A GRAMMAR BASED METHODOLOGY FOR PROTOCOL SPECIFICATION AND IMPLEMENTATION

by

David P. Anderson
Lawrence H. Landweber

Computer Sciences Technical Report #608

July 1985

A GRAMMAR-BASED METHODOLOGY FOR PROTOCOL SPECIFICATION AND IMPLEMENTATION

David P. Anderson
Lawrence H. Landweber

Computer Sciences Department
University of Wisconsin - Madison
Madison, Wisconsin

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 ISO 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 an RTAG-based TP-4 implementation in the DoD Internet domain.

1. INTRODUCTION

Data communication protocols serve to enhance the utility and reliability of communications media. In recent years a great deal of effort has been devoted to understanding the services and mechanisms of such protocols. The ISO, in cooperation with other international organizations is currently in the process of developing protocols corresponding to the layers of its reference model. Earlier work by the U.S. DoD as well as by various computer manufacturers (e.g., IBM's SNA and DEC's DECNET) has resulted in the specification of protocol architectures which are functionally similar to but incompatible with each other and with the proposed ISO protocols.

The need for machine-independent specifications of standard communication protocols has encouraged the definition of a number of formalisms for this purpose. In this paper, we define a new methodology which is rich enough to concisely describe real protocols, is "easy and intuitive" to use, and which we believe can eventually serve as the basis of a software system for automated generation of quality protocol implementations.

Our methodology, "Real-Time Asynchronous Grammars" (RTAG), is based on an extension of attribute grammars in which terminal symbols correspond to messages sent and received. It provides formal constructs for specifying concurrent protocol activities and real-time constraints. RTAG facilitates the expression of most protocol mechanisms, and encourages a top-down approach to designing and specifying protocols.

We have developed a software system consisting of a *grammar analyzer* and an *RTAG parser*. Applied to an RTAG specification of a protocol, these programs provide a major part of an implementation of the protocol (essentially only operating system interface routines and utility routines for functions such as packet assembly need be added).

The protocol implementor writes an RTAG description of a protocol. This description is then processed by the grammar analyzer, which produces a set of "grammar description tables". The grammar description tables for a protocol, plus user supplied interface and utility routines for a particular operating system, are combined with the RTAG parser to obtain an implementation of the

protocol. The parser operates by doing a top-down parse on the grammar in response to external events (e.g., arrival of a packet or a request from a user).

If a number of protocols have been specified by RTAG, and implementations of these protocols are desired for a new computer system, it suffices to implement the RTAG parser and the utility routines on the system. Conversely, experimentation with a protocol running on a heterogeneous network of systems all running the RTAG parser can be done by changing the RTAG specification rather than by rewriting many hand-coded protocol implementations.

We have implemented an RTAG parser under 4.2 BSD UNIX and have written an RTAG specification for the ISO Class 4 Transport Protocol (TP-4). The RTAG parser has been installed in the UNIX kernel and interfaced with the other components of the 4.2 BSD UNIX networking system, thus obtaining an RTAG-based implementation of TP-4. The goal of this component of our work is to study the feasibility of generating production implementations in the manner described above.

Section 2 of this paper provides an overview of related work in protocol specification techniques. The remainder of the paper covers various aspects of RTAG. Sections 3 and 4 describe the syntax and informal semantics of RTAG. Section 5 describes the structure of a simple RTAG parser, and Section 6 discusses RTAG software tools and their integration into environments such as operating systems and prototyping/debugging systems. Section 7 summarizes RTAG, contrasts RTAG with related work, and suggests directions for further work.

2. FORMALISMS FOR PROTOCOL SPECIFICATION

The general goal of a protocol specification is to describe the desired behavior in abstract terms, i.e., without dependence on a particular programming paradigm. Besides their use in protocol verification and performance study, specifications aid in implementation by providing formal or informal guidelines for programmers and/or by allowing some degree of automated implementation. The remainder of this section briefly surveys some existing protocol specification formalisms.

Protocols are often modeled by associating a finite-state automaton with each entity. This approach allows verification by various state-space exploration techniques. Pure FSA cannot easily model data-dependent aspects of protocols, such as the use of large sequence numbers spaces. To deal with this an extension called "augmented FSA" (AFSM) or "abstract machines" (see Sun81) has been defined.

Blumer and Tenney (Blu82) describe the use of AFSM in a project whose goals include automatic implementation and verification. Their paper describes software tools for converting an AFSM description to a C program which implements the protocol. A later paper by Sidhu and Blumer (Sid83) describes tools for automated testing of protocols described by AFSM's; the connection establishment phase of TP-4 is used as an example.

An automated implementation project undertaken by IBM in connection with their Systems Network Architecture (SNA) is described in several papers (Sch80, Poz82, Nas83). It is based on the Format and Protocol Language (FAPL) and has been used to develop a specification of SNA from which a partial implementations can be automatically generated. FAPL is based on a subset of PL/1 and provides AFSM's as a programming abstraction.

Harangozo (Har78) uses a grammar-based formalism to specify a portion of the HDLC data-link protocol. An attribute-like mechanism, in which symbols and productions are indexed by integers, is used to represent sequence numbers. The grammars used are all regular grammars and hence are equivalent to FSA's. Teng and Liu (Ten78) use context-free grammars to specify some simple protocols. Their objectives include implementation and limited verification.

Yemeni and Nounou (Yem83) discuss a "protocol development environment" in which several formalisms (FSA, Petri nets, high-level language) are available for protocol specification. They propose tools to convert these forms into a "canonical semantic model" based on Milner's Calculus of Communicating Systems (CCS) (Mil80).

CCS also serves as the basis for LOTOS (Bri85), a protocol specification language intended for use in specifying "real" protocols. LOTOS is similar to RTAG in that it allows complex protocols to

be broken into processes which can be composed in different ways. RTAG and LOTOS differ in that LOTOS is oriented towards verification (and has a better-understood semantic model), but its utility in generating implementations has not been shown.

3. OVERVIEW OF RTAG

Real-time asynchronous grammars (RTAG) is a protocol specification language oriented towards specifying and implementing complex protocols. An RTAG specification is based on an underlying context-free grammar. Most terminal symbols correspond to an *event*, that is, to the sending or receipt of a particular message. Event symbols are called *input* or *output* symbols depending on the direction of the message relative to the protocol entity being specified.

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

<goal> : [net-data] [user-data] <goal>.

might be part of a trivial protocol which responds to messages from the network layer (the **[net-data]** input event) by relaying them to an upper layer (the **[user-data]** output event).

Each RTAG symbol has an associated set of attributes. The attributes of an input or output symbol correspond to the fields of the associated message or packet. A production may have a boolean *enabling condition*, involving attributes of its symbols. This enabling condition must evaluate to **True** for the production to be applied. Each production can also have an associated set of *attribute assignments* which are performed when the production is applied.

Each RTAG symbol can be thought of as representing a "subprotocol". Productions can combine subprotocols in sequence:

<x> : <y> <z>.

or, using curly brackets, in parallel:

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

In the first case, the events derived from **<y>** must precede (in real time) the events derived from

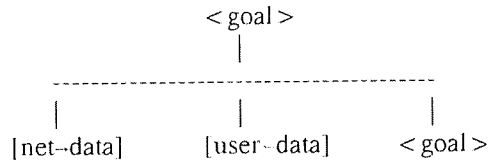
<z>, whereas in the second case the events may be interleaved.

Real-time constraints, such as timeouts, can be represented using a special terminal symbol **/timer/**, whose single attribute is the length of a real-time interval in clock ticks.

The semantics of RTAG can be explained in terms of an *RTAG parser* which maintains the current parse tree. When an input event occurs, the parser attempts to left-derive the corresponding input event symbol from some nonterminal symbol which is a leaf in the parse tree. For example, suppose the grammar contains the production

<goal> : [net-data] [user-data] <goal>.

If the current parse tree is



and the **[net-data]** event occurs, the above production would be applied to the lower instance of **<goal>**. If the production has attribute assignments, these would result in input data received from the network being introduced into the parse tree. As a side-effect of applying the production, the output event symbolized by **[user-data]** would be performed.

If an input symbol cannot be derived, it is discarded and ignored. This is appropriate, for example, with a duplicate connection request or confirm packet, where such an occurrence does not signify a protocol error. In other cases it might be preferable to regard unparsed packets as fatal errors and to abort the protocol, or to log them for debugging purposes.

4. RTAG SYNTAX AND INFORMAL SEMANTICS

Because of their role in automated implementation, RTAG grammars are written as machine-readable files obeying syntactic rules. This section develops in parallel the syntax and informal semantics of RTAG. An RTAG specification consists of two parts: a list of symbol definitions and a list of productions.

4.1. Symbols, Attributes, and Names

Each symbol has an associated set of attributes, each of which has a type (e.g. integer, boolean, pointer to data). Symbol names obey the following rules:

(1) Nonterminal symbols are delimited by angle brackets.

(2) Event symbols (both input and output) are delimited by 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 specification **[U-CR]** is a connection request input from the user layer, and **[N-DT]** is a data packet output to the network layer.

(3) Special symbols are delimited with slashes. There are currently three special symbols: **/timer/**, **/remove/**, and **/freedata/**.

Here is an example symbol declaration section:

```
goal      <connection>

nonterm <send>
          int       seqno

input     [U-CR]
          int       refno
          int       address

output    [N-DT]
          int       refno
          dataptr   data
```

4.2. Productions

The syntax for simple productions with a common LHS symbol is:

```
<x>     :         S1.
          (optional enabling condition)
          (optional attribute assignments)
          |         S2.
          (optional enabling condition)
          (optional attribute assignments)
          ...
;
```

where **<x>** is any nonterminal and S1, S2, ... are strings of symbols. To simplify the parsing task, two restrictions are imposed: 1) input symbols must be leftmost in a RHS, and 2) left recursion is not allowed.

Each symbol instance is initially "inactive" and is later "activated". A nonterminal cannot be expanded if it is inactive, and the output event represented by an output symbol is performed only when its symbol is activated. When a production is applied, the leftmost symbol of the RHS is activated, and each remaining RHS symbol is activated as soon as all its left siblings have been fully expanded. This rule connects left-to-right syntactic order with real-time order. For example, in a production of the form

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

all of the terminal symbols derived from **<y>** will strictly precede (in real time) all those derived from **<z>.** This corresponds to sequential composition of subprotocols. The ability to compose subprotocols in parallel is also needed: for example, the send and receive portions of a transport protocol can be specified separately but must be interleaved in real time. RTAG uses a curly bracket notation to express this parallel composition. A group of RHS symbols surrounded by curly brackets will all be activated as soon as the leftmost symbol in the group is activated. For example, in

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

<a> and **<b>** will be activated when the production is applied, and their subtrees will be expanded concurrently. When they are both fully expanded **<c>** will be activated, and when it is fully expanded **<d>** and **<e>** will be activated.

4.3. Symbol and Attribute References

Symbol references can be *local* or *non-local*. A local reference is of the form $n, and refers to the nth symbol of the production; $0 is the LHS and $1 is the leftmost symbol of the RHS. A nonlocal reference is written as **sym1/sym2.../symn.** The semantics are as follows: find the closest

ancestor named *sym*1 of the symbol instance serving as the LHS of this production; find the leftmost of its children named *sym*2; the leftmost of that symbol's children named *sym*3 and so forth. A reference to a nonexistent symbol (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 symbol instance specified by *symbol-reference*.

4.4. Expressions, Enabling Conditions, and Attribute Assignments

Expressions are formed from constants, attribute references, integer operators (+, −, mod), logical and relational operators (written as in C), and external functions (see sec. 4.7.1).

A production can have an "enabling condition", a boolean-valued expression that must evaluate to **True** for the production to be performed. For example,

```
<x>  :  [N-DT] <z>.
        if ($0.size > 0) && ($1.number == $0.number)
;
```

specifies that the production can be applied only if the *size* attribute of **<x>** is positive and the *number* attributes of **<x>** and **[N-DT]** are equal.

A production can have zero or more "attribute assignments" of the form *attribute-reference* = *expression*, which are used to transfer information between symbols. When a production is applied, attributes of newly-created symbol instances (except input symbols) are initially undefined. Assignments are then performed (meaning that the RHS expression is evaluated and the value is stored in the LHS attribute) in the order in which they appear in the grammar. *Persistent* attribute assignments (denoted with =* instead of =) are performed whenever a value on which the RHS depends is changed. For example,

```
<x>  :  [N-DT] <y>.
        $0.data = $1.data
        $2.ready =* $0.ready
```

specifies that the *data* attribute is to be copied from **[N-DT]** to **<x>** when is applied, and the *ready* attribute is to be copied from **<x>** to **<y>** both when the production is applied and whenever

thereafter <x>.ready is modified; some other enabling condition or attribute assignment may depend on <y>.ready.

4.5. Special Symbols

Special symbols are terminal symbols which represent internal actions of the parser. There are currently three special symbols: **/timer/**, **/remove/**, and **/freedata/**.

The **/timer/** symbol, which has a single integer attribute *interval*, represents the passage of that many ticks of real time. Productions in which **/timer/** is the leftmost symbol of the RHS are called *timed productions*. As an example,

```
<x>  :  /timer/ [N-AK].
        $1.interval = 10
     |  [U-DT] [N-DT].
;
```

specifies that if the **[U-DT]** event occurs within ten ticks of the time an instance of <x> is activated then the second production will be applied to <x>, and **[N-DT]** will be performed; otherwise the first production will be applied ten ticks after <x> is activated, and **[N-AK]** will be performed as a result. This is accomplished by having the parser start a timer when <x> is activated. It cancels the timer and performs the second production if **[U-DT]** occurs before the timeout, otherwise it performs the first production when the timer times out.

The **/timer/** mechanism contributes greatly to the convenience of RTAG for expressing timing constraints because it removes the problem of timer names and the necessity of explicitly starting and canceling timers.

The **/remove/** symbol has an attribute *where* of type symbolptr. When an instance of **/remove/** is activated, the subtree rooted by the symbol instance pointed to by this attribute is removed from the parse tree, and the symbol is considered to be fully expanded. This is used, for example, in handling abrupt closure of connections.

The **/freedata/** symbol has an attribute *data* of type dataptr. When an instance of **/freedata/** is activated, the memory area pointed to by this attribute is freed. This is needed, for example, to

specify memory management in protocols that do retransmission.

4.6. External Functions

External functions are used to perform calculations which are not easily expressed within RTAG or which are installation-dependent. These are notated as #name#(expr-list). For example, our TP specification uses external functions for computing the cyclic "between" relation on sequence numbers, and to check the acceptability of a connection request or confirmation.

4.7. Key Attributes

The work done in processing an input event consists largely of finding where to parse it. Most protocols which handle multiple connections have a "reference number" mechanism for associating input events with connections. Rather than enforce this at the leaf level (by having a reference number equality clause in the enabling condition) we have added a mechanism which allows the parser to search only the subtree of the correct connection.

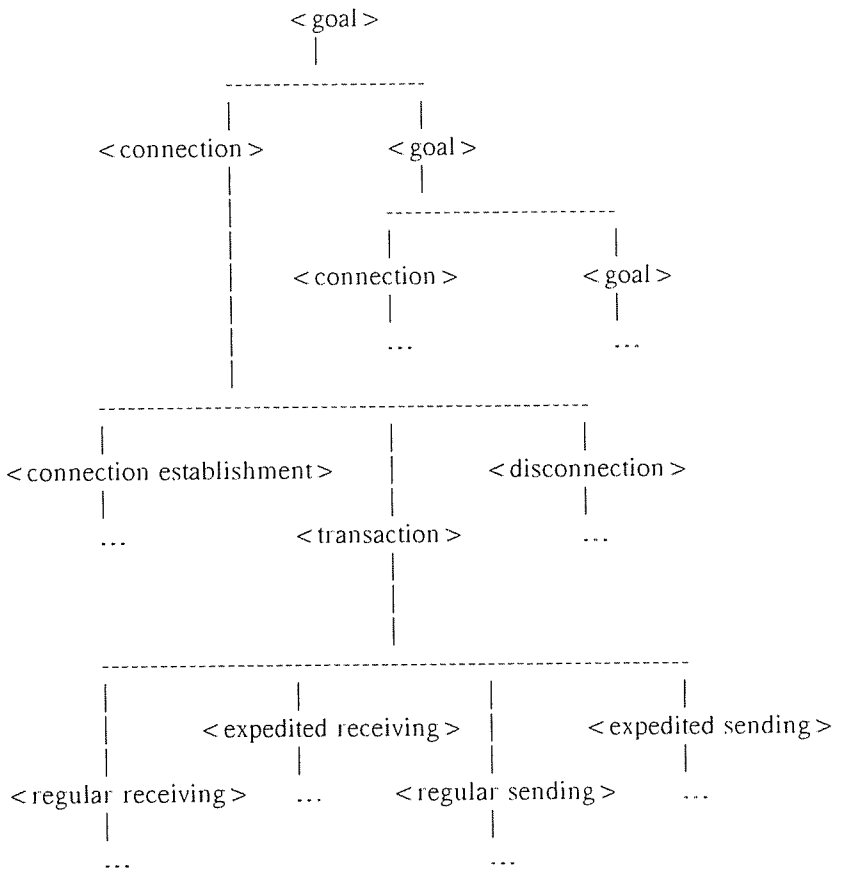
This is done as follows: an attribute name can be designated as *key*. If an input symbol has the key attribute, it can be derived at the end of particular branch only if there is a symbol on the branch having the key attribute, with the same value as that of the input symbol. A second attribute name can be designated as *key−intro*; this is used by input symbols such as connection requests which introduce new key attribute values. If an input symbol has the key-intro attribute, and some nonterminal has the key attribute with the same value, then the input symbol can only be derived below such a nonterminal.

4.8. Parsing at Multiple Places

A single input event may be relevant to several subprotocols. For example, in our specification of the sending component of TP, each packet corresponds to a different subtree, and a single acknowledgement event may apply to several outstanding packets. Therefore, RTAG allows a single input event to be derived at several places in the parse tree.

4.9. Example: an RTAG Specification of TP-4

An RTAG specification has been written for the ISO level 4 transport protocol (TP-4), using the following guidelines: 1) The protocol is recursively divided into logically distinct subprotocols, and each subprotocol is placed in a subtree of the parse tree. 2) Attributes, which are used for reference numbers, sequence numbers, window sizes, etc., are moved to the root of the appropriate subtree. The following diagram shows the top-level decomposition of the protocol.



Appendix A contains an RTAG specification of the "regular sending" subprotocol.

5. AN RTAG PARSING ALGORITHM

This section outlines an algorithm for an RTAG parser, i.e., a real-time program which, given an RTAG specification, processes input messages, handles timers, and generates output events in such a way that the protocol defined by the grammar is obeyed. Our design emphasizes simplicity

and portability rather than efficiency.

5.1. Static Data Structures

The parser uses the following static structures, which depend on the RTAG grammar being parsed, 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 a list of nonterminals that derive X in the underlying CFG, a list of nonterminals that left-derive X in the underlying CFG, and for each nonterminal Y which left-derives X, a list of the production sequences by which X can be left-derived from Y in the underlying CFG. Because left recursion is not allowed in the underlying CFG, the length of such a production sequence is bounded by the number of symbols, and therefore the list is finite and easily computable.

If X is an output symbol, the descriptor contains the address of the corresponding event performance routine.

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

5.2. 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) flags indicating whether the symbol is active, expanded, or fully expanded; 3) if expanded, a pointer to the production descriptor; 4) pointer to attribute descriptors for this symbol; 5) a pointer (possibly null) to a timer descriptor. The attributed parse tree is rooted by an instance (called *root*) of the goal symbol of the grammar.

Attribute Instance Descriptors:

Each attribute is described by a record containing old and new values, and a list of "immediate production links", each of which points to an active symbol instance having an immediate production whose enabling condition depends on this attribute.

Timer Descriptors

Each timer is described by a record containing pointers to a symbol instance, a timed production, and a delay. The parser maintains an incremental delay queue of active timers.

The following FIFO queues are used by the parser during processing:

(1) A queue of input events to be processed.

(2) A queue of timed-out timer descriptors waiting to be processed.

(3) *Newly active queue*, a queue of symbol instances that have just become active and are awaiting processing. A symbol is added to the tail of this queue when the left siblings of its concurrent-activation group are fully expanded.

(4) *Immed queue*, a queue of active nonterminal instances that may be eligible for an immediate production. A symbol is added to this list when an attribute value on which the enabling condition of the production depends is changed.

5.3. Processing Input Events

The first step in parsing an input event **[x]** is to find the "candidate set", namely the set of non-terminal leaves which left-derive **[x]** in the underlying CFG, and which are eligible under the "key attribute" restrictions. This can be done by a depth-first search of the parse tree. For each candidate **<a>**, there is a set of production sequences which left-derive **[x]** from **<a>**. These sequences are attempted in an arbitrary order.

When a sequence longer than one production is being attempted, 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 a production sequence fails (due to an enabling condition not being met) then all its assignments must be undone. Hence while a multi-production sequence is being attempted, its attribute assignments are "logged" (i.e., modified attributes are linked and their old values saved). If the production sequence fails, the changes are undone.

The immediate and newly active queues are processed when the candidate list is exhausted. This processing, which may add new entries to the queues, is continued until both queues are empty.

An element on the immediate queue is processed by testing the enabling conditions of its immediate productions, and applying the first production whose enabling condition is met.

The processing for a newly active symbol X is as follows: if X is a nonterminal, the conditions of its immediate productions are evaluated, and if one is true that production is applied. If none are true, links from the attributes on which the conditions depend are established, so that if one of the attributes is changed the conditions can be retested. For each timed production of X, the enabling condition is evaluated. If it is true, the expression assigned to the "interval" attribute of **/timer/** is evaluated, and a timer is started with that value. A link is set up so that if X is later removed from the tree, or is expanded by another production, the timer can be canceled.

If X is an output symbol then the corresponding external function is called; the attribute values of X are passed as arguments. If X is a special symbol such as **/remove/** or **/freedata/**, the appropriate internal action is taken. **/remove/** removes from the parse tree the descendants of the

symbol instance pointed to by /remove/'s *where* attribute, and marks this symbol as fully expanded.

5.4. Processing Timeouts

A timeout is processed by performing the associated production on the associated symbol instance. As with input events, this can result in additions to the newly active queue and the immediate list, which then must be processed as described above.

5.5. Deallocation of Fully Expanded Subtrees

When all the symbols in the RHS of a production are fully expanded, the parser deallocates them and their attributes, and recursively flags and processes the parent. The other case where symbols are deallocated involves right-recursive productions. In a production of the form

<x> : S <x>.

when the symbols in S have all been fully expanded, and the two instances of <x> have identical attribute values, the symbols in S are deallocated and the two instances of <x> are merged (the child replaces the parent and the parent is deallocated). This is done to prevent right-recursive constructs from using unboundedly large amounts of memory.

6. RTAG SOFTWARE SYSTEM

In the section we describe software tools which, given an RTAG specification, produce an implementation of that part of the protocol which the grammar describes. Tools have also been developed to assist in user-level debugging of RTAG specifications and of interface routines. This software has been developed under 4.2 BSD UNIX, but could be built on other systems as well. There are two main components: the "grammar analyzer" and the "RTAG parser", which correspond roughly to the parser generator and the parser of a compiler generation system.

6.1. Grammar Analyzer

The grammar analyzer accepts a symbolic RTAG specification, and generates the static data structures of the RTAG parser (see Section 5.1). It assigns numbers to input symbols, and produces

an "include" file of these assignments, for use in the user-supplied interface to the RTAG parser. It also produces a C source 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.

The grammar analyzer was constructed using the UNIX Lex and Yacc utilities to facilitate changes to RTAG, and it uses the UNIX C pre-processor to handle comments, macro substitutions, and include files.

6.2. RTAG Parser

The parsing algorithm described in Section 6 has been implemented in C under UNIX. It can be run in user or kernel mode. It must be linked with appropriately-named output routines and external functions, which are called indirectly through the address table generated by the grammar analyzer.

The networking portion of the UNIX kernel includes the "mbuf" memory management system, in which variable-size blocks of data are represented by chains of "mbuf" records, and routines are available to manipulate mbuf chains in various ways. On UNIX, the RTAG *dataptr* attribute type is a pointer to an mbuf chain, and the RTAG parser performs concatenation and other operations by calling the appropriate utility routines.

6.3. Software Environments for the RTAG Parser

To obtain a kernel-mode implementation of an RTAG-specified protocol on UNIX, the RTAG parser is compiled into the kernel, along with the structure initializations generated by the grammar analyzer. Routines must be supplied to interface the RTAG parser to the UNIX socket system on the upper level, and to the UNIX IP implementation or directly to network interface modules on the lower level. We have also developed two user-level environments for protocol experimentation and development:

1) User-level Experimentation System

This consists of a simulated network layer based on the UNIX IPC facility, as well as routines which translate between output symbols and packets (correctly handling data pointed to by *dataptr* attributes). Also included is a routine for interactive perusal of the parse tree, and debugging options that allow logging the actions of the parser (productions, event occurrences, and attribute assignments) on a disk file. This system allows RTAG specifications to be tested with a minimum of programming.

2) 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 kernel code (such as the socket routines and the read/write system call routines) into the user program, together with a software simulation of the IP and network layers. The tree perusal and logging routines, as well as UNIX symbolic debuggers, can be used.

6.4. Example: an Implementation of TP-4 under 4.2 BSD UNIX

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

(1) Network-level event output routines for assembling packets and sending them to the UNIX IP (Internet Protocol) module.

(2) A network-level input routine to accept a packet from IP, verify the checksum, extract the data fields, and generate the appropriate input event.

(3) Upper-level event output routines for conveying information to the UNIX socket system. For example, the data output routine calls a UNIX kernel routine which appends the mbuf chain to the receive buffer of the appropriate socket, and wakes up any waiting user process.

(4) An upper-level input routine to handle "user requests" from the socket system. These, for the most part, translate directly into input events.

7. CONCLUSION

The RTAG methodology is rich enough to express a wide variety of protocol concepts. Furthermore, the RTAG methodology can be used to experiment with variations on protocols (by changing the grammar) and to quickly obtain experimental implementations of protocols. We have implemented an RTAG software system, have specified a RTAG for the TP-4 transport protocol, and have developed a grammar-based TP-4 implementation in the UNIX kernel.

Based on our experience with the current RTAG TP-4 implementation and with a number of other protocol implementation projects we have concluded that grammar-based formalisms are well suited to specifying complex protocols. We believe that RTAG has inherent advantages over protocol specification methodologies based on augmented FSA. In particular RTAG provides:

(1) Ease of Use and Understanding: An RTAG specification of a protocol compactly shows the legal event sequences which are often difficult to deduce from other specification formalisms. In addition, with RTAG one can isolate particular functions in well defined portions of the grammar and of the parse tree. The capability of RTAG to express concurrent activities makes it easier to express natural protocol interactions and dependencies, and for others to understand the mechanisms of the protocol.

(2) Range of Applicability: Distributed computing applications in databases and operating systems have introduced the possibility of protocols more complex than those found in data communication protocols. Such applications may benefit from RTAG's powerful underlying formalism.

We believe that development of the RTAG methodology can have a major impact on the way protocols for a wide range of applications are specified and implemented. In addition, we hope to develop RTAG-based tools to assist in the verification of both protocol specifications and implementations.

REFERENCES

Blu82

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

Bri85

Brinksma, E. A tutorial on LOTOS. Proc. 5th IFIP Workshop on Protocol Specification, Testing, and Verification. June, 1985.

Har78

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

Mil80

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

Nas83

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

Poz82

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

Sch80

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

Sid83b

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

Sun81

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

Ten78

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

Yem83

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

APPENDIX I: RTAG SPECIFICATION FOR TP-4 PROTOCOL FRAGMENT

```
/* RTAG specification for regular send portion of TP-4 */

/* macro definitions for parameters */

#define RETRANS_TIME      5       /* retransmission interval */
#define GIVEUP_TIME       30      /* time before giving up */
#define RETRANS_COUNT     4       /* # of retransmissions */


/* TERMINAL SYMBOL DEFINITIONS */

output  |N-DT|/* data packet to network */
  int     src_ref, dst_ref, seqno
  boolean       eot
  dataptr data

input   |N-AK|/* acknowledgement packet from network */
  int     refno, credit, seqno

input   |U-GR|/* user initiated graceful close */
  int     refno

output  |U-GR|/* user graceful close indication or confirm */
  int     refno

input   |U-DT|/* data from user */
  int     refno
  dataptr data


/* NONTERMINAL SYMBOL DEFINITIONS */

nonterm <transact>      /* data transfer and graceful close */
  boolean start_send    /* set when connection established */
  boolean start_receive
  int     initial_credit
  int     nxoutstanding /* # of unacked expedited packets */
  boolean GRarrived     /* true if have in-sequence GR. */
  boolean GRsent        /* true if have received GR from user */

nonterm <reg send>
  int     windowstart   /* first unacknowledged packet */
  int     nextseq       /* seqno of next packet */
  int     windowend     /* seqno just beyond end of send window */

nonterm <send message>
  boolean ready         /* true iff can start sending message */

nonterm <send message tail>
  boolean ready

nonterm <transmit GR>
```

```
  boolean ready

nonterm <transmit message>
  boolean ready
  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 <get GR>



/* DECLARATIONS OF EXTERNAL FUNCTIONS */

extern   boolean #between#(int,int,int)
         /* (i,j,k); true if i < j <= k in cyclic order */
extern   dataptr #extract#(dataptr,int)
         /* return copy of initial segment of given size */
extern   boolean #eot#(dataptr)
         /* true if empty */
extern   dataptr #copy#(dataptr)
         /* copy mbuf chain */


/* PRODUCTIONS */


<reg send>   :   <send message tail>.
         $0.nextseq = 0
         $0.windowstart = 0
         $0.windowend = <transact>.initial_credit
         $1.ready = true
;

<send message tail>   :   {<send message> <send message tail>}.
         $1.ready =* $0.ready
         $2.ready = false

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

```
<send message>   :  |U-DT| <transmit message>.
        $2.data = $1.data
        $2.ready =* $0.ready

        |       |U-GR| {<transmit GR> <get GR>} |U-GR|.
        if not <transact>.GRarrived
        $2.ready =* $0.ready
        $4.refno = <TC>.refno

        |       |U-GR| <transmit GR>.
        if <transact>.GRarrived
        $2.ready =* $0.ready
;

<get GR>   :  .
        if <transact>.GRarrived
;

/* productions of <transmit GR> omitted for brevity */

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

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

        |       /freedata/.
        if $0.eot
        $1.data = <transmit message>.data
        <send message tail>/<send message 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| /freedata/.
        if #between#(<send packet>.seqno, $1.seqno,
          <reg send>.nextseq)
        <reg send>.windowend = ($1.seqno + $1.credit)
          mod <TC>.max_seqno
        $2.data = <send packet>.data

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