Automated Protocol Implementation with RTAG

DAVID P. ANDERSON

Abstract—**RTAG is a language based on an attribute grammar notation for specifying protocols. Its main design goals are: 1) to support concise and easily understood expression of complex real-world protocols, and 2) to serve as the basis of a portable software system for automated protocol implementation.**

This paper summarizes the RTAG language, gives examples of its use, sketches the algorithms used in generating implementations from these specifications, and describes a UNIX®-based automated implementation system for RTAG.

Index Terms—**Attribute grammars, communication protocols, formal specification.**

I. Introduction

THE number of communication protocol families, operating systems, communication-based applications, network architectures, and network interface hardware types is large and constantly increasing. Thus, considerable programming effort is required to directly implement the protocols needed for communication in large heterogeneous systems.

The separation of communication services into layers with well-defined interfaces, as is done by the ISO model [24], improves the situation because it reduces interlayer dependencies. However, system designers are still faced with the problem of implementing protocols efficiently on a range of operating systems.

The software engineering of communication protocols is complicated by two factors: 1) for efficiency reasons, protocols often must be placed in operating system kernels where debugging is difficult and interfaces are highly system-specific [5]; 2) protocol implementations must conform to an externally defined (and often poorly defined) standard. In contrast, other system software (such as a local operating system service) can serve as its own authoritative specification.

Problem 1 can be minimized by a modular kernel organization that provides well-defined interfaces between protocols and adjacent layers; this has been done, for example, in 4.3 BSD UNIX [12]. Problem 2 can be addressed by the use of *formal description techniques* (FDT's) for specifying protocols. The role of FDT's is discussed in [20], and a survey is given in [19].

Most FDT's in current use are based on finite state machines (FSM), and are oriented towards verification and/or performance analysis. Pure FSM's are amenable to automated protocol verification, but can represent 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 [2], [15].

Manuscript received February 15, 1987; revised October 1, 1987. This work was supported by the IBM Corporation and by the National Science Foundation under Grant DCR-8619302.

The author is with the Department of Electrical Engineering and Computer Science, Computer Science Division, University of California, Berkley, CA 94720.

IEEE Log Number 8718690.

®UNIX is a registered trademark of AT&T Bell Laboratories.

A. RTAG: A Grammar-Based Formal Description Technique

RTAG (real-time asynchronous grammars) is a formal description technique designed with the following goals:

- To make it easy to specify complex protocols.
- To allow efficient automated implementation.

RTAG 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.

For certain applications, RTAG has significant advantages over FSM-based techniques. It supports encapsulation and abstraction in protocol design and specification by allowing a protocol to be decomposed into *subprotocols* that are specified separately. These subprotocols may be instantiated dynamically and may operate concurrently. RTAG allows most protocol mechanisms to be expressed without resort to HLL code; this is in large part due to its use of grammars, rather than FSM, as the underlying formalism.

In addition, it is possible to automatically generate efficient protocol implementations based on RTAG specifications. This is done using an RTAG *parser*, a real-time program that, given an RTAG specification, responds to input messages in a way that satisfies the specification. This provides a major part of an implementation of the protocol. Essentially, only packet assembly/disassembly and interface routines need be added.

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 direct implementations.

We have developed an RTAG parser under 4.3 BSD UNIX and have written an RTAG specification of the NBS Class 4 Transport Protocol, TP-4 [22]. 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.

This paper is organized as follows: Section II describes the syntax and semantics of RTAG; Section III gives examples from the RTAG TP-4 specification; Section IV sketches algorithms for an RTAG parser; Section V describes a UNIX-based RTAG development system; Section VI compares RTAG to related work, and Section VII gives conclusions.

II. RTAG Syntax and Semantics

This section summarizes the syntax and semantics of RTAG. An RTAG specification defines the behavior of a protocol entity that must respond to input messages, and to the passage of real time, by generating output messages. RTAG uses a context-free grammar notation, but its semantics will be defined in terms of a process-based model. Each grammar symbol X is associated with a process definition P_X. Event (terminal) symbol processes read or write a single message, and each production defines a nonterminal process as the sequential or parallel composition of subprocesses. For example, the production

$$<x> \quad : \quad <y><z>$$

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

In an RTAG protocol entity there is at any point a tree of process *instances*; initially there is an instance of the goal-symbol process. The life of a process is as follows: it is first *instantiated*, creating its per-process variables, and 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 occur, time to elapse, or expressions to become true.

RTAG semantics are described in terms of processes for pedagogical purposes only. In an implementation, RTAG processes would be represented by symbol descriptors with a field encoding the process state, rather than by distinct processes at the operating system level.

A. Symbols and Attributes

RTAG symbols are divided into the following classes:

- *Nonterminal symbols* are delimited by angle brackets (<, >).
- *Event symbols* (both input and output) are delimited by square brackets ([,]). By convention, the symbol name contains a character indicating the other protocol layer involved, followed by an arrow indicating whether the symbol is an input or output symbol. 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.
- *Special terminal symbols* represent internal actions of the protocol entity; their names are delimited with slashes. There are two special terminal symbols; **/timer/** and **/remove/**.

Each symbol has an associated set of attributes, each of which has a type from among the following: **integer**, **boolean**, **dataptr** (pointer to message data), and **symbol-ref** (pointer to a symbol instance; used by the **/remove/** special terminal symbol).

The semantics of a terminal symbol X are as follows:

Output Symbols: P_X sends the corresponding message, using the attribute values of X as values for the message fields. In practice, this is done by calling an *event performance routine*, parameterized by the attribute values of X.

Input Symbols: P_X blocks until an arrival of the corresponding message. Its per-process variables are then initialized with the values contained in the message fields.

Special Terminal Symbols: **/timer/** has an integer attribute *interval*. $P_{/\text{timer}/}$ 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 Y. When an instance of $P_{/\text{remove}/}$ is executed, Y and all its descendants in the process tree are aborted and flagged as finished. This is used, for example, in handling abnormal closure of connections.

B. Attribute References and Expressions

Expressions associated with a production P can refer to symbols and attributes of symbols that are *local* (within P), or *nonlocal* (at a relative position in the process tree). A local reference is of the form **$n**, and refers to the nth symbol of P; **$0** is the parent (LHS) symbol and **$1** is the first symbol of the RHS. A nonlocal reference is either a nonterminal symbol name (referring to the closest ancestor of that name), or a pair of symbol names separated with a slash (referring to a particular child of that ancestor).

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 built up from attribute references and constants using arithmetic, logical and relational operators written as in C language [10].

C. Simple Productions

A simple production (the unique production of its LHS symbol) is written as:

$$<x> \quad : \quad \alpha \; .$$

;

where $<x>$ is a nonterminal and α is a string of grammar symbols.

$P_{<x>}$ is the composition of the processes in α (possibly modified by *attribute assignments* or an *enabling condition*, see below). α may be empty, in which case $P_{<x>}$ is immediately finished.

By default, process composition is sequential; each subprocess in α is started when its left neighbor finishes. Processes may also be composed in parallel, allowing the

events they derive to be interleaved in time. This is done by enclosing a RHS in curly brackets; its processes are then executed in parallel, and the parent process finishes when all subprocesses are finished (see Section III for examples).

D. Attribute Assignments

A production can have *attribute assignments* of the form **attribute-reference = expression**. Attribute assignments are used for synchronization or data transfer between process instances.

When a process starts, its subprocesses are instantiated and their attributes are initially undefined. In general, attribute assignments are performed before starting the first subprocess, and are performed in order. There are two special cases. 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 $P_{/\text{timer}/}$ 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 symbol, $P_{<x>}$ 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.

E. 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**. 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 [$N < - DT$].

F. Alternative Productions

A nonterminal $< X >$ may have several *alternative* productions, denoted as follows:

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

$P_{<X>}$ blocks until some alternative becomes *selected*, at which point that process definition is executed and the other alternatives are discarded. An alternative *P* is said to be *selected* when either

- an input event [*y*] occurs such that *P* yields, without blocking, an instance of [*y*] which accepts the message. In particular, the enabling conditions of the productions in the sequence by which [*y*] is derived from $< X >$ must all be satisfied
- *P* yields, without blocking, the epsilon process, an output symbol, or **/remove/**
- *P* is defined by a production of the form

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

and an instance of *P*, started when $< X >$ is started, finishes its **/timer/** subprocess. As an example, consider the following:

```
<get ack>  :  [N->AK].
              |  .
              if <connection>.closed
              |  /timer/ <timed out>.
              $1.interval = 10
;
```

The semantics of $P_{<\text{get ack}>}$ 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. The selection depends on what happens first, in terms of input events, attribute value changes, and the passage of real time.

G. Externally Defined Functions

External function names can be used in attribute assignments and enabling conditions. They typically perform calculations that are not easily expressed within RTAG or that are installation-dependent. The function definitions are not part of the RTAG specification, and must in general be supplied for each implementation.

H. Multiple Derivation of Input Symbols

When an input event occurs, there may be many processes able to accept it. RTAG supports *broadcast* semantics: a message is accepted by all processes that can accept it. This is useful because, in a complex protocol, a single input message may be relevant to several subprotocol processes. For example, in our specification of

TP-4, the acknowledged delivery of each packet is handled in a separate process. An acknowledgment event may serve to acknowledge several packets, and hence be relevant to several processes.

If an input message is not accepted by any process, it is discarded and ignored. In some applications it might be preferable to regard unaccepted messages as fatal errors or to log them for debugging purposes.

I. Key Attributes

Some protocols have a "reference number" mechanism for associating input messages with connections or transactions. While this could be enforced in RTAG at the leaf level by having a reference number equality clause in each enabling condition, this would be cumbersome and inefficient. Instead, RTAG has the following mechanism: an attribute name can be declared as being the *key* attribute. No two nonterminal process instances may simultaneously have the same value for the key attribute. Suppose an input symbol instance X has the key attribute with value k. If there is a nonterminal instance Y with key attribute value k, X can be accepted only by descendants of Y. Otherwise, X can be accepted only by processes of which no ancestor has the key attribute. If X does not have the key attribute then there are no restrictions.

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

III. An Example: The TP-4 Transport Protocol

As an example and test case for RTAG, we have written a specification of the NBS class 4 transport protocol (TP-4), working from an AFSM specification of the protocol [22]. The entire specification is given in [1]; in this section, we give the overall structure of the specification, and describe a few parts in detail.

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 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), acknowledgment (AK), and graceful close (GR). Data messages passed between TP-4 and upper layer clients are called *transport service data units* (TSDU's).

A. Design Principles

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

1) Logically distinct parts of the protocol, or *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.

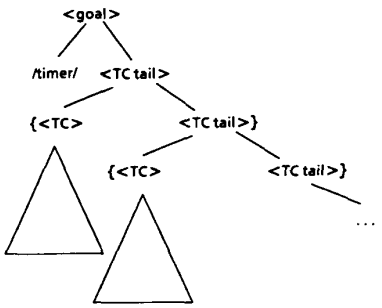

Fig. 1. Multiple Transport Connections.

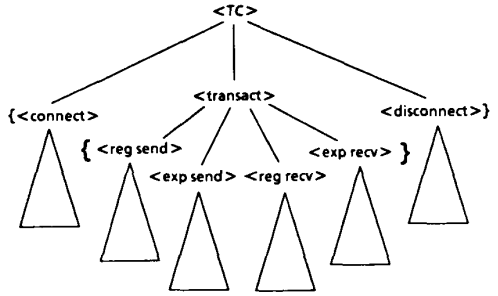

Fig. 2. Subprotocols Within a Connection.

B. Multiple Transport Connections

The following productions provide an initial inactive period which protects the integrity of connections, and allow many transport connections to exist concurrently:

```
<goal>    :  /timer/ <TC tail>.
             $1.interval = QUIET_TIME
;
<TC tail> :  {<TC> <TC tail>}.
          |  [U->FIN].
;
```

This produces the parse tree structure shown in Fig. 1. Each connection (i.e., < TC > process) consists of three concurrent subprotocols: connection establishment, transaction, and disconnection (see Fig. 2). The attributes of <TC> contain information, such as local and foreign addresses and reference numbers, needed by more than one of these subprotocols.

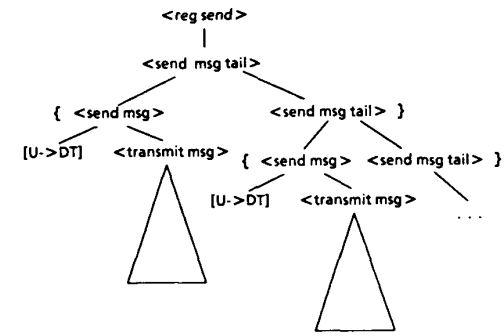

Fig. 3. TSDU Sending.

```
<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
;
```

C. The Regular Send Subprotocol

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

For sending regular-priority data, TP-4 uses a sliding window whose size cannot exceed the receive window size (or "credit") supplied, in acknowledgment messages, by the peer. The regular send subprotocol is divided into subprotocols for sending data and for handling acknowledgments. The send-data subprotocol is further divided into subprotocols for each TSDU (**<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.

Fig. 3 shows the queueing and transmission of TSDU's. The delivery of each TSDU involves a recursive process **<send packet tail>** that splits the TSDU's data into packets and invokes a **<send packet>** process to handle the delivery of each one. The subprotocol for delivering a TSDU is defined below; **extract** splits off a packet-sized portion of a longer message, and **eot** returns true iff its argument 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) % MAXSEQ
                   $2.eot = $1.eot
                   |       /freedata/.
                   if $0.eot
                   $1.data = <transmit msg>.data
                   <seng 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: **between**(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).

```
<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 between (<send packet>.seqno, $1.seqno, <reg send>.next seq)
                   $2.refno = <TC>.refno
                   $2.data = <send packet>.data
                   |   /timer/.
                   if $0.count == 0
                   $1.interval = GIVEUP_TIME
                   <TC>.transerror = true
;
```

Fig. 4 shows the delivery of packets within a TSDU.

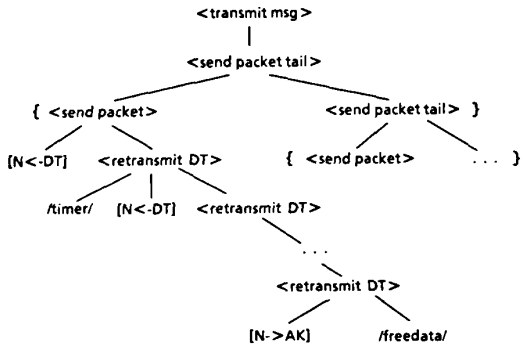

Fig. 4. Packet Sending.

IV. Design of an RTAG Parser

An RTAG parser is a program that, given the RTAG specification of a protocol, implements the protocol. It does so by maintaining a parse tree of attributed symbol instances, and responding to input events by attempting to "derive" them by applying productions to leaf nonterminal symbols. In this section we sketch the algorithm used by an RTAG parser.

A. Definitions

Some productions may be applied for reasons other than message arrival. A production P is *immediate* if either

- P is the only production of its parent symbol and the RHS starts with a nonterminal;
- P is part of a production sequence which left-derives epsilon, an output symbol, or /**remove**/;
- the RHS of P starts with /**timer**/.

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 RHS, when the parent symbol is expanded).

Symbol status can be mapped 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.

B. Algorithms

The parsing algorithms use the following data structures.

- The *newly active queue* is a queue of symbol instances awaiting processing after initially becoming active.
- The *immediate queue* is a queue of active nonterminal instances that may be eligible for expansion by an immediate production. If the enabling condition of an immediate production of a symbol S depends an attribute A, then S is added to this queue whenever the value of A changes.

The parser is initialized by creating an instance of the goal symbol to serve as the parse tree root, and setting all queues to empty. Its actions, thereafter, are initiated by input events and timeouts.

1) Processing Input Events: The steps in processing an input event are:

- Compute a candidate set of nonterminal leaf symbols that might yield the input symbol. 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.
- Attempt to yield the input symbol from each candidate $< a >$ in turn. This is done by going through all the production sequences that left-derive $[x]$ from $< a >$ in the underlying CFG. The first production sequence that succeeds (yields the input symbol without blocking) is used. As a side-effect of applying productions, new active symbols may be created, and attribute value changes may enable immediate productions.
- Process the symbols in the newly active and immediate queues.

Whenever an attribute value is changed, it is possible that the value of the enabling condition of some immediate production becomes true. Therefore, whenever such a change occurs, all active symbols with an immediate production whose enabling condition involves the attribute are added to the immediate queue.

2) Processing Timeouts: A timer is started when a symbol having a timed production becomes active and is not immediately expanded. A timeout is processed by applying the production to the symbol instance. As with input events, this can result in additions to the newly active and immediate queues, that must then be processed.

3) Processing the Immediate and Newly Active Queues: 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.

A symbol on the immediate queue is processed by testing the enabling conditions of its immediate productions. If the condition of a production whose RHS starts with /**timer**/ is satisfied, a timer is started. If the condition of a nontimed production is satisfied, the production is immediately applied.

A newly active symbol is processed as follows:

- If X is an output symbol, the corresponding output routine is called; the attribute values of X are passed as arguments.
- If X is /**remove**/, the symbols and associated storage below the symbol instance Y pointed to by /**remove**/ *where* are removed. Y is left in the tree but marked as finished.
- If X is a nonterminal and has an enabled immediate production, this production is applied. If there are immediate productions but none are enabled, links are established between the attributes on which the conditions of the productions depend, and X. Timers are started for any enabled timed productions.

C. Implementation Alternatives

The algorithm sketched above can be implemented in a variety of ways. Two actual implementations exist. The first was done with little concern for efficiency. It is interpretive, even at the level of expression evaluation. Measurements of its performance [1] show that it uses 20 to 30 times the CPU of hand-coded counterparts. While this level of performance is sufficient to demonstrate feasibility, it is not acceptable for production use.

The second implementation improves performance at the cost of a larger machine-dependent part. The goals of the second implementation are to reduce total CPU requirements, and to minimize the time between message arrival and response. The following techniques are used.

- Functions for enabling conditions and attribute assignments are translated into a native language of the target machine.
- Nonlocal symbol references are evaluated ahead of time. Pointers to all ancestors of a nonterminal X that could be made by descendants of X are stored with X, and propagated to its children as needed.
- The candidate sets for all input symbol types are calculated ahead of time. When a nonterminal symbol is activated and is not immediately expanded, it is added to the candidate sets of any input symbols that it left-derives.

Performance studies of the new parser are underway. A preliminary measurement shows that for the alternating bit protocol shown in Section II-J, the parser uses about 2 ms on a VAX® 11/785 to process an acknowledgment, cancel a timer, send a new data packet, and start a new timer. This does not include the allocation and checksumming of packets or the generation of headers.

V. RTAG-Based Software Systems

As indicated earlier, a primary goal of RTAG is to provide a software system for protocol development and experimentation. A possible use for such a system is to develop protocols in an environment that is heterogeneous with respect to processor types or operating systems. Therefore, we distinguish between the *host* system on which the specification is being developed, and the *target* systems on which the protocol is to run. It is natural to break up the work between host and target systems as follows.

- An RTAG *compiler* runs on the host system. It checks the RTAG specification for errors, parses it, and converts it into a machine-independent form from which the data structures needed by the target machine can be easily derived. An RTAG compiler for UNIX was constructed using Lex [13], Yacc [8], and it uses the C preprocessor [23] to handle comments, macro substitutions, and include files.
- An RTAG parser must be implemented for each target system. The two versions of the RTAG parser described in Section IV-C have been written in C. The first version used the compiler's output file directly. The second version adds a preprocessing step in which portions of the specification are converted to C code (this is straightforward since RTAG uses C expression syntax). In either case, certain aspects are system dependent, such as memory allocation and the access to timers.

®VAX is a registered trademark of Digital Equipment Corporation.

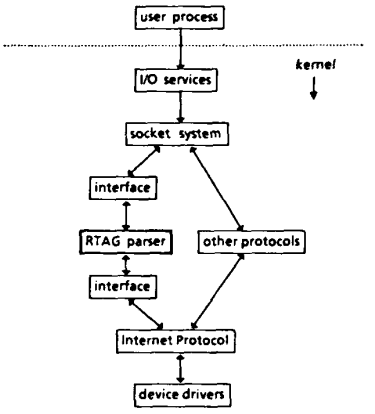

Fig. 5. An RTAG Software System.

The relationship between the components of the RTAG software system is shown in Fig. 5.

A. RTAG Environments in BSD UNIX

The RTAG parsers can run in the following execution environments in 4.3 BSD UNIX:

- *Kernel Mode:* In 4.2 BSD and later versions of Berkeley UNIX, the kernel is designed to facilitate the addition of new communication protocols [23]. Protocols are interfaced to user-level processes by the "socket system," which provides services such as process synchronization, data buffer management, and connection queueing. The interfaces between 1) protocols and the socket system, 2) different protocol layers, and 3) protocols and network interfaces drivers, are standardized to some extent. The position of the RTAG parser in this setting is shown in Fig. 6.
- *User-Level Kernel Simulation:* To allow debugging kernel-level protocols at the user level, we simulated parts of the kernel in a user program. 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). We also developed routines for interactive perusal of the parse tree, and debugging options that allow logging the actions of the parser (productions, event occurrences, and attribute assignments) in a disk file.
- *User-Level Prototyping 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 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.

B. Interface Routines for TP-4 under 4.3 BSD UNIX

Our RTAG specification of TP-4 was combined with the kernel-mode RTAG parser to obtain a working version

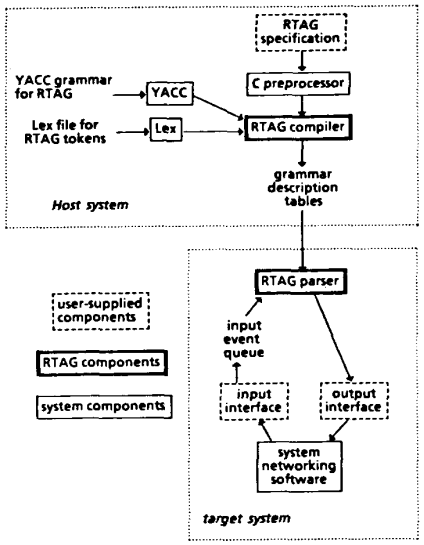

Fig. 6. RTAG in the UNIX Kernel Environment.

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

VI. Related Work

We first compare RTAG to other systems for protocol development. These systems may be categorized as follows.

- Systems such as IBM's FAPL system [15] and that described by Blumer and Tenney [2] are intended primarily to automate the development of production-quality implementations. These systems use FSM-based formalisms augmented with HLL code. The efficiency of their output is approximately that of direct implementations, and RTAG has not yet attained this goal. However, RTAG has some advantages over FSM as a specification language (see Section VII).
- Systems such as PANDORA [7] and CUPID [21] are intended primarily for automated analysis and verification. RTAG currently has no analysis or verification facility, and has no theory to support such a facility. This has not been a goal of the project, and our experience is that it is possible to develop error-free protocols using RTAG without automated analysis tools.
- LOTOS [3], [4], is an FDT based on algebraic event expressions. Researchers have also used logic programming languages [14], [18] to specify protocols. These formalisms have features, analogous to those of RTAG, allowing them to support well-structured specifications. Interpreters have been developed for both formalisms, allowing (in theory) automated implementation. However, it is not clear that these implementations can be made efficient enough for production use.

A second area of comparison is with work in programming languages and environments. Attribute grammars have been used to specify programming languages [11], and to aid in the automatic generation of compilers and language-based editors [6]. There are some ideas in common between these areas and RTAG; for example, references to attributes of nonlocal symbols [9]. However, the similarity is almost entirely superficial. Because of the completely different semantics, none of the language work appears to carry over to RTAG. Among these semantic differences:

- The "semantic equations" in language grammars correspond roughly to RTAG's attribute assignments. However, attribute assignments are not equations; they are performed once (when the production is applied) and need not hold after that.
- In RTAG, attribute information is used (in enabling conditions) to control parsing, whereas in language systems attributes are generally evaluated after parsing to check for errors.
- RTAG's subtree concurrency and multiple derivation of input symbols appears to preclude application of parsing techniques for context-free grammars.

VII. Conclusion

Nounou and Yemini [16] state that

> State-transition-based specifications (such as state machines and Petri-net-based models) lend themselves more easily than event-based specifications (such as sequence expressions) to translations into implementations. This is because specifications for the former describe the flow of execution of a protocol step by step, while specifications for the latter are concerned with the valid outcomes of the protocol operation and not with how the outcomes are produced.

Piatkowski [17] repeats this argument, and concludes that the goal of event-based techniques are "socially irresponsible" for this reason. The conclusions of the RTAG project, in contrast, are that:

- Event-based specifications (such as grammars) can, in general, be translated into implementations as easily as state-based specifications. A regular grammar has a trivial translation into an FSM. There are very efficient parsing techniques for large classes of context-free grammars. RTAG is more complex than either of these, but its semantics were intentionally designed to allow efficient real-time parsing. Preliminary results (see Section IV-C) suggest that RTAG implementations can be fast enough for production use.
- Because event-based specifications express "the valid

outcomes of the protocol operation," they are easier to understand.

- The limited expressive power of FSM-based formalisms is most evident when they are applied to protocols that consist of a dynamically varying set of concurrent "subprotocols." RTAG, because it allows a direct specification of such structures, is better-suited to these protocols. For example, the RTAG specification of TP-4 is significantly more complete, concise, and well structured than its FSM counterpart.

Work remains to be done towards increasing and measuring the efficiency of RTAG-based implementations, and towards the development of RTAG-based design tools.

Acknowledgment

I would like to thank A. Bricker and T. Lebeck, who helped with the UNIX kernel implementation and provided most of the code for the TP-4 interface routines and the UNIX kernel simulation. L. Chin and D. Hernek assisted in the implementation of the second version of the RTAG parser.

References

[1] D. P. Anderson, "A grammar-based methodology for protocol specification and implementation," Ph.D. dissertation, Univ. Wisconsin—Madison, WI, Aug. 1985.

[2] T. P. Blumer and R. L. Tenney, "A formal specification technique and implementation method for protocols," *Comput. Networks*, vol. 6, no. 3, pp. 201–217, July 1982.

[3] J. P. Briand, M. C. Fehri, L. Logrippo, and A. Obaid, "Structure of a LOTOS interpreter," *SIGCOMM'86 Symp.*, Stowe, VT, Aug. 1986.

[4] E. Brinksma, "A tutorial on LOTOS," in *Proc. 5th IFIP Symp. Protocol Spec. Test. Verif.*, June 1985.

[5] D. Clark, "Modularity and efficiency in protocol implementation," *RFC 817, SRI Network Information Center*, 1983.

[6] A. T. Demers, T. Reps, and T. Teitelbaum, "Incremental evaluation for attribute grammars with applications to syntax-directed editors," in *Proc. Eighth Annu. ACM Symp. Principles Program. Languages*, pp. 105–116, 1981.

[7] G. Holtzmann, "The PANDORA system: An interactive system for the design of data communication protocols," Delft Univ. Technol., Rep. 39, Aug. 1982.

[8] S. C. Johnson, "Yacc: Yet another compiler-compiler," *Unix Programmer's Manual*, vol. 2B, 1979.

[9] G. F. Johnson and C. N. Fischer, "A meta-language and system for nonlocal incremental attribute evaluation in language-based editors," in *Proc. 12th ACM Symp. Principles Program. Languages*, pp. 141–151, Jan. 1985.

[10] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[11] D. E. Knuth, "Semantics of context free languages," *Math Syst. Theory J. 2*, vol. 2, pp. 127–145, 1968.

[12] S. J. Leffler, W. N. Joy and R. S. Fabry, *4.2BSD networking implementation notes, UNIX programmer's manual, vol. 2C*," Computer Systems Research Group, Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, Aug. 1983.

[13] M. E. Lesk and E. Schmidt, "Lex—a lexical analyzer generator," *Unix Programmer's Manual, vol. 2B*, 1979.

[14] L. Logrippo, D. Simon, and H. Ural, "Executable description of the OSI transport service in Prolog," in *Proc. 4th IFIP Workshop Protocol Spec. Test. Verif.*, pp. 279–294, June 1984.

[15] S. C. Nash, "Automated implementation of SNA communication protocols," in *Proc. IEEE Int. Conf. Commun.*, pp. 1316–1322, June 1983.

[16] N. Nounou and Y. Yemini, "Development tools for communication protocols: An overview," in *Proc. IEEE Global Telecommun. Conf.*, Nov. 1984.

[17] T. F. Piatkowski, "Protocol engineering," in *Proc. IEEE Int. Conf. Commun.*, Boston, MA, June 1983.

[18] D. P. Sidhu, "Protocol verification via executable logic specifications," *IFIP Workshop Protocol Spec. Test. Verif.*, 1983.

[19] C. A. Sunshine, "Formal techniques for protocol specification and verification," *Computer*, vol. 12, pp. 20–27, 1979.

[20] C. A. Vissers, R. L. Tenney, and G. Bochmann, "Formal description techniques," *Proc. IEEE 71*, pp. 1356–1364, Dec. 1983.

[21] Y. Yemini and N. Nounou, "CUPID: a protocol development environment," in *Proc. 3rd IFIP Workshop Protocol Spec. Test. Verif.*, May 1983.

[22] "Specification of a transport protocol for computer communications, volume 3: Class 4 protocol," Nat. Bureau Stand., Rep. ICST/HLNP-83-3, 1983.

[23] *4.2BSD UNIX Programmer's Manual*, Comput. Syst. Res. Group, Dep. Elec. Eng. Comput. Sci., Univ. California, Berkeley, Aug. 1983.

[24] "Basic reference model for Open Systems Interconnection," International Standards Organization 7498, 1983.

David P. Anderson received the B.A. degree in mathematics from Wesleyan University, Middletown, CT, in 1977, the M.A. degree in mathematics, and the M.S. and Ph.D. degrees in computer science from the University of Wisconsin—Madison in 1979, 1983, and 1985, respectively.

Since 1985, he has been an Assistant Professor with the Computer Science Division, Department of Electrical Engineering and Computer Science, at the University of California, Berkeley. His fields of research are operating systems, distributed systems, and computer music.

Dr. Anderson is a member of the Association for Computing Machinery and the IEEE Computer Society.