Grammatical Channels

Thomas E. Portegys
Illinois State University, USA
portegys@ilstu.edu

Abstract

A grammatical channel is a full-duplex message passing channel which is capable of parsing a user level message protocol upon request. Multiple application-level protocols may be supported in an application. Grammatical channels may be used to improve interface specifications, software maintenance, error-checking, and security.

Keywords:
Communication channel, interface specification, security.

Introduction

A grammatical channel is a full-duplex message passing channel which is capable of parsing an application-level message protocol upon request. When no parsing is requested, the channel simply passes messages through the channel transparently.

In this context a channel is simply a two-ended path along which messages may be sent. The ends of a channel are distinct. A protocol is a set of conventions, or grammar, governing the passing of messages on a channel.

The concept of an application-level protocol is not new, of course. It is a language that allows application processes to talk to each other. It has been the subject of extensive research based largely on Hoare's Communicating Sequential Processes (Hoare, 1978), and Milner's Calculus of Communicating Systems (Milner, 1993). However, when the concept of a protocol is combined with the concept of a channel, especially in the sense of embedding a protocol into a channel, one typically thinks of data transport protocols rather than application protocols.

The purpose of an application protocol, however, is different from the purpose of a data transport protocol. A data transport protocol, such as IP, HTTP, or X.25 (Tannenbaum, 1996), serves to move data from place to place, typically over some distance. A data transport protocol is characterized as being highly standardized, somewhat formal, and difficult to change. This is as it should be, since it is used for transporting data between diverse machines and applications.

An application protocol, on the other hand, must be able to take varied forms and to change to suit the user application. Indeed, varied forms of protocols may exist within a single application. Some systems which involve many communicating processes performing a wide variety of tasks may support many message protocols in the system.

A grammatical channel is a channel with an application level protocol embedded into it. The purpose of embedding the protocol into a channel is to gain the advantages of a data transport protocol, namely, formality of specification and run-time enforcement, for an application protocol, while allowing it to remain easy to define and change. In addition, as many application protocols as needed by a system can be supported.

The service provided by a grammatical channel is two-fold:

- To transport application messages through the channel. This may involve the utilization of an underlying data transport object, e.g. a TCP socket, as part of the channel implementation.

- To parse an application protocol upon request. This service provides a syntactic check on the correctness of a message exchange. The grammar expressed in the protocol is an essential part of the application message interface.

An important aspect of the parsing service could be the detection and prevention of intentional attacks on a system via the exploitation of protocol vulnerabilities. A grammatical channel, if detailed sufficiently, could serve as an independent specification of what legal exchanges may be made between processes. For example, a buffer-overflow attack exploits a coding error that may be caught by checking a specification of the valid contents of an incoming message.

Background

A small amount of history is in order to explain how the concept of a grammatical channel originated, and what an application of it might be.

Years ago, I was assigned with another engineer to look at the problem of describing the interfaces between the software modules of the Lucent Technologies' 5ESS Digital Telecommunication Switching System (Portegys and Van der Gaag, 1985), which contains millions of lines of C code and thousand of modules. One aspect of this task was to attempt to describe the message interfaces between processes at work in the switch. In order to do this, it was necessary to scan the code for message send and receive primitive calls, and to piece together scenarios. There was some documentation available to help with this, but it was not kept up to date. In general, what makes this task difficult is that multiple processes may be involved in message exchanges, and an individual process may be involved in more than one message exchange at a time.

At some point the thought of defining a protocol associated with each message exchange sequence came about. The protocol would be associated with a message medium, not a process, and would therefore be independent of the process configuration. It would also be possible to more easily find the code which references a particular protocol via searching for symbols and/or trapping messages. Finally, the protocol would not be simply an inert design description which is divorced from the environment in which it

runs, and therefore becomes rapidly obsolete, but rather an entity which is also active at run-time.

This is the history of the grammatical channel. Although to adapt some version of it to the 5ESS software would require a good deal of work, the concept is applicable to it and other applications.

A concept similar to a grammatical channel is the UNIX stream I/O system (Ritchie, 1984). This system possesses active processing elements set into a message stream which can be dynamically altered. These processing elements serve to translate messages into forms which are compatible with the users at the ends of the stream. The end users are typically a process communicating with a device.

To better illustrate the concept of a grammatical channel, the next section presents a design of a prototype. It is not the only implementation which could be made.

Prototype Design

A design of a grammatical channel prototype is presented in this section. A daemon process sitting inside the channel performs the parsing service, as shown in Figure 1 for communicating processes A and B.

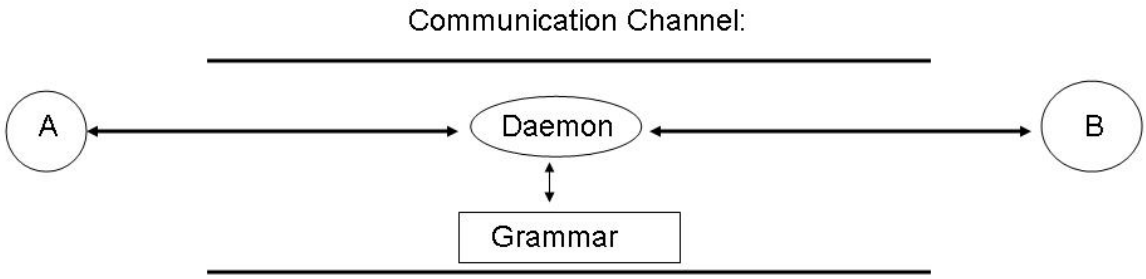

Figure 1 – Grammatical Channel

Channel Functions

A number of channel functions allow an application to make use of a grammatical channel. These are:

- *chclose*(2) - close a channel.
- *chcreat*(2) - create a channel.
- *chflush*(2) - flush messages out of a channel.
- *chkill*(2) - kill a channel.
- *chopen*(2) - open a channel.
- *chpc*(1) - channel application protocol compiler.
- *chrcv*(2) - receive a channel message.
- *chsend*(2) - send a channel message.

The normal operational scenario is as follows:

1. Use *chcreat*() to create a channel.
2. Use *chopen*() to open an end of the channel for sending and receiving messages.
3. Use *chsend*() and *chrcv*() to send and receive messages. *Chrcv*() allows a process to wait on a message from any of its open channels.
4. Use *chclose*() to close the channel.
5. Use *chkill*() to destroy the channel.
6. *Chflush*() may be used to recover from a protocol error.

Of special interest is the channel function STARTP in *chsend*(), which initiates the user protocol checking program given in the protocol name argument. This program would have been previously created by the *chpc* command from a protocol specification (see next section). The ENDP channel function is the signal to normally return to non-parsing mode. STARTP and ENDP may be sent from either end of the channel. Using these commands, any protocol may be made active on any channel.

Once activated, the protocol-parsing demon begins to parse messages by message type. Upon encountering an error, the parsing is terminated and an error is indicated, but the message passing continues. In other words, if a message is sent through a channel which violates an active protocol, the message will be flagged and passed on to the receiving end, where the error will first be detected by the process which reads the message. It is then the responsibility of that process to initiate error handling activities.

Some other notes of interest:

- There may be more than one process sending and receiving messages on the same end of a channel. The channel itself is not concerned with this: special arrangements for cooperation must be made among the processes.
- No time-out arrangements are made by the channel for user processes. This is also their responsibility.
- There is no facility for waiting on a message of a particular type.
- A process must always wait for a message if none is immediately available.

Protocol Compiler

It is necessary to have a means to describe an application protocol in such a way that it may be compiled into a message parsing program.

An application protocol is a context-free grammar with a simple syntax. C-type comments are allowed. The direction of messages on the channel is important and denoted by enclosing messages from the "A" end in (), and enclosing the messages from the "B" end in []. These messages are the terminal symbols in the grammar. Non-terminal symbols are enclosed in <> as in Backus-Naur form. Symbols consist of strings

of alphanumeric characters and underscores. An or symbol, "|", allows productions with the same left hand sides to be combined.

The following is an example of a protocol for a conversation between neighbors:

```
<start> -> (ANY_BODY_HOME) <answer>;

<answer> -> [GO_AWAY] |
            [WHATS_UP] <transaction>;

<transaction> -> (CAN_I_HAVE_A_CUP_OF_SUGAR) <sugar_reply> |
           (WHATS_THE_WEATHER_LIKE_OVER_THERE) <weather_reply>;

<sugar_reply> -> [SORRY_PLUMB_OUT] <next> |
                 [SURE_THING] <next>;

<weather_reply> -> [NICE_DAY] <next> |
                   [IT_STINKS] <next>;

<next> -> <transaction> |
          (BYE_BYE);
```

Appendix 1 presents a more realistic example of a telephone call protocol.

Usefulness

The usefulness of a grammatical channel is as follows:

- Process interface specification. The protocol grammar is a statement of a dynamic interface between processes at an application level. This can be viewed as a contract between processes communicating over a channel. The fact that the protocol is associated with the channel allows process activities that are not essential to the interface, such as message exchanges over other channels, to be ignored.
- Maintenance. Not only is the protocol stated explicitly, but it is also built into the software in a form that allows identification of the programs which employ the protocol, thereby allowing the impact of protocol changes to be evaluated. In addition, the protocol specification is easy to change.
- Error checking and security. For testing and for live running, the use of the protocol checking capability provides an additional ability to catch errors and security violations. Of course, grammatical channels do not in any way guarantee the correctness in the design of an interface, nor do they help in checking the semantic content of user messages. They are a major step toward formalizing these interfaces, however, which is helpful toward a correct design. The specification of a user protocol may be checked for correctness by tools along the lines of *Spanner* (Aggarwal et. al., 1985).

Conclusion

Grammatical channels are a different way to look at protocols. The emphasis is on the specification of a protocol as a language spoken among application processes rather than a means to transport messages, and the embedding of this language into a channel. The protocol is actively enforced during operation, which in turn forces the specification to be maintained. In addition, in applications which have a need to support multiple application protocols, ease of creating and modifying protocols is accommodated.

References

Aggarwal, S., Barbara, D., and Kalman, M. Z., "SPANNER: A Tool for the Specification, Analysis, and Evaluation of Protocols", *Bell Laboratories Technical Memorandum* 11211-850208-02TM, February 8, 1985.

Durroei, S., and Crocker, M., "A Dynamic Representation of Grammatical Relations", *Proceedings of the LFG97 Conference*, University of California, San Diego, 1997.

Hoare, C. A., "Communicating Sequential Processes", *Communications of the ACM*, 21(8), pp. 666-677, 1978.

Milner, R., "The polyadic π-calculus: a tutorial", In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pp. 2-3-246, Springer Verlag, 1993.

Portegys, T. E., and van der Gaag, P. J, "A 5ESS Module Interface Specification Proposal", unpublished, 1985.

Ritchie, D. M., "A Stream Input-Output System", *Bell System Technical Journal*, 63, No. 8, October 1984.

Tanenbaum, A. S., *Computer Networks, Third Edition*, Prentice-Hall Inc., 1996.

Appendix 1 – Sample Telephone Call Protocol

```
/*
 * A protocol to model the messages in a telephone call
 * between a process handling the originating and
 * terminating end.  The originator sends to the A end
 * of the channel, and the terminator sends to the B
 * end.  The terminator is able to hold the originator,
 * i.e. prevent from disconnecting, with a HOLD signal.
 *
 * Message tokens: ACK -            acknowledgement
 *                 ANSWER -         answer
 *                 BAD_ADDRESS -    invalid called party address
 *                 BUSY -           busy
 *                 DISC -           disconnect
```

```
*                 HOLD -          hold request
*               RELEASE -   release of hold
*
*/

     /* caller to find out the called party condition */

     <start> -> [BUSY] |           /* called is busy */
                [BAD_ADDRESS] |    /* invalid called party */
                [ANSWER] <talk>;   /* called answers */

     /* talk state */
     <talk> -> <disconnect> |      /* disconnect */
                 <hold>;           /* hold request */

     /* disconnect state */
     <disconnect> -> (DISC) [ACK] |    /* caller disconnect and
                                          acknowledge */
                   [DISC] (ACK) |      /* called disconnect and ack */
                   (DISC) [DISC] [ACK] | /* race conditions */
                   [DISC] (DISC) [ACK];

     /* hold request state */
     <hold> -> [HOLD] (ACK) [RELEASE] <talk> |  /* hold, then return
                                                   to talk */
               [HOLD] (ACK) [DISC] (ACK) |      /* hold, then
                                                   disconnect */
               [HOLD] (DISC) [ACK] |            /* race for hold */
               (DISC) [HOLD] [ACK];
```