, tentatively apply it.
parent is next nonterminal to be expanded. */

         parent = nonterm_sym;
         for i = 1 to n {
            if i == n
               temporarily link input_sym as leftmost child of parent;
            if prod_enabled(Pi, parent) {

/* production succeeded -- apply it and keep going */

               add_production(parent, Pi);
               if i == n
                  copy attribute values from input_sym to leftmost child of parent;
               prod_assigns(parent, Pi, true);
            } else {

/* production sequence failed -- undo all changes */

               if nonterm_sym is expanded {
                  undo_changes();
                  remove_children(nonterm_sym);
```

```
                remove "expanded" flag from nonterm_sym;
                break;
            }
        }

    /* because of concurrent groups, next symbol to expand is not necessarily
    the first RHS symbol of the production just applied. */

        if i ≠ n
            parent = leftmost child of parent which is LHS symbol of P(i+1)
    }       /* for */

    /* entire production sequence succeeded -- make it permanent */

        remove_immed_links(nonterm_sym);
        make_permanent();
        cancel_timer(nonterm_sym);
        for i = 1 to n - 1 {
            S = leftmost child of production P(i);
            activate(S);
        }
        break;
    }       /* if */
}           /* for */

/* each production may have more than one leftmost symbol;
activate those that were not expanded */

    if nonterm is expanded {
        finished(input_sym);
        if sym is on immed_queue
            remove sym from immed_queue;
    }
}
```

6.8.2. Processing Timeouts

Each timer record points to 1) a production and 2) an active nonterminal symbol instance. A timeout is processed by applying the production to the symbol instance. As with input events, this can result in additions to *newly_active_queue* and *immed_queue*, which must then be processed.

```
/* apply timer production production to nonterminal sym */

process_timeout(production, sym) {
   remove_immed_links(sym);
   add_production(sym, production);
   prod_assigns(sym, production, false);
   finished(leftmost child of sym);
   do_consequences();
}
```

6.8.3. Attribute Assignments

Whenever an attribute value is changed, it is possible that that the enabling condition of some immediate production is changed from **false** to **true**. This is handled by adding to *immed_queue* all relevant symbol instances. Hence the following is called each time an attribute value is changed permanently:

```
/* see if changing attribute triggers immediate productions */

add_to_immed(attribute) {
   for each immediate link L of attribute {
      symbol = L.symbol;
      if symbol is not in immed_queue
         add symbol to immed_queue;
   }
}
```

When a production is applied as part of a sequence (in **yield_input**), its attribute assignments must be performed since later productions in the sequence may use the target attributes in their enabling conditions. On the other hand, if the sequence fails then all its assignments must be undone. Hence while multi-production sequences are being tested, their attribute assignments are "logged" (i.e. modified attributes are put in a "change log" along with their old values). If the sequence fails, the changes are undone.

The following routine performs the attribute assignments for a particular production in which *nonterm* is the parent instance. The *tentative* parameter is **true** if the changes are to be logged; otherwise the changes are permanent. Each permanent change may enable new immediate productions, which are added to the immediate queue by **add_to_immed**.

```
prod_assigns(nonterm, production, tentative) {
   for each attribute assignment S of production {
      find target attribute A (LHS of S);
      compute value X of RHS expression of S;
      if tentative {
         if A is not in change_log {
            A.oldvalue = A.value;
            add A to change_log;
         }
      } else
         add_to_immed(A);
      A.value = X;
   }
}
```

When a production sequence fails, the attribute assignments it generated must be undone:

```
undo_changes() {
   for each element A of change_log
      A.value = A.oldvalue;
   change_log = empty;
}
```

If a production sequence succeeds, the following routine makes the attribute changes permanent by purging the change log, and handles their consequences by adding to *immed_queue* those symbol instances which may have an immediate production enabled by the attribute changes.

```
make_permanent() {
   for each element A of change_log
      add_to_immed(A);
   change_log = empty;
}
```

6.8.4. Processing Immediate Productions and Newly Active Symbols

The immediate and the newly active queues are processed after handling an input event or timeout. This processing, which may add new entries to the queues, is continued until both queues are empty.

```
do_consequences() {
   while (newly_active_queue ≠ empty) or (immed_queue ≠ empty) {
      if newly_active_queue ≠ empty {
         sym = head of newly_active_queue;
         process_newly_active(sym);
      }
      if immed_queue ≠ empty {
         sym = head of immed_queue;
         do_immediate(sym);
      }
   }
}
```

An element on the immediate queue is processed by testing the enabling conditions of its immediate productions. If a condition is satisfied for a production whose RHS starts with **/timer/**, a timer is started; if a condition for a different type of production is satisfied, the production is applied. **start_timer**(interval, P, sym) is a procedure which starts a timer for the given time interval, linking it to the given nonterminal symbol instance and production.

```
do_immediate(sym) {
   for each immediate production P of sym {
      if prod_enabled(P, sym) {
         if first symbol of P's RHS is /timer/ {
            evaluate x = /timer/.interval;
            start_timer(x, P, sym);
         } else {
            remove_immed_links(sym);
            add_production(sym, P);
            prod_assigns(sym, P, false);
            cancel_timer(sym);
            if RHS of P is empty
               finished(sym);
            else
               activate(leftmost child of sym);
            break;
         }
      }
   }
}
```

Special actions are taken when a symbol *X* is first activated:

(1) If *X* is an output symbol then the corresponding external function is called; the attribute values of *X* are passed as arguments.

(2) If *X* is **/remove/**, the appropriate action is taken. **/remove/** removes the symbols and associated storage below the symbol instance *Y* pointed to by **/remove/**.where; *Y* is left in the tree but marked as finished.

(3) If *X* is a nonterminal, **do_immediate** is called to start a timer or apply an immediate production, as appropriate. If no production was applied, the set of attributes on which the conditions depend is found, and immediate links from these attributes to *X* are established, so that if one of the attributes is later changed the conditions can be retested. **get_attr_list**(sym) returns a list of all attributes instances that are operands of enabling expressions of

immediate productions of *sym*.

The processing for a newly activated symbol instance *sym* is as follows:

```
process_newly_active(sym) {
   if sym is an output symbol {
      call the event routine for sym, passing sym's attributes as parameters;
      finished(sym);
   } else if sym is a nonterminal {
      do_immediate(sym);
      if sym is expanded, return;
      list = get_attr_list(sym);
      for each attribute A in list
         add an entry, pointing to sym, to A's immediate-link set;
   } else if sym is /remove/ {
      remove_children(sym.where);
      finished(sym.where);
      finished(sym);
   }
}
```

6.8.5. Processing of Finished Symbols

When a symbol becomes finished, symbols may have to be removed from the tree as described in Section 4.11. This is done as follows:

```
/* process a newly finished symbol */

finished(sym) {
   if sym is the root
      exit;                    /* protocol is finished */
   mark sym as finished;
   parent = sym.parent;
   if sym is a nonterminal
      free children of sym;

/* recurse if necessary */

   if all siblings of sym are finished {
      finished(parent);
      return;
   }

/* remove symbols from recursive constructs */

   if all siblings of parent are finished , and sym and parent are the same symbol {
      grandparent = parent.parent;
      free parent and its siblings;
      relink children of parent as children of grandparent;
      sym2 = right sibling of sym;
      if sym2 is not active
         activate(sym2);
      return;
   }

/* activate right sibling if necessary */

   if all symbols in sym's concurrent group are finished,
     and this group has a right sibling sym2
      activate(sym2);
}
```

CHAPTER 7

RTAG SOFTWARE ENVIRONMENTS

In the section we describe an RTAG-based software system for protocol development, debugging, and implementation. This software has been written in C under 4.2 BSD UNIX. The two central components are an RTAG parser based on the algorithm described in Chapter 6, and an "RTAG compiler", an off-line program which converts an RTAG specification into the form required by the RTAG parser.

In using RTAG in a heterogeneous network environment, the RTAG compiler will generally be needed only on a "host" system, while versions of the RTAG parser run on the "target" systems on which the protocol must run. The relationship between the components of the RTAG software system is shown in Figure 12.

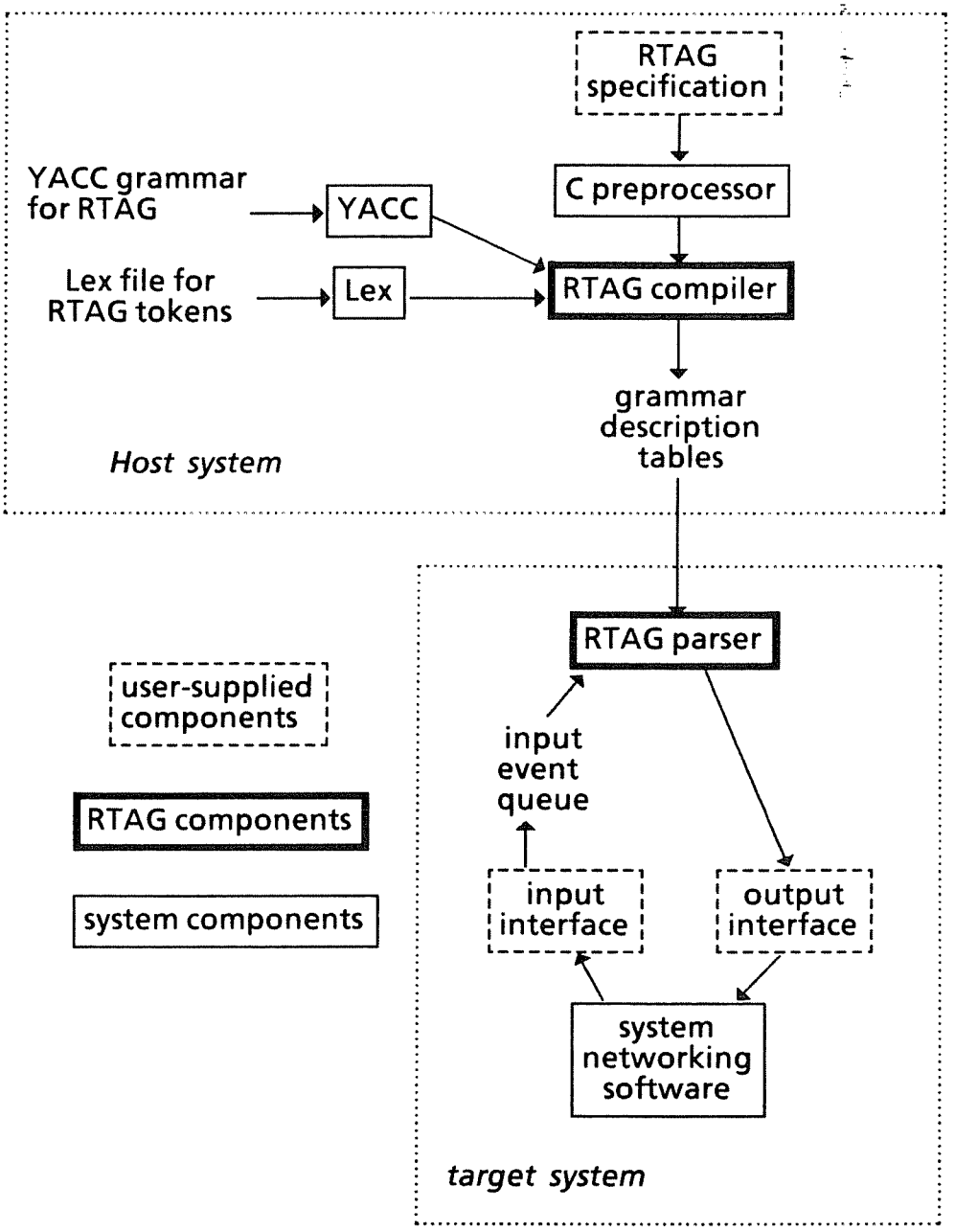

Figure 12: The RTAG software system

The primary goal of the implementation work to date has been to produce protocol implementations in the UNIX kernel. Tools have also been developed to assist in user-level debugging of RTAG specifications and interface routines. Analogous systems could be built in the context of other operating systems and languages.

7.1. The RTAG Compiler

The RTAG compiler converts an RTAG grammar from its symbolic form into a form that encodes the static data structures of the RTAG parser (see Section 6.6). The output of the RTAG compiler is a file of integers that can be read by the initialization phase of an RTAG parser and converted into more efficiently accessible structures. This is a "portable" format in that it can be directly used by RTAG parsers on all target systems.

The RTAG compiler can optionally convert these integer files into C source files containing initializer declarations for the parser's static data structures. This file can be compiled and linked with the parser. This format is used by the UNIX kernel implementations because it eliminates the initialization phase.

The RTAG parser and the user-supplied interface routines must have a common way of referring to symbols. For this purpose, the RTAG compiler assigns numbers to input symbols, and produces a file of these assignments. For example, if the grammar contains input symbols **[N→DT]** and **[U→DT]**, this file might contain

```
#define NDT_in 0
#define UDT_in 1
```

Note that the symbol names are converted to legal C identifiers. The file is intended to be #included in the interface routines which generate input events (Section 7.3.2)

The RTAG compiler must also provide a means for associating procedures with output symbols. This is done by producing an "include" file containing declarations for the output event routines and external functions which the user must supply, as well as an address table through which these routines can be called by the RTAG parser. For example, if the RTAG specification contains output symbol **[U-DT]** and external function **#eot#**, this file would be

```
extern int UDT_out(), eot();
extern int (*externtable[])() = {UDT_out, eot};
```

The RTAG compiler was constructed using the Lex scanner generator [Les79] and the Yacc parser generator [Joh79]; this facilitates changing or extending RTAG. The RTAG compiler also uses the C pre-processor [UNI83] to handle comments, macro substitutions, and include files.

7.2. An RTAG Parser Under 4.2 BSD UNIX

The parsing algorithm described in Section 6 has been implemented in C under UNIX. This section discusses the software environments and debugging aids that have been developed for the parser, and gives some system-dependent internal details.

7.2.1. Software Environments for the RTAG Parser

The following execution environments are available for the RTAG parser:

- Kernel Mode

The 4.2 BSD UNIX kernel is designed to facilitate experimentation with communication protocols [UNI83]. All protocols are interfaced to user-level processes by the "socket system", which provides services such as process synchronization, data buffer management, and queueing of pending connections. 4.2 BSD UNIX supports the coexistence of multiple "protocol families", and the interfaces between 1) protocols and the socket system, 2) protocol layers within a family, and 3) protocols and network interfaces drivers, are standardized to some extent. The position of the RTAG parser in this setting is shown in Figure 13.

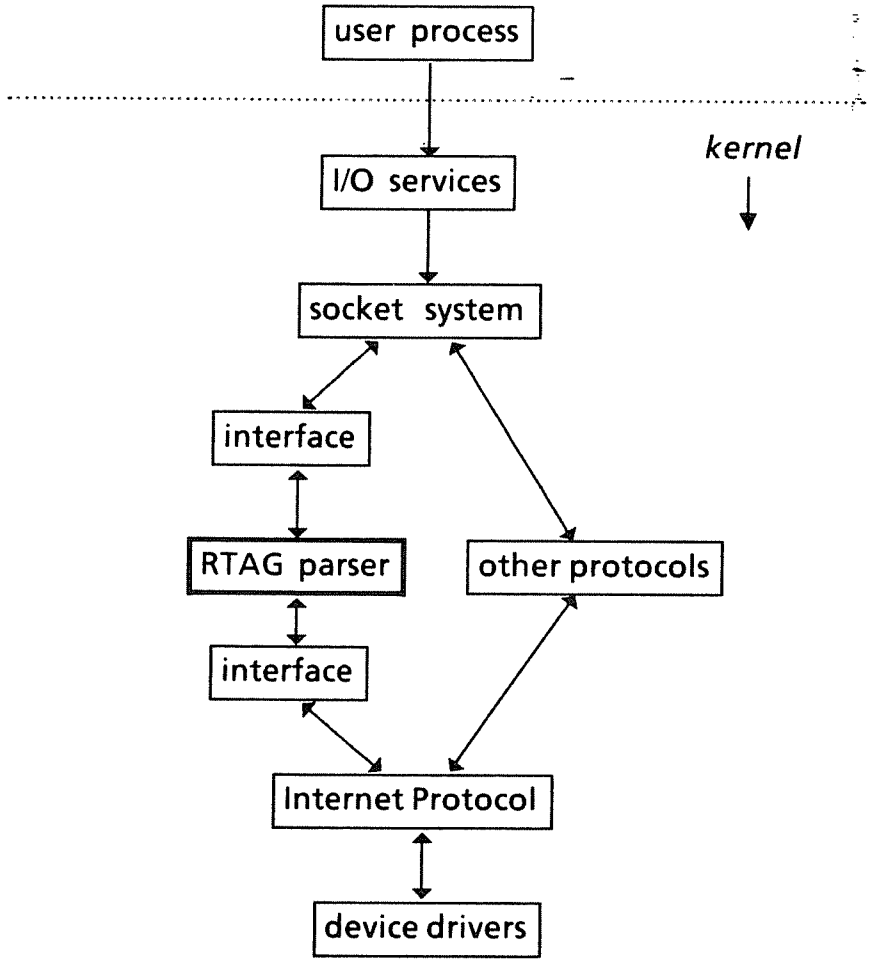

Figure 13: The UNIX kernel environment

- User-level Kernel Simulation

In debugging kernel-level protocols it has been extremely helpful to simulate the kernel at the user level. This was done by compiling the relevant parts of the kernel (such as the socket routines and the read/write system call routines) into

the user program, and using a software simulation of the lower layers (IP and the network interfaces). Also available in this mode is a routine for interactive perusal of the parse tree, and debugging options that allow logging the actions of the parser (productions, event occurrence, and attribute assignments) in a disk file. See Figure 14.

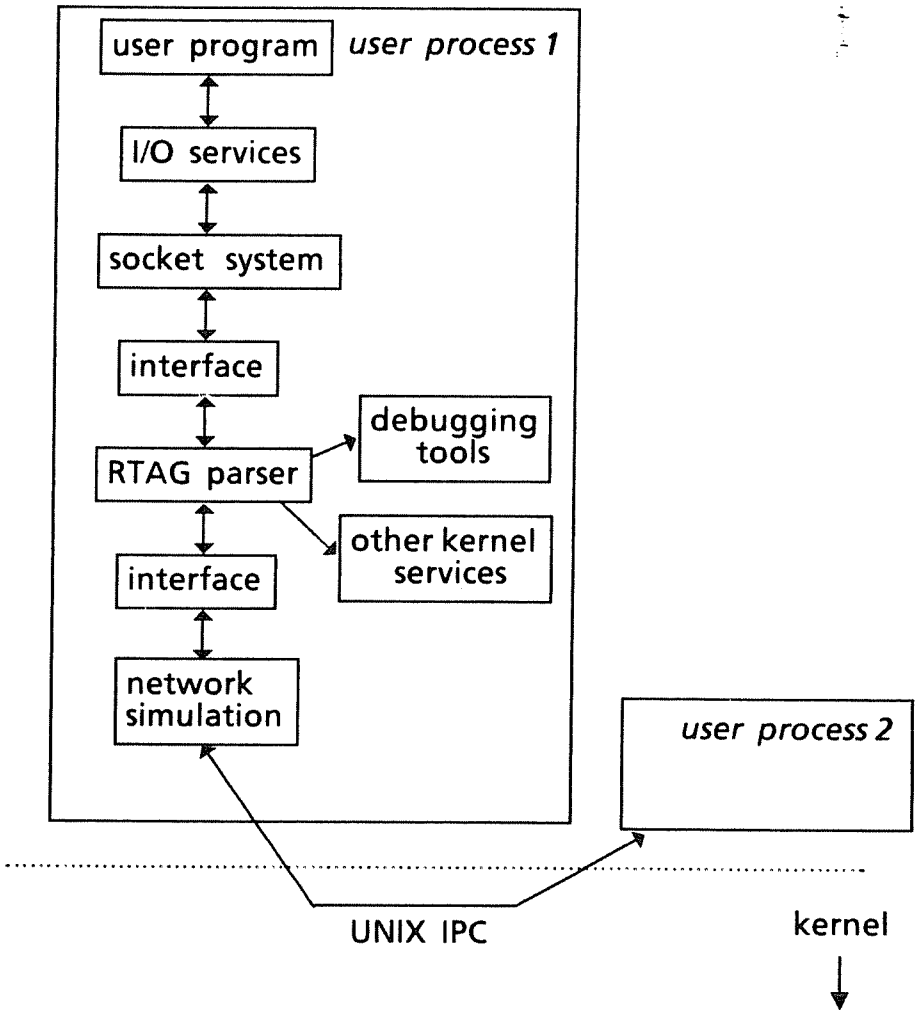

Figure 14: The UNIX kernel simulation environment

- User-level Experimentation System

This system allows RTAG specifications to be tested with a minimum of programming. It consists of a simulated network layer based on the UNIX IPC facility, as well as a routines that translate between event symbols and packets

(correctly handling data pointed to by *dataptr* attributes). The tree perusal and logging routines are available here also.

7.2.2. Memory Management in the RTAG Parser

The networking portion of the 4.2 BSD UNIX kernel includes a memory management system which stores variable-size byte strings using chains of "mbuf" records. Utility routines are available to manipulate mbuf chains in various ways. In the UNIX version of RTAG, the *dataptr* attribute type is a pointer to an mbuf chain; the RTAG parser and external functions perform *dataptr* operations by calling kernel mbuf utility routines.

A second memory management task is allocation of the various descriptors used by the RTAG parser (symbols instances, attributes, timers, and immediate links). The parser allocates and frees these descriptors at a high rate, so a general-purpose memory allocation scheme (like *malloc* and *free* on UNIX) may impose a prohibitive performance penalty. The mbuf system could be used, but this would also be inefficient, particularly in terms of memory. Instead, a dedicated allocation system is used: pools of free records of each type are maintained, so allocation and deallocation are trivial and fast. The RTAG parser demands that records for the RHS symbols of each production be contiguous, so the allocator maintains separate pools of blocks of symbol records, in sizes ranging up to the maximum RHS length. A similar system is used for attributes, which are also allocated in blocks.

7.2.3. Entry Points

The entry points to the RTAG parser are the following:

init_parser() must be called first; it initializes all dynamic data structures.

gen_event(symnum, x1, x2, ...) adds an input event with the given symbol number and attributes to the input queue. *Symnum* is a name defined in the "include" file generated by the RTAG compiler (Section 7.1).

/clock() must be called by the clock interrupt routine; it counts a tick off of all timers and processes any resulting timeouts.

User-supplied output routines and external functions must be linked with the parser and with the address table generated by the RTAG compiler. The parser calls them indirectly through this table.

7.3. Interface Routines for TP-4 under 4.2 BSD UNIX

The RTAG specification for the TP-4 transport protocol described in Chapter 5 has been combined with the kernel-mode RTAG parser to obtain a production version of TP-4 operating in the Internet domain. The following interface routines and external functions were needed:

- Lower-layer event output routines for assembling TPDU's and sending them via IP.
- A lower-layer input routine to accept a TPDU from IP, verify the checksum, extract the TPDU parameters, and generate the appropriate input

event.

- Upper-layer output event routines for conveying information to the UNIX socket system. For example, the data output routine simply appends the mbuf chain pointed to by its argument to the receive buffer of the appropriate socket.

- An upper-layer input routine to handle requests from the socket system. These, for the most part, translate directly into input events.

7.4. Summary of RTAG Protocol Implementation Procedure

We now summarize the steps involved in generating an RTAG-based UNIX kernel protocol implementation:

(1) Based on the nature of the adjacent protocol entities and the services supplied/required by the protocol, choose an event set for the RTAG specification.

(2) Develop an RTAG specification of the protocol.

(3) Write input interface routines which handle messages from adjacent entities, and clock interrupts, by calling the RTAG parser entry points described in Section 7.3.2.

(4) Write output interface routines (output event performance routines) which send messages to adjacent entities.

(5) Compile the RTAG specification with the RTAG compiler, producing C initialization files.

(6) Compile and link the C initialization files, the RTAG parser, the interface routines, and the kernel simulation code.

(7) Debug the protocol implementation in user-mode kernel simulation.

(8) Generate a UNIX kernel which includes the C initialization files, the RTAG parser, and the interface routines.

(9) Debug the protocol implementation in kernel mode.

CHAPTER 8

EFFICIENCY OF RTAG IMPLEMENTATIONS

Time and space efficiency are critical in protocol implementations [Cla83]. Time efficiency is necessary for high data transfer rates, to avoid interference with the timing of the protocol, and to avoid degradation of the host system performance. Space efficiency (code size and buffer use) is important since space is often at a premium in operating system kernels.

The viability of RTAG as a means for producing production-quality implementations thus depends on the time and space efficiency of the RTAG parser. In this section we report various performance measurements of the RTAG-based TP-4 implementation under UNIX, and compare these measurements with those of a conventional (hand-coded) implementation of the same protocol, written by Allan Bricker and Tad Lebeck at the University of Wisconsin - Madison. We then suggest several ways in which the efficiency of the RTAG parser could be improved.

8.1. Efficiency of the RTAG TP-4 Implementation

The memory usage of the RTAG TP-4 implementation under UNIX is broken down as follows:

RTAG parser object code size:	9640 bytes
interface routines object code size:	7968 bytes
static data structures:	42728 bytes
total:	60336 bytes

The conventional implementation has an object code size of 34412 bytes. Both implementations use a small (1-2 Kbytes) additional amount of memory per connection. Thus the RTAG version is somewhat larger, mostly due to the static data structures. Use of more space-efficient representations could almost certainly eliminate this difference. RTAG implementations would then be feasible in memory-restricted environments such as personal computers.

Throughput measurements were made as follows: two user-level processes running on different unloaded hosts (VAX 11/750's) establish a TP-4 connection. One process reads the time-of-day clock, sends a large amount (> 1 Mbyte) of data, then reads the clock again and reports the elapsed time. The receiving process works analogously. Neither process does any disk I/O; the data sent is meaningless. A TPDU size of 1024 bytes was used, and the data was checksummed at both ends.

When the RTAG version sends and the conventional version receives, throughput varies between 20 and 22 Kbytes/second. When the roles are reversed, throughput drops to about 15 Kbytes/second. In both cases the CPU utilization (as indicated by the load average) is between .3 and .5. When the conventional version is used on both ends, throughput ranges from 40 to 45 Kbytes/second, and CPU utilization is about .07.

These figures suggest that the RTAG version is about 10 to 15 times slower than the conventional version in terms of CPU time. The throughput rate depends on other factors (such as network delay) which are identical in the two versions. Therefore the discrepancy in throughput will be smaller, and will decrease as the packet size is increased.

The current RTAG TP-4 implementation would not provide acceptable performance on a moderately-loaded multi-user system. However, with reasonable timer values, its speed on an unloaded system doesn't interfere with the protocol functionality (e.g. by causing spurious retransmissions). Hence it could be useable for implementations which are experimental or run in a front-end processor.

8.2. Increasing the Performance of the RTAG Parser

Simplicity, rather than efficiency, was the major factor in the design of the RTAG parser, and there are many ways in which its efficiency can be improved. Some of these are "portable" in the sense that they don't affect the RTAG compiler or the format of the grammar description file. Others require that the RTAG compiler depend somewhat on the language of the target machine.

8.2.1. Candidate Set Computation

Consider the task of computing the candidate set for an input event; the set is a function of the type of the input symbol and the values of its key attribute, if

any. In the current implementation, this set is computed, on message arrival, by a top-down search of the parse tree. Even if all possible pruning is done, this may use time proportional to the size of the tree.

A possibly better approach is to compute candidate sets in advance, and store them in an easily-accessible form. For example, one could consider *(symbol name, key attribute value)* pairs as indices, and build a hash table whose entries are candidate sets for particular indices. When a nonterminal symbol is activated it must be added to the appropriate sets, and when it is later expanded it must be removed. This would have a constant average lookup time, although the overhead of maintaining the hash table might be significant.

8.2.2. Evaluation of Attribute References

In the current implementation, attribute references are interpreted. A local attribute reference, for example, is described by a record containing the offset of the symbol in the production, and the offset of the attribute in that symbol's attribute set. Locating the attribute involves a moderate amount of work.

Speeding up attribute reference evaluation requires a suitable organization of symbol and attribute records in memory. The current implementation has contiguous symbol records for a RHS, and contiguous attribute records for a symbol, but no fixed relation between a symbol and its attributes. A better organization has the symbol records for a RHS followed immediately by records for all attributes of those symbols. This reduces the time spent allocating

memory and also provides constant offsets of attributes relative to their symbols, and to the start of the aggregate RHS record.

It would also be possible to calculate the addresses of non-local symbol references at the time of production application, and put these at fixed locations in the aggregate RHS record. Combined with the above, this could reduce attribute references to a few machine instructions.

8.2.3. Expression Evaluation

Expression evaluation is also interpretive in the current implementation; the RTAG compiler produces parse trees for expressions, and the RTAG parser interprets these trees. An alternative would have enabling conditions and attribute assignments be converted into function definitions in the language of the target machine. This could be done by the RTAG compiler using a per-target-machine expression translator. The approach would be particularly efficient in combination with the attribute reference mechanism described above; each function would take as arguments pointers to the parent's attribute block and the aggregate RHS record.

For example, suppose the target language is C, and the grammar contains

```
<x>   :       <y> <z>.
      if $0.attr == <foo>.attr
      $1.number = <blah>.seq + 1
;
```

The RTAG compiler would produce something like

```
func1(par_attr,rhs_block)
int *par_attr, *rhs_block;
{
        return(par_attr[3] == *(rhs_block[47]));
}
```

for the enabling condition and

```
func2(rhs_block)
int *rhs_block;
{
        rhs_block[22] = *(rhs_block[59])+1;
}
```

for the attribute assignment. These functions would then be compiled and linked with the RTAG parser; production descriptors would contain pointers to the code for their enabling conditions and attribute assignments.

This approach would make expression evaluation roughly as efficient as in hand-coded implementations. It has the drawback that a portion of the RTAG compiler must be modified to accommodate different target languages. On the other hand, the relevant part of the target language (expressions and functions calls with only simple types) is small and is similar among many HLL's.

8.3. Finite-State Constructs

The parser could treat certain grammar constructs as special cases, and bypass the normal mechanisms. This would be particularly useful for RTAG constructs which are equivalent to FSA's. For example, the net effect of a production like:

```
<foo> : [N→AK] [U←AK] <foo>.
;
```

could be achieved without modifying the tree.

8.4. Performance Summary

The current RTAG software system can produce protocols which, though less efficient than their hand-coded counterparts, are usably efficient. There are means by which the efficiency of RTAG implementations can be increased, possibly to the point of rivaling hand-coded implementations. These may tend to increase the complexity of the RTAG parser and the RTAG compiler, and to decrease the portability of the latter. Increasing RTAG efficiency is an area for future study.

CHAPTER 9

COMPARISONS TO RELATED WORK

In this section we contrast RTAG with conventional attribute grammars, and compare it to other protocol specification formalisms.

9.1. Context-Free Grammars

The use of CFG programming language specification as a basis for compiler construction has been a great success, and RTAG is in some sense an attempt to translate this approach to the realm of protocols. There are, however, two fundamental differences between RTAG and CFG which make it unlikely that CFG parsing techniques can be applied directly to RTAG:

(1) The presence of multiple concurrent processes means that a single CFG parser instance is not sufficient.

(2) There is no notion of temporal ordering in language parsers; for example, consider a production of the form

<x> : [input] [output] [input].

In order to maintain sequential order, the RTAG parser must perform the output event immediately after the first input event. A language parser, however, can wait until the second input event occurs (and, in fact, may need to do so if the grammar is not LL(1)) before performing the output

event (i.e., action routine).

9.2. Attribute Grammars

Attribute grammars [Knu68] are an extension of CFG's in which symbols have associated data attributes. Each production has an associated set of *semantic functions* which express functional dependencies among the attributes of the symbols in the productions. The notion of derivation is defined in terms of a parse tree in which all attributes have values and the dependencies are obeyed. Attributes are classified as *inherited* or *synthesized*. *Affix grammars* [Kos71] are a related formalism in which attribute dependencies are expressed by *primitive predicates*, special terminal symbols which can fail or succeed based on their inherited attributes, and which in the latter case provide values for their synthesized attributes.

Most applications of attribute grammars have involved programming languages, for formally expressing the semantics of programming language [Knu71], or for expressing context-sensitive syntactic properties such as type consistency. The latter application has been especially useful for language-based editors ([Joh83], [Dem81]). Recently Fischer and Ganapathi used attribute grammars as a specification language for CPU architectures, and used this as a basis for constructing retargetable code generators [Gan82].

Many algorithms have been developed for evaluation of attribute and affix grammars ([Ken76], [Saa78], [Boc76], [Wat77]). These generally require that

the attribute grammar be free from circular attribute dependencies, and often additional properties that allow one-pass evaluation.

Some of the existing results for attribute grammars may be applicable to RTAG, but many of them are rendered inapplicable by the following differences between attribute grammars and RTAG:

(1) The semantics of attribute grammars are defined in terms of a single parse tree in which attributes have fixed values. Some attribute evaluation algorithms may involve changing attribute values, either as part of the attribute evaluation algorithm or because of an edit of the parse tree, but the goal is still to find fixed final values. In RTAG, on the other hand, the change of values of an attribute over time is an essential part of the semantics. Attributes are properly thought of as "variables" which can be used for temporary storage of information.

(2) In most language-related uses, the entire parse tree (in the underlying CFG) is generated before attribute evaluation, and enabling conditions (or their equivalent) are used for after-the-fact error checking rather than being incorporated into the parsing process as a means of selecting a production from among several alternatives.

Johnson [Joh83] describes a system for expressing and evaluating non-local attribute references that is different from ours. His system uses "non-local productions" which are matched, whenever possible, with symbol instances created

by conventional (local) productions. The attribute evaluation rules associated with non-local productions can cause the flow of attribute information between widely separated symbols in the tree. Johnson's use of attribute grammars involves language-based editors, in which an edit (such as changing the type of a variable) necessitates the flow of updated attribute information to the places in the tree where the variable is referenced. He presents algorithms for updating the parse tree when a subtree replacement is done. The primary connection between Johnson's work and RTAG is that in both cases attribute instances are dynamically linked to expressions that refer to them, so that the consequences of modifying an attribute value can be done efficiently.

Skedzeleski [Ske78] defines "time-varying attribute grammars" as a means for non-procedural description of iterative algorithms, and uses this to specify code generation and optimization techniques. The semantics of attribute assignments in RTAG are similar to those used by Skedzeleski; in both cases assignments are thought of as operations that must be performed (possibly several times) rather than as conditions that must be satisfied. Like us, Skedzeleski uses the idea of linking attributes to assignments that refer to them. One difference is that Skedzeleski attaches enabling conditions to individual attribute assignments so that circular data dependencies are not necessarily meaningless.

9.3. Comparison with Other Implementation Systems

The AFSM system reported by Blumer and Tenney and IBM's FAPL system (see Section 2.4.1) are representative of current work in automated implementation of data communications protocols. As described earlier, these have a similar structural basis: the major control flow for each protocol entity is provided by a finite state machine whose transitions are triggered by input events or timeouts, the processing done at each transition (which may generate output events or schedule timers) is written in a high level language, and there are global variables and data structures that are manipulated by this processing. FAPL also has scheduling features for handling multiple protocol layers within a single specification. In both systems, a program converts the information contained in the FSA into a driver program or a set of tables to be used by a driver program. This, interspersed with calls to interface routines and to the HLL code, comprises an implementation of the protocol.

These approaches use an FSA as an event-coupled control structure, but many aspects of the specification of existing protocols must be relegated to HLL. In contrast, RTAG uses a more powerful control structure that includes simple data manipulation capabilities, and the use of HLL code can be considerably reduced. These differences result in several advantages of RTAG over the other systems:

- RTAG provides greater portability.

  The AFSM and FAPL systems provide automatic implementation only in a particular target language (C and PL/1 respectively), and, because their designs are strongly linked to the target language, it would most likely not be practical to retarget them to, say, an object-oriented or functional language. There are also likely to be problems resulting from differences between implementations of the target language on different systems.

  The RTAG system, on the other hand, provides an attractive form of portability. There is the one-time task of implementing an RTAG parser for each system, in whatever programming style is most appropriate to that system. This is likely to be an easier task than writing a C or PL/1 compiler. From that point, the task of implementing an arbitrary RTAG-specified protocol is reduced to writing a few packet-formatting and interface routines.

- RTAG specifications are easier to read and write.

  A grammar shows legal sequences of related events more compactly than an AFSM specification. In addition, RTAG's ability to express activities in a modular form, and to combine activities both sequentially and in parallel with synchronization, makes it easier for both reader and writer to isolate and correlate the mechanisms of the protocol.

  This is not to say that RTAG specifications are initially easy to write. However, once techniques are developed for expressing common protocol

features, writing a RTAG grammar for a new protocol is most likely easier than writing a complete and precise AFSM-type specification for the protocol.

- RTAG is better suited to expressing complex protocols.

  Special-purpose protocols for distributed operating systems, distributed database algorithms, and other applications could require different or more complex features than those of current data communications protocols. RTAG is well-suited to expressing this type of complexity, especially that involving the dynamic creation of concurrent subprotocols. AFSM-based systems, because they are based on a static FSA, must resort to HLL to express dynamic structure.

9.4. Concurrent programming languages

It was observed that RTAG can be viewed as concurrent programming language with very lightweight processes and an unusual form of process scheduling. The idea of organizing a protocol as a collection of lightweight processes has also been used in implementing TCP/IP on microcomputers ([MIT83]). It also raises the possibility of using a concurrent programming language such as Ada [DOD83], CSP [Hoa78] or StarMod [Coo81] for protocol specification. Like RTAG, such a specification language could express concurrent hierarchical structures (which, as argued above, is an important feature of a protocol specification formalism).

RTAG has the following advantages over such an approach:

- More portability.
- RTAG's mechanism for selecting among alternative productions, which is not part of conventional concurrent languages, and would have to be implemented *ad hoc* in each program.
- Efficiency: If a concurrent-language specification were run in a conventional multiprogramming environment, the expense of process creation, context switching, and interprocess communication would limit the efficiency of protocol implementations. The RTAG parser can be seen as a process scheduler and language interpreter that exploits the limited nature of RTAG process and does minimal work.

9.5. Comparison with Other FDT's

LOTOS is strongly related to RTAG in the sense that it allows decomposition into a dynamic set of subprotocols with serial and parallel composition, and it can express data dependencies. There are, however, the following major differences:

- In RTAG, attribute values can be re-assigned, and subprocesses can communicate via attribute assignments, as in Section 5.5. In LOTOS, value-identifiers (which correspond to attributes) can be assigned once only, and all communication between processes is via messages.

- LOTOS is designed for protocol verification rather than automated implementation. The semantics of LOTOS are defined in terms of legal "observable action sequences", and say nothing about the algorithm to be executed by an implementation of the specified protocol.

Harangozo's work [Har77] uses grammars and attempts to specify real-world protocols (in his case, the HDLC data link protocol). Harangozo uses only regular grammars; all productions are of the form

$$A \rightarrow X\ B$$

where A and B are nonterminals and X is an input or output event. This is equivalent to FSA.

To model sequence numbers, Harangozo attaches a numeric attribute to certain symbols, and uses notation of the form

$$A(i) \rightarrow X(i)\ B(i+1)$$

to denote "indexed families" of productions.

Harangozo's work does not include notions of enabling conditions, nonlocal attribute references, timers, etc., and so the formalism is probably inadequate for specifying real protocols.

Teng and Liu [Ten78] use a formalism based on context-free grammars. Their system lacks attributes and explicit timers and is hence incapable of expressing complex protocols. They articulate some of the principles and advan-

tages in using grammars: that logically independent protocol activities can be isolated, and that automated parsers can be built. However, they fail to provide a means for specifying concurrency within an entity. They also fail to distinguish between input and output events, or to notice that the necessity of performing output events in real time makes conventional parsing techniques inapplicable.

CHAPTER 10

CONCLUSION

10.1. Summary

RTAG has been developed to allow complete, concise, and well-structured specification of complex communications protocols. Furthermore, automated generation of protocol implementations based on RTAG specifications has been achieved, and the performance of these implementations is acceptable for many applications. Thus, RTAG can serve as the basis for many protocol-related activities, ranging from bringing up international-standard communications protocols to implementing distributed algorithms on heterogeneous local area networks.

We believe that, with additional development, the RTAG methodology can have a major effect on the way protocols for a wide range of applications are specified and implemented.

10.2. Directions for Future Work

Considerable work remains to be done towards the goal of obtaining RTAG-based implementations whose efficiency is competitive with that of conventional implementations. In addition, RTAG specifications of more standard protocols need to be written.

RTAG can be viewed as means of specifying complex real-time interactions, and may be applicable to areas outside data communications. For example, it may be useful in specifying user interfaces in which the man/machine dialogue involves several I/O devices, and in expert systems as a way of expressing certain types of time-dependent activities or knowledge.

It is likely that future applications of RTAG will suggest additional semantic features. For example, it may be useful to provide a multi-level "key attribute" mechanism, and to extend the priority mechanism to provide closer control over process scheduling and production selection,

10.3. Acknowledgments

I would like to thank Allan Bricker and Tad Lebeck, who provided considerable help with the UNIX kernel implementation. The TP-4 interface routines and the UNIX kernel simulation consist mostly of their code. Marvin Solomon and Steve Patterson both provided valuable suggestions, and Steve also made improvements to the RTAG compiler. My work has been supported in part by Bell Laboratories and IBM.

APPENDIX I: RTAG SPECIFICATION OF TP-4

The following is the RTAG specification of the TP-4 fragment described in Chapter 5.

```
/* RTAG specification of a TP-4 subset, based on NBS spec.

David P. Anderson, Computer Sciences Dept., Univ. of Wisconsin - Madison
started 6/84, last revised 8/85.

Some differences from NBS spec:
        interface to user level is different, to accomodate interactions
          with buffer manager (and no "system read").
        interface to network level is datagram, not virtual circuit.
*/

/* macro definitions for timers (200 msec units for VAX/4.2UNIX) */

#define QUIET_TIME          40      /* rest period after reboot */
#define RETRANS_TIME        10      /* retransmission interval */
#define GIVEUP_TIME         40      /* time after last retransmission to end connection */
#define INACTIVITY_TIME     60      /* end connection if no activity in this time */
#define REF_WAIT_TIME       80      /* don't re-use ref# after conn ends */
#define WINDOW_TIME         20      /* transmit acks periodically */

/* other tuneable parameters */

#define RETRANS_COUNT       4       /* number of retransmissions before give up */
#define MAXSEQ              128

/************************************************
                SYMBOL DEFINITIONS
************************************************/

key     refno           /* define key attribute name */

/* NETWORK EVENTS:  These correspond to different types of TPDU's.
The symbol and attribute names should be self-explanatory */

input   [N-CR]                  /* connection request TPDU */
        int     refno
        int     src_ref
        int     from_net_addr
        int     from_port
        int     to_port
        int     credit

output  [N-CR]
        int     src_ref
        int     from_port
```

```
            int     to_port
            int     credit

    input   [N-CC]                  /* connection confirm TPDU */
            int     refno
            int     src_ref
            int     credit

    output  [N-CC]
            int     src_ref
            int     dst_ref
            int     from_port
            int     to_port
            int     credit

    input   [N-DR]                  /* disconnect request TPDU */
            int     refno

    output  [N-DR]
            int     src_ref
            int     dst_ref

    input   [N-DC]                  /* disconnect confirm TPDU */
            int     refno

    output  [N-DC]
            int     src_ref
            int     dst_ref

    input   [N-GR]                  /* graceful close TPDU */
            int     refno
            int     seqno

    output  [N-GR]
            int     src_ref
            int     dst_ref
            int     seqno

    input   [N-DT]                  /* data TPDU */
            int     refno
            boolean eot
            int     seqno
            dataptr data

    output  [N-DT]
            int     src_ref
            int     dst_ref
            boolean eot
            int     seqno
            dataptr data
```

```
input   [N-XD]                  /* expedited data TPDU */
        int     refno
        int     xseqno
        dataptr data

output  [N-XD]
        int     src_ref
        int     dst_ref
        int     xseqno
        dataptr data

input   [N-AK]                  /* acknowledgement TPDU */
        int     refno
        int     credit
        int     seqno

output  [N-AK]
        int     src_ref
        int     dst_ref
        int     credit
        int     seqno

input   [N-XAK]                 /* expedited acknowledgement TPDU */
        int     refno
        int     xseqno

output  [N-XAK]
        int     src_ref
        int     dst_ref
        int     xseqno

/* EVENTS AT UPPER-LEVEL INTERFACE */

input   [U-CR]                  /* connection request */
        int     refno
        int     from_port
        int     to_net_addr
        int     to_port

output  [U-ACC]                 /* connection confirmation at active end */
        int     refno

output  [U-PCC]                 /* connection confirmation at passive end */
        int     refno

input   [U-DR]                  /* disconnect request */
        int     refno

output  [U-DR]                  /* disconnect indication */
        int     refno
```

```
output  [U-FR]                  /* free reference number */
        int     refno

input   [U-GR]                  /* graceful close request */
        int     refno

output  [U-GR]                  /* graceful close indication */
        int     refno

input   [U-DT]                  /* send regular data */
        int     refno
        dataptr data

output  [U-DT]                  /* data indication */
        int     refno
        dataptr data

output  [U-AK]                  /* notification that data has been acknowledged */
        int     refno
        dataptr data

input   [U-AK]                  /* user process has accepted data */
        int     refno

input   [U-XD]                  /* send expedited data */
        int     refno
        dataptr data

output  [U-XD]                  /* expedited data indication */
        int     refno
        dataptr data

input   [U-FIN]                 /* no new connections allowed */


/* NONTERMINAL SYMBOLS */

goal    <goal>

nonterm <TC tail>

nonterm<TC>                     /* handles one transport connection */
        int     refno           /* local transport connection reference # */
        int     foreign_refno
        int     foreign_net_addr
        int     local_port
        int     foreign_port
        boolean connfailed      /* set if connection establishment fails */
        boolean transerror      /* set if transaction error occurs */
```

```
nonterm <connect>                /* connection establishment */

nonterm <active open>

nonterm <passive open>

nonterm <retransmit CR>
        int     count

nonterm <get CR>

nonterm <deliver CC>

nonterm <retransmit CC>
        int     count

nonterm <disconnect>   /* disconnection subprotocol */

nonterm <deliver DR>

nonterm <retransmit DR>
        int     count

nonterm <ref wait>

nonterm <transact>               /* data transfer and graceful close */
        boolean start_send
        boolean start_recv
        int     initial_credit
        int     nxoutstanding    /* # of unacknowledged expedited packets */
        boolean GRarrived        /* true if have received in-sequence GR. */
        boolean GRsent           /* true if have delivered GR */

nonterm <reg send>
        int     nextseq          /* next seqno to assign */
        int     windowend        /* seqno just beyond end of send window */

nonterm <send msg tail>
        boolean ready

nonterm <recv acks>

nonterm <send msg>

nonterm <transmit GR>

nonterm <retransmit GR>
        int count

nonterm <transmit msg>
        dataptr data
```

```
nonterm <send packet>
        dataptr data
        int     seqno
        boolean eot

nonterm <send packet tail>
        boolean eot

nonterm <deliver DT>

nonterm <retransmit DT>
        int     count

nonterm <exp send>

nonterm <exp send tail>
        boolean ready
        int     xseqno

nonterm <send exp packet>
        int     xseqno

nonterm <deliver xdata>
        dataptr data

nonterm <retransmit xdata>
        int     count

nonterm <reg recv>
        int     recv_next       /* first seqno in receive window */
        int     window_end      /* first seqno beyond receive window */

nonterm <send acks>             /* handles acknowledgements */

nonterm <recv packet>
        int     seqno           /* sequence # of packet */

nonterm <recv packet tail>
        int     seqno
        dataptr data            /* message thus far */

nonterm <transfer data>
        dataptr data
        int seqno
        boolean eot

nonterm <send ack>

nonterm <exp recv>
        int     xrecv_next
```

```
nonterm <exp recv tail>
        int     xrecv_next

nonterm <process GRP>
        int seqno

/***********************************************
        DECLARATIONS OF EXTERNAL FUNCTIONS
***********************************************/

extern  boolean #between1#(int,int,int)  /* (i,j,k); true if i < j <= k in cyclic order */
extern  boolean #between2#(int,int,int)  /* (i,j,k); true if i <= j < k in cyclic order */
extern  dataptr #extract#(dataptr)       /* copy initial segment of data */
extern  int #bufslots#(int)              /* number of rec. window slots for connection */
extern  boolean #eot#(dataptr)           /* true if data is empty string */
extern  dataptr #copy#(dataptr)          /* copy data */
/

/***********************************************
        MULTIPLE TRANSPORT CONNECTIONS
***********************************************/

<goal>              :       /timer/ <TC tail>.
                    $1.interval = QUIET_TIME
;

<TC tail>       :       {<TC> <TC tail>}.
                |       [U-FIN].
;

<TC>            :       {<connect> <transact> <disconnect> }.
                $0.transerror = false
                $0.connfailed = false
;

<connect>       :       <active open>.
                |       <passive open>.
;

/***********************************************
                ACTIVE OPEN
***********************************************/

<active open>   :       [U-CR] [N-CR] <retransmit CR>.
                <TC>.foreign_net_addr = $1.to_net_addr
                <TC>.local_port = $1.from_port
                <TC>.foreign_port = $1.to_port
                <TC>.refno = $1.refno
                $2.src_ref = <TC>.refno
                $2.from_port = <TC>.local_port
                $2.to_port = <TC>.foreign_port
```

```
                $2.credit = #bufslots#($1.refno)
                $3.count = RETRANS_COUNT

    ;

    <retransmit CR>     :       /timer/ [N-CR] <retransmit CR>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                $2.src_ref = <TC>.refno
                $2.from_port = <TC>.local_port
                $2.to_port = <TC>.foreign_port
                $2.credit = #bufslots#(<TC>.refno)
                $3.count = $0.count - 1

                |       /timer/ .
                if $0.count = = 0
                $1.interval = GIVEUP_TIME
                <TC>.connfailed = true

                |       [N-CC] [U-ACC] [N-AK].
                <TC>/<transact>.initial_credit = $1.credit
                <TC>/<transact>.start_send = true
                <TC>/<transact>.start_recv = true
                <TC>.foreign_refno = $1.src_ref
                $2.refno = $1.refno
                $3.src_ref = <TC>.refno
                $3.dst_ref = <TC>.foreign_refno
                $3.seqno = 0
                $3.credit = #bufslots#(<TC>.refno)

    ;


    /***********************************************
                PASSIVE OPEN
    ***********************************************/

    <passive open>:     [N-CR] <deliver CC>.
                <TC>.refno = $1.refno
                <TC>.foreign_refno = $1.src_ref
                <TC>.foreign_net_addr = $1.from_net_addr
                <TC>.local_port = $1.to_port
                <TC>.foreign_port = $1.from_port
                <TC>/<transact>.initial_credit = $1.credit
                <TC>/<transact>.start_recv = true
    ;

    <deliver CC>  :     [N-CC] <retransmit CC>.
                $1.src_ref = <TC>.refno
                $1.dst_ref = <TC>.foreign_refno
                $1.from_port = <TC>.local_port
```

```
                $1.to_port = <TC>.foreign_port
                $1.credit = #bufslots#(<TC>.refno)
                $2.count = RETRANS_COUNT
;

<retransmit CC> :       /timer/ [N-CC] <retransmit CC>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                $2.src_ref = <TC>.refno
                $2.dst_ref = <TC>.foreign_refno
                $2.from_port = <TC>.local_port
                $2.to_port = <TC>.foreign_port
                $2.credit = #bufslots#(<TC>.refno)
                $3.count = $0.count - 1

                |       /timer/.
                if $0.count = = 0
                $1.interval = GIVEUP_TIME
                <TC>.connfailed = true

                |       [N-AK] [U-PCC].
                if $1.seqno = = 0
                <TC>/<transact>.start_send = true
                $2.refno = <TC>.refno

                |       [N-DT] [U-PCC].
                if #between2#(0, $1.seqno, <TC>/<transact>/<reg recv>.window_end)
                <TC>/<transact>.start_send = true
                $2.refno = <TC>.refno

                |       [N-XD].
                if $1.xseqno = = 0
                <TC>/<transact>.start_send = true
;


/*****************************************************
                DISCONNECTION SUBPROTOCOL
*****************************************************/

<disconnect>  :       [U-DR] /remove/ /remove/ <deliver DR> <ref wait>.
                if <TC>.transerror || <TC>.connfailed
                $1.refno = <TC>.refno
                $2.where = <TC>/<transact>
                $3.where = <TC>/<connect>

                |       /remove/ /remove/ <ref wait>.
                if <TC>/<transact>.GRsent && <TC>/<transact>.GRarrived
                $1.where = <TC>/<transact>
                $2.where = <TC>/<connect>
```

```
                |       [U-DR] /remove/ /remove/ <deliver DR> <ref wait>.
                $2.where = <TC>/<transact>
                $3.where = <TC>/<connect>

                |       [N-DR] /remove/ /remove/ [U-DR] [N-DC] <ref wait>.
                $2.where = <TC>/<transact>
                $3.where = <TC>/<connect>
                $4.refno = <TC>.refno
                $5.src_ref = <TC>.refno
                $5.dst_ref = <TC>.foreign_refno

;

<deliver DR>    :       [N-DR] <retransmit DR>.
                $1.src_ref = <TC>.refno
                $1.dst_ref = <TC>.foreign_refno
                $2.count = RETRANS_COUNT
;

<retransmit DR> :       /timer/ [N-DR] <retransmit DR>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                $2.src_ref = <TC>.refno
                $2.dst_ref = <TC>.foreign_refno
                $3.count = $0.count - 1

                |       /timer/.
                if $0.count == 0
                $1.interval = GIVEUP_TIME

                |       [N-DC].
;

<ref wait>      :       /timer/ [U-FR] .
                $1.interval = REF_WAIT_TIME
                $2.refno = <TC>.refno

                |       [N-DR] [N-DC] <ref wait>.
                $2.src_ref = <TC>.refno
                $2.dst_ref = <TC>.foreign_refno

                |       [N-DT] <ref wait>.

                |       [N-GR] <ref wait>.
;

/*****************************************************
                TRANSACTION SUBPROTOCOL
        Handles sending and receiving of regular and expedited data.
        If a timeout error occurs, sets <TC>.transerror and
        lets the disconnection phase finish up.
```

```
        If close gracefully, set flags in <TC>.
*****************************************************/

<transact>      :       { <reg send> <exp send> <reg recv> <exp recv> }.
                if $0.start_recv
                $0.nxoutstanding = 0
                $0.GRarrived = false
                $0.GRsent = false
;

/*************************************************
                REGULAR SEND
*************************************************/

<reg send>      :       {<send msg tail> <recv acks>}.
                if <transact>.start_send
                $0.nextseq = 0
                $0.windowend = <transact>.initial_credit
                $1.ready = true
;

<recv acks>     :       /timer/.
                $1.interval = INACTIVITY_TIME
                <TC>.transerror = true

                |       [N→AK] <recv acks>.
                <reg send>.windowend = ($1.seqno + $1.credit) mod MAXSEQ
;

<send msg tail> :       {<send msg> <send msg tail>}.
                $2.ready = false

                |       [U→GR] <transmit GR>.
;

<send msg>      :       [U→DT] <transmit msg>.
                $2.data = $1.data
;

<transmit GR>:          [N←GR] <retransmit GR>.
                if <send msg tail>.ready
                $1.src_ref = <TC>.refno
                $1.dst_ref = <TC>.foreign_refno
                $1.seqno = <reg send>.nextseq
                $2.count = RETRANS_COUNT
;

<retransmit GR> :       /timer/ [N←GR] <retransmit GR>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                $2.src_ref = <TC>.refno
```

```
                $2.dst_ref = <TC>.foreign_refno
                $2.seqno = <reg send>.nextseq
                $3.count = $0.count - 1

                |       [N-AK].
                if $1.seqno == ((<reg send>.nextseq + 1) mod MAXSEQ)
                <transact>.GRsent = true

                |       /timer/.
                if $0.count == 0
                $1.interval = GIVEUP_TIME
                <TC>.transerror = true
;

<transmit msg> :        <send packet tail>.
                if <send msg tail>.ready
                $1.eot = #eot#($0.data)
;

<send packet tail> :    {<send packet> <send packet tail>}.
                if not $0.eot
                $1.data = #extract#(<transmit msg>.data)
                $1.eot = #eot#(<transmit msg>.data)
                $1.seqno = <reg send>.nextseq
                <reg send>.nextseq = (<reg send>.nextseq + 1) mod MAXSEQ
                $2.eot = $1.eot

                |       /freedata/.
                if $0.eot
                $1.data = <transmit msg>.data
                <send msg tail>/<send msg tail>.ready = true
;

<send packet> :         [N-DT] <retransmit DT>.
                if (<reg send>.nextseq != <reg send>.windowend)
                        && (<transact>.nxoutstanding == 0)
                $1.src_ref = <TC>.refno
                $1.dst_ref = <TC>.foreign_refno
                $1.eot = $0.eot
                $1.seqno = $0.seqno
                $1.data = #copy#($0.data)
                $2.count = RETRANS_COUNT
;

<retransmit DT> :       /timer/ [N-DT] <retransmit DT>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                $2.src_ref = <TC>.refno
                $2.dst_ref = <TC>.foreign_refno
                $2.eot = <send packet>.eot
                $2.seqno = <send packet>.seqno
```

```
                $2.data = #copy#(<send packet>.data)
                $3.count = $0.count - 1

                |       [N-AK] [U-AK].
                if #between1#(<send packet>.seqno, $1.seqno, <reg send>.nextseq)
                $2.refno = <TC>.refno
                $2.data = <send packet>.data

                |       /timer/.
                if $0.count == 0
                $1.interval = GIVEUP_TIME
                <TC>.transerror = true
;

/***********************************************
        EXPEDITED DATA SENDING
***********************************************/

<exp send>      :       <exp send tail>.
                if <transact>.start_send
                $1.xseqno = 0
                $1.ready = true
;

<exp send tail> :       {<send exp packet> <exp send tail>}.
                $1.xseqno = $0.xseqno
                $2.ready = false
                $2.xseqno = ($0.xseqno + 1) mod MAXSEQ

                |       .
                if <transact>.GRsent
;

<send exp packet> :     [U-XD] <deliver xdata>.
                <transact>.nxoutstanding = <transact>.nxoutstanding + 1
                $2.data = $1.data
;

<deliver xdata>         :       [N-XD] <retransmit xdata>.
                if <exp send tail>.ready
                $1.src_ref = <TC>.refno
                $1.dst_ref = <TC>.foreign_refno
                $1.data = #copy#($0.data)
                $2.count = RETRANS_COUNT
;

<retransmit xdata> :    /timer/ [N-XD] <retransmit xdata>.
                if $0.count > 0
                $1.interval = RETRANS_TIME
                $2.src_ref = <TC>.refno
                $2.dst_ref = <TC>.foreign_refno
```

```
                $2.data = #copy#(<deliver xdata>.data)
                $3.count = $0.count - 1

                |       [N→XAK] /freedata/.
                if $1.xseqno == <send exp packet>.xseqno
                $2.data = <deliver xdata>.data
                <exp send tail>/<exp send tail>.ready = true
                <transact>.nxoutstanding = <transact>.nxoutstanding - 1

                |       /timer/.
                if $0.count == 0
                $1.interval = GIVEUP_TIME
                <TC>.transerror = true
;

/*******************************************************
                REGULAR RECEIVE
*******************************************************/

<reg recv> :    {<recv packet tail> <send acks>}.
                $0.recv_next = 0
                $0.window_end = #bufslots#(<TC>.refno)
                $1.data = empty
                $1.seqno = 0
;

<recv packet tail> :    {<recv packet> <recv packet tail>}.
                if #between2#(<reg recv>.recv_next, $0.seqno, <reg recv>.window_end)
                $1.seqno = $0.seqno
                $2.seqno = ($0.seqno + 1) mod MAXSEQ
;

<recv packet> : [N→DT] <transfer data>.
                if $0.seqno == $1.seqno
                $2.data = $1.data
                $2.eot = $1.eot
                $2.seqno = $0.seqno

                |       [N→GR] /remove/ <process GRP>.
                if $1.seqno == $0.seqno
                $2.where = <recv packet tail>/<recv packet tail>
                $3.seqno = $0.seqno
;

<transfer data>:        .
                if (not $0.eot) && ($0.seqno == <reg recv>.recv_next)
                <reg recv>.recv_next = ($0.seqno + 1) mod MAXSEQ
                <recv packet tail>/<recv packet tail>.data =
                        <recv packet tail>.data cat $0.data
```

```
                |       [U-DT].
                if $0.eot && ($0.seqno == <reg recv>.recv_next)
                <reg recv>.recv_next = ($0.seqno + 1) mod MAXSEQ
                $1.refno = <TC>.refno
                $1.data = <recv packet tail>.data cat $0.data
                <recv packet tail>/<recv packet tail>.data = empty
;

<process GRP>   :       [U-GR].
                if $0.seqno == <reg recv>.recv_next
                <reg recv>.recv_next = (<reg recv>.recv_next + 1) mod MAXSEQ
                <transact>.GRarrived = true
                $1.refno = <TC>.refno
;

<send acks>     :       /timer/ <send ack> <send acks>.
                $1.interval = WINDOW_TIME

                |       [N-DT] <send ack> <send acks>.

                |       [N-GR] <send ack> <send acks>.

                |       [U-AK] <send ack> <send acks>.
                <reg recv>.window_end =
                        (<reg recv>.recv_next + #bufslots#(<TC>.refno)) mod MAXSEQ
;

<send ack>      :       [N-AK].
                $1.src_ref = <TC>.refno
                $1.dst_ref = <TC>.foreign_refno
                $1.seqno = <reg recv>.recv_next
                $1.credit = #bufslots#(<TC>.refno)
;


/*******************************************************
                EXPEDITED RECEIVE
*******************************************************/

<exp recv>      :       <exp recv tail>.
                $1.xrecv_next = 0
;

<exp recv tail> :       [N-XD] [U-XD] [N-XAK] <exp recv tail>.
                if $1.xseqno == $0.xrecv_next
                $2.refno = <TC>.refno
                $2.data = $1.data
                $3.src_ref = <TC>.refno
                $3.dst_ref = <TC>.foreign_refno
                $3.xseqno = $0.xrecv_next
                $4.xrecv_next = ($0.xrecv_next + 1) mod MAXSEQ
```

```
|       [N-XD] [N-XAK] <exp recv tail>.
if $1.xseqno != $0.xrecv_next
$2.src_ref = <TC>.refno
$2.dst_ref = <TC>.foreign_refno
$2.xseqno = $0.xrecv_next
$3.xrecv_next = $0.xrecv_next

|       .
if <transact>.GRarrived
;
```

APPENDIX II: RTAG FORMAL SYNTAX

The following is the *Lex* input file used to generate the scanner portion of the RTAG compiler.

```
alpha           [a-zA-Z]
digit           [0-9]
%%
key             return(KEY);
int             return(INT);
boolean         return(BOOLEAN);
dataptr         return(DATAPTR);
empty           return(EMPTY);
procid          return(PROCID);
output          return(OUTPUT);
input           return(INPUT);
nonterm         return(NONTERM);
goal            return(GOAL);
extern          return(EXTERN);
if              return(IF);
true            return(TRUE);
false           return(FALSE);
not             return(NOTOP);
mod             return(MODOP);
cat             return(CONCATOP);
"["({alpha}|{digit}|"-"|"<"|">")+"]"  {yylval.ptrval = yytext; return(TERMSYM);}
"<"({alpha}|{digit}|" ")+">"  {yylval.ptrval = yytext; return(NONTERMSYM);}
"/"({alpha}|{digit}|"_")+"/"    {yylval.ptrval=yytext; return(SPECIALSYM);}
"#"({alpha}|{digit}|"_")+"#"    {yylval.ptrval=yytext; return(EXTERNSYM);}
{alpha}({alpha}|{digit}|"_")*   {yylval.ptrval = yytext; return(ATTRNAME);}
-?{digit}+      {yylval.intval = atoi(yytext); return(NUMBER);}
"="             return(ASSIGNOP);
"=*"            return(SASSIGNOP);
"+"             return(PLUSOP);
"-"             return(MINUSOP);
"=="            return(RELEQ);
"!="            return(RELNE);
"<"             return(RELLT);
">"             return(RELGT);
"<="            return(RELLE);
">="            return(RELGE);
"&&"            return(ANDOP);
"||"            return(OROP);
","             return(COMMA);
"."             return(PERIOD);
":"             return(COLON);
```

```
";"         return(SEMICOLON);
"("         return(LPAREN);
")"         return(RPAREN);
"{"         return(LCURL);
"}"         return(RCURL);
"/"         return(SLASH);
"$"         return(DOLLAR);
"|"         return(BAR);
[ 0         ;
```

The following is the *Yacc* input file giving the context-free grammar description of RTAG syntax. The RTAG compiler is based on this grammar, with "action routine" code inserted at various points.

```
%token <intval> INT BOOLEAN DATAPTR PROCID KEY
        OROP ANDOP PLUSOP MINUSOP CONCATOP MODOP NOTOP
        COLON SEMICOLON PERIOD COMMA LPAREN RPAREN
        LCURL RCURL DOLLAR SLASH BAR
        INPUT OUTPUT NONTERM GOAL EXTERN SPECIAL
        PASSWORD SASSIGNOP ASSIGNOP NUMBER
        TRUE FALSE IF EMPTY
        RELLT RELLE RELEQ RELNE RELGE RELGT
%token <ptrval> NONTERMSYM TERMSYM SPECIALSYM EXTERNSYM ATTRNAME

%left OROP
%left ANDOP
%nonassoc RELEQ RELNE
%nonassoc RELLT RELLE RELGE RELGT
%left PLUSOP MINUSOP CONCATOP
%left MODOP
%left UMINUS NOTOP

%%
grammar     :       keydef symdefs productions
            ;

keydef      :
            |       KEY ATTRNAME
            ;

symdefs     :
            |       symdefs symdef
            ;
```

```
symdef      :   INPUT TERMSYM attrdefs
            |   OUTPUT TERMSYM attrdefs
            |   NONTERM NONTERMSYM attrdefs
            |   GOAL NONTERMSYM attrdefs
            |   EXTERN attrtype EXTERNSYM LPAREN params RPAREN
            |   error
            ;

params      :
            |   attrtype
            |   params COMMA attrtype
            ;

attrdefs    :
            |   attrdefs attrdef
            ;

attrdef     :   attrtype attrdeflist
            ;

attrdeflist :   ATTRNAME
            |   attrdeflist COMMA ATTRNAME
            ;

attrtype    :   INT
            |   BOOLEAN
            |   DATAPTR
            |   PROCID
            ;

productions :   production
            |   productions production
            ;

production  :   NONTERMSYM COLON prodlist SEMICOLON
            |   error SEMICOLON
            ;

prodlist    :   prod
            |   prodlist BAR prod
            ;

prod        :   rhs PERIOD enabling assignlist
            ;

rhs         :
            |   rhs rhssymbol
            |   rhs rhsgroup
            ;

rhsgroup    :   LCURL rhssymbollist RCURL
```

```
                ;

rhssymbollist   :   rhssymbol
                |   rhssymbollist rhssymbol
                ;

rhssymbol       :   NONTERMSYM
                |   TERMSYM
                |   SPECIALSYM
                ;

enabling        :
                |   IF expr
                ;

assignlist      :
                |   assignments
                ;

assignments     :   assignment
                |   assignments assignment
                ;

assignment      :   attref ASSIGNOP expr
                ;

attref          :   symexpr PERIOD ATTRNAME
                ;

symexpr         :   DOLLAR NUMBER
                |   NONTERMSYM
                        symlisttail
                ;

symlisttail     :
                |   SLASH NONTERMSYM symlisttail
                ;

expr            :   LPAREN expr RPAREN
                |   EXTERNSYM
                    LPAREN arglist RPAREN
                |   attref
                |   symexpr
                |   TRUE
                |   FALSE
                |   EMPTY
                |   expr OROP expr
                |   expr ANDOP expr
                |   NOTOP expr %prec UMINUS
                |   NUMBER
                |   expr PLUSOP expr
```

```
        |   expr MINUSOP expr
        |   expr MODOP expr
        |   MINUSOP expr %prec UMINUS
        |   expr CONCATOP expr
        |   expr RELEQ expr
        |   expr RELNE expr
        |   expr RELLT expr
        |   expr RELLE expr
        |   expr RELGT expr
        |   expr RELGE expr
        ;

arglist :
        |   expr
        |   arglist COMMA expr
        ;

%%
```

REFERENCES

And84

Anderson, D.P. and L.H. Landweber. Protocol specification by real-time attribute grammars. Proc. 4th IFIP Symp. on Protocol Spec., Testing and Verification (1984) 309-348.

Bjo70

Bjorner, D. Finite State Automaton definitions of data communication line control procedures. AFIPS Conference Proceedings, vol. 37 (1970) 477-491.

Blu82

Blumer, T.P. and R.L Tenney. A formal specification technique and implementation method for protocols, Computer Networks 6,3 (July 1982) 201-217.

Boc76

Bochmann, G.v. Semantic evaluation from left to right. Comm. ACM 19 (1976) 55-62.

Boc78

Bochmann, G.v. Finite state description of communication protocols, Computer Networks 2 (1978) 361-372.

Boc80

Bochmann, G.v. and C.A. Sunshine. Formal methods in communication protocol design. IEEE Trans. on Commun. 28,4 (April 1980) 624-631.

Bog80

Boggs, D.R., J.F. Shoch, E.A. Taft, and R.M Metcalfe. Pup: an internetwork architecture. IEEE Trans. on Commun. 28,4 (April 1980) 612-623.

Bow83

Bowers, A.W and E.B. Connell. A checklist of communications protocol functions organized using the open system interconnection seven-layer reference model. IEEE Comp. Con. (Fall 1983) 479-487.

Bri84

Brinksma, E. and G. Karjoth. A specification of the OSI transport service in LOTOS. Proc. 4th IFIP Symp. on Protocol Spec., Testing, and Verification (June 1984) 227-252.

Bri85

Brinksma, E. A tutorial on LOTOS. Proc. 5th IFIP Symp. on Protocol Spec., Testing, and Verification (June 1985).

Bur84

Burkhardt, H.J, H. Eckert and R. Prinoth. Modelling of OSI communication services and protocols using predicate/transition nets. Proc. 4th IFIP Symp. on Protocol Spec., Testing, and Verification (June 1984) 165-192.

Cer83

Cerf, V.G. and E. Cain. The DoD internet architecture model. Computer Networks 7,5 (Oct. 1983) 307-318.

Cla83

Clark, D. Modularity and efficiency in protocol implementation. RFC817, SRI Network Information Center (1983).

Coo81

Cook, R.P. Abstractions for distributed programming. Computer Languages 6, 3-4 (1981) 131-138.

Cyp78

Cypser, R.J. Communications architecture for distributed systems. Reading, Mass: Addison-Wesley, 1978.

Dan77

Danthine, A.A.S. Petri nets for protocol modeling and verification. Proc. Computer Networks and Teleprocess. Symp. (1977) 663-685.

Dan80

Danthine, A.A.S. Protocol representation with finite-state machines, IEEE Trans. on Commun., COM-28 no. 4 (April 1980) 632-643.

Day83

Day, J.D. and H. Zimmermann. The OSI reference model. Proc. IEEE 71,12 (Dec. 1983) 1334-1340.

Dem81

Demers, A., T. Reps and T. Teitlebaum. Incremental evaluation for attribute grammars with applications to syntax-directed editors. Eighth Annual ACM Symposium on Principles of Programming Languages (1981) 105-116.

DOD83

Reference manual for the ADA programming language. U.S. Department of Defense. Springer Verlag, N.Y., 1983.

Fle78

Fletcher, J.G. and R.W Watson. Mechanisms for a reliable time-based protocol. Computer Networks 2, 4-5 (Oct. 1978) 271-290.

Gan82

Ganapathi, M. and C.N. Fischer. Description-driven code generation using attribute grammars. Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, Jan. 25-27, 1982.

Gin82

Ginsparg, J.M. and D.R. Gordon. Automatic programming of communications software via nonprocedural descriptions, IEEE Trans. Commun., vol. COM-30 (June 1982) 1343-1348.

Gou76

Gouda, M.G. and E.G. Manning. On the modeling, analysis and design of protocols - a special class of software structures, Proc. 2nd International Conf. on Software Engineering, San Francisco (Oct. 1976) 256-262.

Gra79

Gray, J.P. and T.B. McNeill. SNA multiple-system networking. IBM Syst. J. 18 (1979) 263-297.

Hai80

Hailpern, B. and S. Owicki. Verifying network protocols using temporal logic. Proc. Trends and Appl. Symp., NBS (1980).

Har77

Harangozo, J. An approach to describing a link level protocol with a formal language. Proceedings 5th Data Communication Symposium (Oct. 1977) 4.37-4.49.

Har78

Harangozo, J. Protocol definitions with formal grammars. Proceedings Symposium on Computer Network Protocols, Liege, Belgium (Feb. 1978) F6.1-F6.10.

Hoa78

Hoare, C.A.R. Communicating Sequential Processes. Comm. ACM 21,8 (1978) 666-677.

Hof70

Hoffman, H.J. On linguistic aspects of communication line control procedures. IBM Report RZ 345, 1970.

ISO83

ISO. Basic reference model for Open Systems Interconnection. ISO 7498, 1983.

Joh79

Johnson, S.C. Yacc: Yet Another Compiler-Compiler. Unix Programmer's Manual, vol. 2B (1979).

Joh83

Johnson, G.F. An approach to incremental semantics. Ph.D. dissertation, University of Wisconsin - Madison, 1983.

Jon80

Jones, C.B. Software development: a rigorous approach. Prentice-Hall International, London, 1980.

Kat84

Katayama, T. Translation of attribute grammars into procedures. ACM Trans. on Prog. Lang. and Syst. 6,3 (1984) 345-369.

Ken76

Kennedy, K. and S.K. Warren. Automatic generation of efficient evaluators for attribute grammars. Third ACM Symposium on Principles of Programming Languages (1976) 32-49.

Ker78

Kernighan, B.W. and D.M Ritchie. The C Programming Language. Prentice-Hall, New Jersey, 1978.

Knu68

Knuth, D.E. Semantics of context free languages. Math. Systems Theory Journal 2, 2 (1968) 127-145.

Knu71

Knuth, D.E. Examples of Formal Semantics. Lecture Notes in Mathematics, vol. 188, Springer Verlag, New York, 1971.

Kos71

Koster, C.H.A. Affix grammars. In: ALGOL 68 Implementation (J.E. Peck, ed.) Amsterdam: North Holland (1971).

Les79

Lesk, M.E. and E. Schmidt. Lex - a lexical analyzer generator. Unix Programmer's Manual, vol. 2B (1979).

Log84

Logrippo, L., D. Simon and H. Ural. Executable description of the OSI transport service in Prolog. Proc. 4th IFIP Int. Workshop on Protocol Spec., Testing, and Verification (June 1984) 279-294.

Mer76

Merlin, P.M. A methodology for the design and implementation of communication protocols. IEEE Trans. Commun., Com-24, (1976) 614-616.

Mer79

Merlin, P.M. Specification and validation of protocols. IEEE Trans. Commun. (Nov. 1979) 1671-1680.

Mil80

Milner, R. A calculus of communicating systems. Springer Verlag, 1980.

MIT83

Laboratory for Computer Science Progress Report. MIT, June 1983.

Mol82

Molloy, M.K. Performance analysis using stochastic Petri nets. IEEE Trans. Comput., C-31, 9 (Sept. 1982) 913-917.

Nas83

Nash, S.C. Automated implementation of SNA communication protocols. IEEE International Conference on Communication. Boston, MA., (June 19-22, 1983), 1316-1322.

NBS83

Specification of a transport protocol for computer communications, volume 3: Class 4 protocol, Report ICST/HLNP-83-3, National Bureau of Standards, Washington, D.C., 1983.

Ozs82

Ozsu, M.T. and B.W. Weide. Modeling of distributed database concurrency control mechanisms using an extended Petri net formalism. Proc. IEEE 1982 660-665.

Par84

Parrow, J. and R. Gustavsson. Modelling distributed systems in an extension of CCS with infinite experiments and temporal logic. Proc. 4th IFIP Symp. on Protocol Spec., Testing and Verification (June 1984) 309-348.

Pet77

Peterson, J.L. Petri nets. Computing Surveys 9 (1977) 223-252.

Pop85

Pope, A.R. Encoding CCITT X.409 presentation transfer syntax. Bolt Beranek and Newman Inc. (1985).

Pou78

Pouzin, L. and H. Zimmermann. A tutorial on protocols. Proc. IEEE 66 (1978) 1346-1370.

Poz82

Pozefsky, D.P. and F.D. Smith. A meta-implementation for Systems Network Architecture. IEEE Trans. Commun., vol. COM-30 (1982) 1348-1355.

Rub82

Rubin, J. and C.H. West. An improved protocol validation technique. Computer Networks 6,2 (1982) 65-73.

Saa78

Saarinen, M. On constructing efficient evaluators for attribute grammars. Lecture Notes in Computer Science, vol. 62. Springer Verlag, New York, 1978.

ScF80

Schindler, S., U. Flasche, and D. Altenkruger. The OSA project: formal specification of the ISO transport service. Proceedings of the Computer Network Symposium, NBS, Dec. 1980.

Sch80

Schultz, G.D., D.B. Rose, C.H. West, and J.P. Gray. Executable Description and Validation of SNA. IEEE Trans. Commun., vol COM-28 (1980) 661-677.

Sco85

Scott, M.L. Design and implementation of a distributed systems language. Ph.D. Thesis, Computer Sciences Department, University of Wisconsin - Madison, 1985.

Sed80

Sedillot, S. and G. Sergeant. A protocol for distributed execution and consistent resource allocation. IEEE COMPCON (1980) 535-542.

Sid83a

Sidhu, D.P. Protocol verification via executable logic specifications. Protocol Specification, Testing and Verification, III (H. Rudin and C.H. West, editors). Elsevier Science Publishers B.V. (North Holland) 1983.

Sid83b

Sidhu, D.P. and T.P. Blumer. Verification of NBS class 4 transport protocol. SDC Report (Sept. 1983).

Ske78

Skedzeleski, S.K. Definition and use of attribute reevaluation in attributed grammars. Ph.D. dissertation, University of Wisconsin - Madison, 1978.

Stu83

Studnitz, P v. Transport protocols: their performance and status in international standardization (July 1982). Computer Networks 7,5 (Oct. 1983) 27-35.

Sun78

Sunshine, C.A. and Y.K. Dalal. Connection management in transport protocols. Computer Network 2,6 (1978) 454-473.

Sun79

Sunshine, C.A. Formal techniques for protocol specification and verification. Computer 12 (1979) 20-27.

Sun80

Sunshine, C.A. Formal modeling of communication protocols. USC/ISI report ISI/RR-81-89 (1981).

Tan81

Tanenbaum, A.S. Computer networks. Prentice-Hall, Inc., New Jersey, 1981.

Ten78

Teng, A.Y. and M.T. Liu. A formal approach to the design and implementation of network communication protocols, Proc. COMPSAC 78, Chicago (Nov. 1978) 722-727.

UNI83

The Berkeley UNIX Programmer's Manual, volume 2C. Computer Systems Research Group, Dept. of Electrical Engineering and Comp. Sci., University of California, Berkeley CA.

Vis83

Vissers, C.A., R.L. Tenney and G.v. Bochmann. Formal description techniques. Proc. IEEE 71,12 (Dec. 1983) 1356-1364.

Wat77

Watt, D.A. The parsing problem for affix grammars. Acta Informatica 8 (1977) 1-20.

Wec80

Wecker, S. DNA: the Digital Network Architecture. IEEE Trans. Commun., COM-28 (April 1980) 510-526.

Yem83

Yemini, Y. and N. Nounou. CUPID: a protocol development environment. Proc. 3rd IFIP Workshop on Protocol Specification, Testing and Verification, May 1983.

Zim80

Zimmermann, H. OSI reference model - the ISO model of architecture for open systems interconnection. IEEE Trans. Commun. 28 (1980) 425-432.