The Pandora System: An Interactive System for the Design of Data Communication Protocols

Gerard J. Holzmann
Delft University of Technology, P.O. Box 5, 2600 AA Delft, The Netherlands

PANDORA is an interactive system for the analysis, synthesis, and real-time assessment of data communication protocols. The Pandora system is being developed at the Delft University of Technology in cooperation with the Dr. Neher Laboratories of the Netherlands PTT. This paper gives an overview of the structure of the system and discusses the main design goals.

Keywords: automated protocol validation, deadlock detection, validation algebra, protocol synthesis, protocol assessment.

Gerard J. Holzmann was born in Amsterdam, The Netherlands, in 1951. He received the B.S. (1973) and M.S. (1976) degree in electrical engineering, and the Ph.D. (1979) degree in technical sciences from the Delft University of Technology in The Netherlands. He obtained a Fulbright Fellowship in 1979. From September 1979 to June 1980 he was with the University of Southern California in Los Angeles. From June 1980 to June 1981 he worked at the Computing Science Research Center of Bell Laboratories in Murray Hill. Dr. Holzmann is now an assistant professor at the Delft University of Technology. In Oct. 1981 he was awarded the Prof. Bähler prize by the Royal Dutch Institute of Engineers (KIVI) for his research on telecommunication systems.

North-Holland
Computer Networks 8 (1984) 71–79

1. Introduction

Data communication protocols are used to formalize the interactions in distributed computer systems [10,11]. The protocols take care of such issues as routing, flow control, error recovery, and connection establishment. A badly designed protocol can degrade a system's performance significantly, it can introduce errors, and it may even surprise its users by bringing their system to a complete standstill at the occurrence of certain unexpected combinations of events. Such occurrences are almost always time-dependent and thus very hard to trace.

So, protocols are an important part of distributed systems. But, "good" protocols, that is: protocols with provable properties and with measurable real-time characteristics, turn out to be extremely hard to design.

The Pandora system, an acronym for "interactive system for Protocol ANalysis, Design and OpeRation Assessment," aims to provide the protocol designer with a set of tools that can facilitate this task.

The Pandora system consists of three major parts: analysis, synthesis, and real-time assessment.

(1) In protocol analysis we form an algebraic model of a protocol and prove its consistency, completeness, and freedom from deadlocks.

(2) In protocol synthesis we try to guide a protocol designer to a description that is succinct, complete, and correct.

(3) In protocol assessment we measure the real-time characteristics of protocols.

Each of the following sections will describe a small part of this Pandora system.

In section 2 we give a general overview of the hardware and software configuration. Section 3 describes the protocol validation method. In section 4 we discuss protocol synthesis guidance. Section 5 describes the design of the network simula-

tor, used in protocol assessment studies. Section 6, finally, summarizes the paper.

2. System Overview

2.1. Hardware configuration

The Pandora system consists of two 11/23 computer systems, each equipped with 256 kb random access memory (Fig. 1). The two systems are connected via a programmable network simulator (section 5) used for protocol assessments. Each station is equipped with a 480×1024 dot graphical display, used for protocol synthesis. One of the two stations is, furthermore, connected to an 80 Mb winchester disk for the storage of analysis data, and to a plotter used for drawing SDL diagrams [2,3] documenting protocol designs.

Either station can be used for protocol synthesis and logical analysis. The two stations cooperate, though, in real-time assessments of protocols, exchanging data via the network simulator.

2.2. Software structure

The Pandora software is written in C and lives in a Unix environment. (Unix is a trademark of Bell Laboratories.) It consists of five programs: 'prosy,' 'proco,' 'pan,' 'pass,' and 'plot.' Figure 2 illustrates the dependencies.

Prosy

'Prosy' is an interactive program used for protocol synthesis. The program provides the user with a rich editing environment that allows for elaborate experiments with partial protocol designs. 'Prosy' forms the 'presentation layer' of the Pandora system. It gives access to the services of the other four programs: to 'proco', for static protocol analyses and conversions, to 'pan' for dynamic analyses, to 'pass' for real-time assessment, and to 'plot' for documentation.

The user can describe a design in an abstract 'meta-language,' derived from SDL (section 4). The language supports the SDL enhancements described in [4], and thus encourages structured design by stepwise refinement.

Via 'proco' designs can be converted into flow diagrams, which can be plotted on either the graphics display or the hardcopy plotter (Fig. 1).

Proco

'Proco,' the protocol compiler, is the heart of the Pandora system. It interprets the SDL dialect used by the protocol synthesizer, and reports back to it with the results of syntactic and semantic analyses.

Once the initial (syntax) errors in a design have been overcome, 'proco' can convert the description into the proper format for each of the programs 'pan,' 'plot' and 'pass.' For 'pass' it generates a C implementation of the protocol, for 'pan' it extracts an algebraic skeleton of the protocol in the form of protocol expressions [5,6], and for 'plot' it can construct the flow diagrams documenting a design.'

When generating the protocol implementation 'proco' by default fills in the details left unspecified by the user, using library routines. It thus

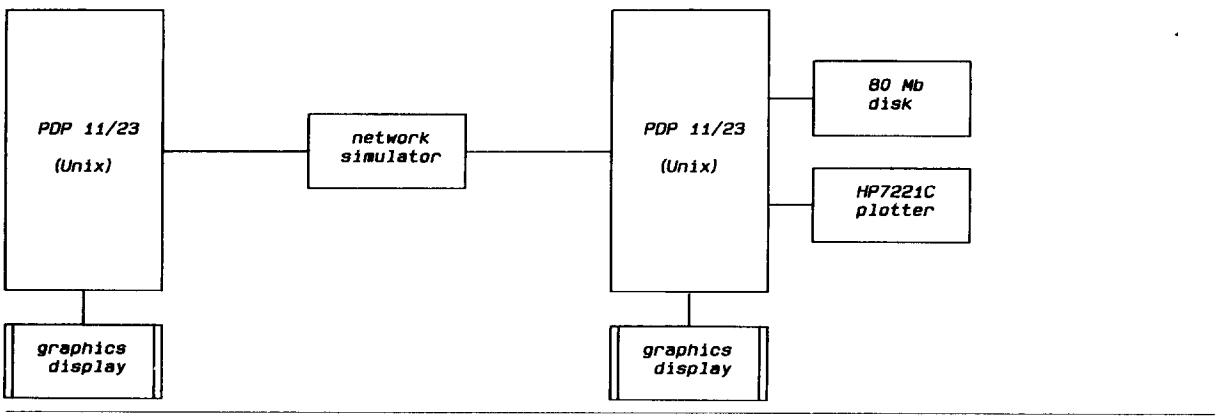

Fig. 1. System Overview

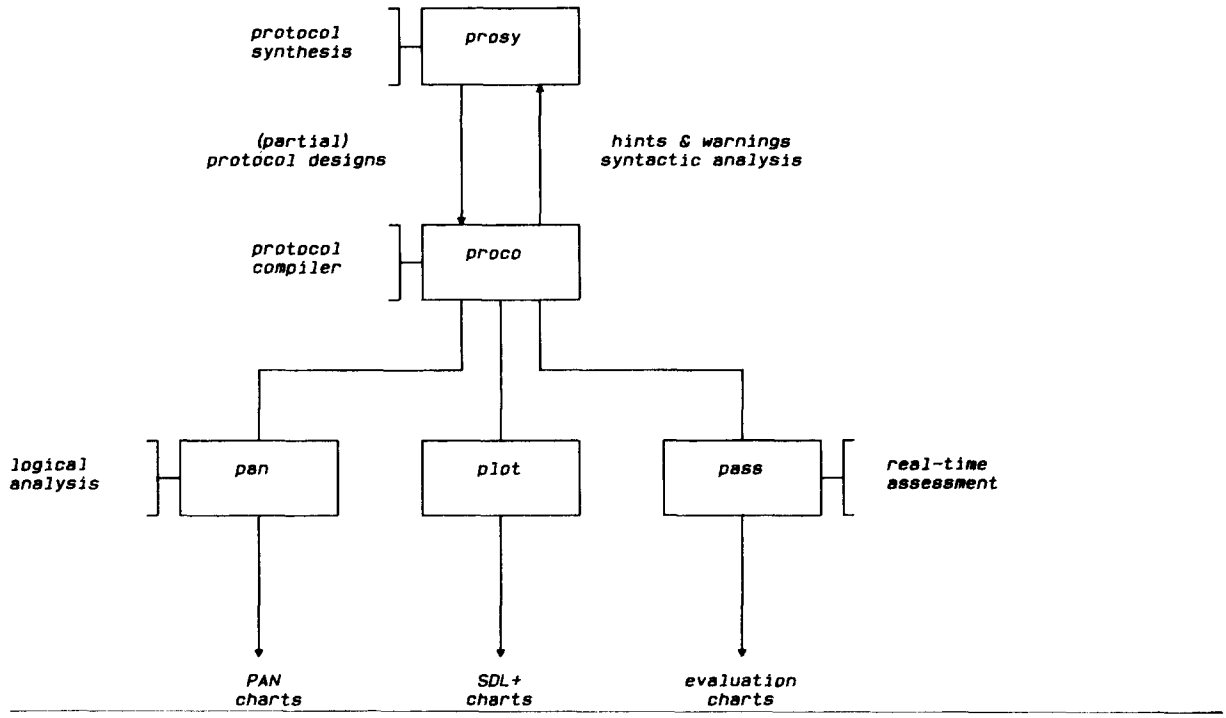

Fig. 2. Software Structure

closely approaches the ideal of an 'automatic implementation' (e.g. [8,9,13]). The program was written with the aid of Unix tools Yacc and Lex.

Pan

'Pan,' the protocol analyzer, tries to exploit the protocol validation algebra described in [5,6], in a search for the logical design errors that were missed (or ignored) in the design phase. 'Pan' will analyze a given protocol for the presence of deadlocks, logical incompleteness, and potential problems caused by timeouts.

Pass

'Pass' is the name of the software package that will be used for the real-time assessment of protocols that survive the earlier Pandora phases. The network simulator (a hardware device) works under the direct control of the 'pass' programs. The evaluation report prepared by 'pass' contains its real-time characteristics in a given network environment. 'Pass' will measure the protocol's sensitivity to errors, and the overhead involved in its operation.

Plot

'Plot,' finally, is an interpreter that generates the proper code for drawing symbols on either the screen of a graphics terminal, or on the hardcopy plotter. The use of 'plot' as an interpreter rules out the need for storing bulky 'plot-image files' describing diagrams as series of detailed plotcommand codes. The input to 'plot' is a normal textfile, which can be edited like any other file. The descriptions, interpreted by 'plot,' are written in an intermediate level language, with statements of the form "moveup (15mm), rectangle, text ('string')," etc. The language contains commands for all symbols from the protocol specification language SDL section (4.1).

Normally, the plot language is invisible to the protocol designer. The user can, however, equally well design and plot figures directly in this language, independent of the protocol editing environment provided by the other four programs described above.

3. Protocol Validation

3.1. The validation algebra

The validation algebra on which the Pandora system is founded is briefly described below. For a

more thorough introduction to the algebra the reader is referred to [5].

The behavior of each process that takes part in a protocol message exchange can be modeled in a formal language. In [5] we define a special class of algebraic expressions, named 'protocol expressions,' that can be used to formalize such a language.

To analyze a protocol one can define special operators that mimic the effects of concurrency, and then formalize the analogons of standard protocol design errors within the algebra (e.g. the inclusion of a deadlock). The problem of analyzing a protocol is thus transformed into the problem of analyzing an algebraic expression. Fortunately, it is relatively easy to write a program that can accomplish this task efficiently for a fairly large class of protocols. The program 'pan' is one such program. A first version of 'pan' has been in use, under a Unix operating system, since September 1980.

To get the flavor of an analysis in the validation algebra, consider the following expression:

$$[(1/m) \cdot (r/m)^* \cdot (a/e)] \times [m/(a + r)) \cdot e]$$

The expression contains a cross product of two terms. Each term formalizes the behavior of one specific process. The first process (the left-hand term) can send two messages to its partner: 'm' and 'e' ('message' and 'end of transmission'). The second process can respond with two messages 'r' and 'a' ('reject' and 'acknowledge').

It should be observed that the symbols that represent messages to be transmitted by a process are always written in the denominator of fractions; similarly, symbols to be received occur in the numerators only. This is in fact the rule for distinguishing input from output: every numerator is an input, and every denominator is an output message.

The other operators that occur in the above expression are simple, that is, once you get used to the notation. The dot formalizes a time ordering, ("a.b" means "first 'a' then 'b' "), the plus indicates a choice (' 'a + b' means that the process can perform 'either 'a' or 'b' ' '), and the Kleene star indicates repetition ("a*" means "perform 'a', zero or more times").

In this example we have assumed that all messages transmitted will eventually be delivered unharmed by the transmission channel. The validation algebra itself does not have these restrictions, that is: it can equally well model mutation, distortion, and deletion errors [5,6].

The expression can now be interpreted as follows: the left-hand side process is trying to send a message 'm' to its partner on the right-hand side of the cross. This accounts for the first fraction '(1/m)' in the expression. The right-hand side process will either acknowledge ('a') or reject ('r') that message, which explains the term '(m/(a + r)),' and will die after the closing symbol 'e' has come in. The first process meanwhile can either skip or repeat the term '(r/m).' In either case it should await the reception of the response from its partner to make the choice of whether to skip (if the response is an 'a') or repeat (if it turns out to be an 'r').

When the response is 'r' the 'm' message will be retransmitted; when the response is 'a' the process will send the closing symbol 'e' and die.

Elaborating the above expression while following the rules of the algebra [5,6] we find the following result:

$$(1/m) \cdot ((m/a) \cdot (a/e) \cdot e + (m/r) \cdot (r/m) \cdot 0)$$

which can be reduced to:

$$1 + (1/m) \cdot 0$$

The presence of a term ending in a zero indicates that the protocol examined has a design error. Note that the symbol '0' turns up when the second process tries to receive a symbol 'e' when none was sent. The process will lock forever. The first process is also locked while awaiting another 'a' or 'r.' The zero thus conveniently flags process deadlocks.

The complete validation process can trace potential timeout problems, residual messages, and incompleteness in protocol specifications [5,6].

3.2 Reduction technique

Compared to global state space exploration techniques [14], the validation method used in the Pandora system allows for a number of important reductions in the size and the complexity of an analysis.

These reductions are based on the notion of equivalence classes of execution sequences, which will briefly be illustrated below.

If we restrict ourselves to execution sequences

of finite duration (in most cases this is quite acceptable [5,7]), we can define an equivalence relation on the (finite) set of all possible execution sequences S. This equivalence relation partitions the large set S into a smaller number of equivalence classes.

The validation process can now be restricted to examining just one characteristic sequence from each equivalence class.

The method works for any number of processes, and applies to any type of protocol. The gain over earlier reduction techniques [12,14] can indeed be quite significant. Experience with this reduction method, furthermore, dates back to the very first versions of the program 'pan' (see section 3).

A detailed description of the equivalence relations used in the Pandora system and the implementation of the reduction process is presented in [7]. Previous results with the use of the analysis method can be found in [5].

4. Synthesis Methodology

4.1. The specification language

The specification language used in the Pandora system was designed to be compatible with both SDL [2,3] and the algebraic notation discussed in section 3. For an introduction to the specification and description language SDL the interested reader is referred to [2]. The following discussion, though, can be read without prior knowledge of SDL.

The Pandora specification language is a true meta-language for protocol specifications. It is used to specify the outline, the skeleton of a protocol, while leaving standard details to be filled in by special purpose programs such as 'prosy' and 'proco.'

Like SDL, this language is used only for control-flow specifications. Its expressiveness is strictly linked to the analytical power of the validation method used in the Pandora system. Specifically, no attempt is made to extend the language beyond the power required for generating the formulas for protocol analyses. Without restriction, though, the user can include code of the target programming language (see below under 'Compilation') in a specification. Anything that is not directly related to the exchange of messages can freely be specified in this manner, using all facilities available in the target language (e.g. abstract data types, standard library routines). In the analyses the included code segments are simply treated as comments.

This approach has important advantages. For one thing, our task of language design is greatly simplified. We can permit ourselves the luxury of a terse well-structured control-flow specification language and still maintain the versatility of the target language. We inherit the full power of the target programming language without having to mimic the complete compiler for that language.

Unlike SDL, the specification language we use is block structured: it allows for proper nestings of three basic control structures (sequencing, selection, and repetition). It also includes "reference tasks" and a version of "control tasks," both described in [4].

Control tasks are used for specifying selection and repetition (see below). The reference task is similar to a macro or procedure call. It is used to divide larger specifications into a number of smaller logical entities. This method is preferred over the method of splitting diagrams into subdiagrams using the SDL (e.g. page to page) connectors.

Possibly odd at first, the meta specification language does contain the goto statement. This means that the user is free to specify protocols as state transition tables, using a multitude of jumps. Since we do not want to rule out this option, the best we can do in synthesis guidance is to discourage the user from using it too lightheartedly. 'Prosy' tries to enforce a design discipline by considering all jumps potential violations of the control structure, and issues warnings wherever they appear.

The restrictions of the meta-language itself serve two purposes:

(1) they should encourage the usage of a design discipline that maximizes the chances for a well-structured, self-documenting protocol design, and

(2) they are intended to optimize the protocol validation process.

Note that, rather than attempting to solve the validation problem for arbitrary protocols, the Pandora system is meant to guide its users toward protocol designs that can fairly easily be validated.

The appendix lists the syntax rules for the meta-language, together with a description of the example protocol from section 3 in that language. In figure 3 the same protocol is represented graphically.

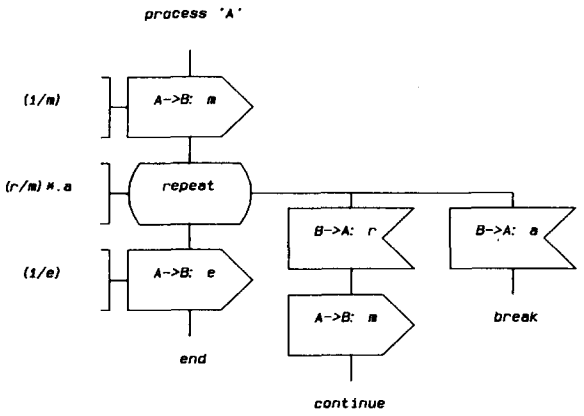

Fig. 3a. Process "A".

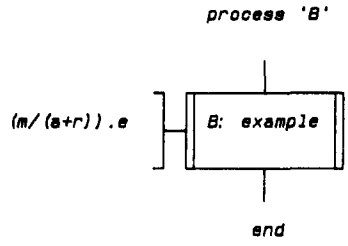

Fig. 3b. Process "B".

Every box in figure 3 represents an action: an output (arrow pointing out), an input (arrow pointing in), a reference task (two vertical bars), a control task (labeled "repeat" or "select"), or an internal action (rectangle, not shown).

In the example some of the actions have been tagged with the protocol expression they represent. Except for these annotations the diagrams are exactly as generated by the protocol compiler 'proco,' and drawn by the program 'plot' on the HP plotter (Fig. 1). The diagrams in figure 3 are discussed below.

Two different types of control tasks have been used: a repetition task ("repeat") and a decision task ("select"). The body of these control tasks is detailed in separate branches of the main control-flow tree. In fig. 3a the body of the control task is executed once for every 'r' message received. The task is terminated when the first 'a' message comes in.

In general, the repetition task is terminated only when a "break" label is encountered. By default, each branch in a repetition task is terminated with a "continue" label, which implies that execution is to resume at the start of the loop.

The body of process B (fig. 3b) is, by way of example, specified as a reference task. The specification for that task is given in fig. 3c. Here we see the second type of control task: a selection. In this case the body is a pure case switch, where one of the branches is to be selected and executed.

Clearly, both in repetition tasks and in selection tasks, branches starting with an input can only be selected if the corresponding message has been received.

There are two special cases for input specifications: "timeouts" and "defaults" (see appendix). An input specified as a "default" can be matched by any incoming message. An input specified as a "timeout" can be matched only if no other input message is available.

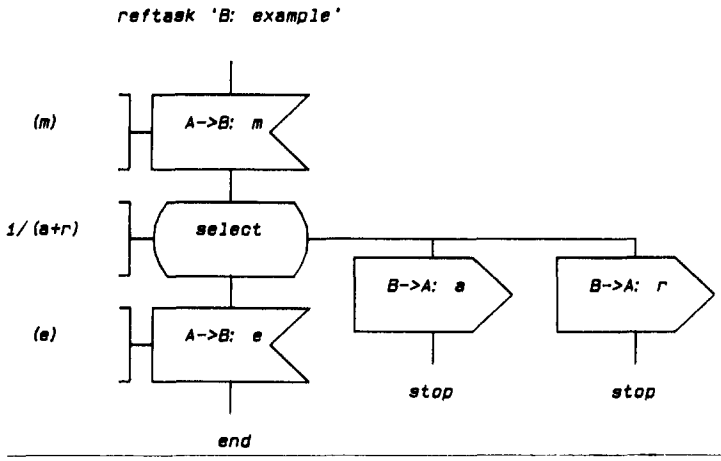

Fig. 3c. Reference Task Specification.

Branches that start with an output can always be selected. If according to these rules more than one branch could be selected the choice between the two will be made randomly (a non-deterministic choice). In the example protocol this situation occurs in the B process (fig. 3c, see also the appendix).

For a detailed motivation of the use of control and reference tasks we refer to [4].

4.1.1. Compilation

The abstract protocol specifications written in the specification language described above can be compiled into executable implementations by the protocol compiler 'proco.' The compiler uses a set of system dependent library routines (e.g. for sending and receiving messages) to link the implementation into a specific environment.

To override, to bend, or to extend the services of 'proco,' the designer can also include portions of C programs, such as the declaration of abstract data types or procedure bodies, within an abstract specification. These C modules are picked up and linked into the implementation by the protocol compiler.

4.2. Synthesis guidance

Prosy provides the protocol designer with two types of synthesis guidance: syntactical guidance and logical guidance.

The syntactical guidance is simple, but effective: 'prosy' can understand only a protocol designer who writes specifications in enhanced SDL, and it will flag all violations of that input language. In this respect, 'prosy' is as clever about the correctness of a protocol specification as an Algol compiler can be about the correctness of a program. But, there is one important difference: 'prosy' is interactive while most Algol compilers are not. That means that 'prosy' can flag and diagnose a mistake when it is made, and never has to deal with cases in which the diagnosis of an error is diffused by errors made before.

In addition to the syntax check performed by 'prosy,' the user can request a logical check of a partially specified protocol. This check is not performed autonomously for two reasons.

First of all, we found that a protocol designer is usually very much aware of the incompleteness of unfinished work, especially in the early stages of a design. It can be very irritating to have a machine remind you incessantly about things you have decided not to consider. Secondly, an autonomous logical check in the later design stages, when they are needed most, would slow down the response time for normal edit functions, which can again be a source of irritation.

A logical check, then, is performed only at the designer's request. Prosy will check a protocol's completeness and warn against redundancy. It will trace and flag:

(1) control tasks and reference tasks that remain to be detailed,
(2) messages that are sent but not received,
(3) messages received that are never sent,
(4) missing links in the control-flow, and
(5) violations of the control structure (see above).

In the same check 'prosy' will also report some statistics on the protocol designed so far:

(1) the number of processes specified,
(2) the relative size of the processes (measured in numbers of states),
(3) a listing of the system vocabulary, and
(4) a review of the communication structure (all communication links with a summary of the messages they carry).

5. The Network Simulator

The network simulator is a device that can imitate the behavior of large data nets. It is used both for the evaluation of protocol behavior in networks with known properties, and for measurements on the 'real-time characteristics' of protocols.

The simulator can introduce several different types of changes in a bit stream: sequencing errors (e.g. changing the order of messages), distortion and mutation errors (changing the contents of a message arbitrarily), as well as deletion and duplication errors. It can further simulate more specific errors of hostile transmission media, e.g. the systematic mapping of messages from one set to those of another set. The following requirements have determined the design of the simulator:

(1) The simulator is protocol independent. It can be programmed for any specific protocol format and procedure.
(2) Every single type of error can be simulated both at the bit level and at higher levels of

abstraction, per bit structure (e.g. a frame).

(3) The error distribution functions are programmable.

Since the simulator imitates network behaviors rather than network implementations, the precise speed at which it operates is irrelevant. In particular, the simulation speed need not be related to the actual transmission speed of the network being modeled. In the device being developed the maximum simulation speed has been set, for practical reasons only, to 64 kbit/sec.

Similarly, the physical implementation of the communication channels to and from the simulator may well differ from those of the network being modeled. The only criterion is that the behavior of the actual implementation can indeed faithfully be reproduced in the model. If, for instance, the transmission lines in the data network being modeled are really time-multiplexed channels, the simulator can be programmed to reproduce the characteristic errors of these channels, while communicating to the host machines (Fig. 1) at a lower speed via error-free RS-232 interfaces.

Note that the approach to protocol assessment described here differs from the one described, for instance, in [1]. The major advantage of the present scheme is that it provides the user with a completely controlled environment in which every single test can be repeated, and in which tests for different protocol designs can be compared.

Protocol assessment by conformance testing against reference implementations [1] would not work in a system like Pandora since reference implementations for new protocols are, and should be, quite rare.

6. Summary

With the design of the Pandora system we have attempted to create a versatile tool for the development of reliable and provably correct communication protocols. The system provides the means to work on nearly every aspect of protocol design: synthesis, formal analysis, and assessment.

Even though the design goals for the system are quite ambitious, the system itself is far from complex.

The uninitiated user can design a first protocol on Pandora with minimal effort, since there is no need to know anything about its inner workings; in particular, the user does not have to know anything about the validation algebra or the assessment techniques. Those who do need to know can understand the inner workings with relative ease, since all software is written in a higher level language (the C language, native to Unix environments).

Work on the Pandora system started in March 1982. At the time of writing, all software except the part to be used for protocol assessments (pass) is operational. For the time being, though, the system is run on just a single PDP computer with a prototype version of the network simulator. The full system is scheduled to be completed within the next year.

Acknowledgement

The protocol validation algebra and the protocol analyzer (pan) were developed while the author was with Bell Laboratories, Murray Hill, N.J., USA. The design and construction of the Pandora system was made possible by a grant from the Netherlands PTT, under contract number 45155 OCA.

Appendix: Meta-language Grammar.

The following defines the syntax of the Pandora protocol specification meta-language. Names given in uppercase (PROC) and quoted strings (':::'), are keywords: "tokens" to the lexical analyzer. The uppercase is used for clarity in this overview; it is optional in actual specifications.

The rules are listed in alphabetical order. The term on the left-hand side of each rule is defined by the right-hand side of the same rule. Alternatives on the right-hand side of a rule are separated by vertical bars: '|'.

A complete syntactically correct specification should conform to the description for "prot_spec."

tokens: IF FI DO OD PROC REF TIMEOUT DEFAULT GOTO BREAK SKIP END

prot_spec:	module_specs '.'
annotation	/* ... any string ... */
comment	annotation | includedcode
cycle:	DO options OD
flag:	':'
includedcode	%% ... any C code ... %%
jump:	GOTO labelname | BREAK
labelname, msgname, procname, taskname:	'text string'
module_spec:	task_spec | proc spec | comment
module_specs:	module_spec | module_specs separator module_spec
oneoption:	flag sequence
options:	oneoption | oneoption options
proc_spec:	PROC procname sequence END

```
recv:          TIMEOUT | DEFAULT | procname '?' msgname
select:        IF options FI
send:          SKIP | procname '!' msgname
separator:     ' → | ';'
sequence:      stmnt | sequence separator stmnt
stmnt:         select | cycle | send | recv | taskname
               | labelname ':' stmnt | jump | comment
task_spec:     REF proname ':' taskname sequence END
```

A "select" statement, for instance, should be written as a keyword IF followed by a set of "options" and should be terminated by the keyword FI. Two different symbols can be used as statement separators: the traditional semi-colon ';' and the arrow ' → '. The latter symbol is often used in "select" and "cycle" statements to separate incoming messages from the corresponding responses (see below).

To illustrate the rules we conclude this appendix with an annotated program-like specification of the example protocol described algebraically in section 3, and represented graphically in figure 3. The example below illustrates how protocols can be designed and analyzed step by step. In an initial design phase the protocol can be described with minimal detail as shown here, not more than a rough sketch, to be checked merely on its logical consistency. In a later phase the description can be extended to a more complete design, supplemented with included code.

```
proc A                      /* spec of process A          */
        B!m;                /* send message m             */
        do                  /* repetition task            */
        :: B?r → B!m        /* fail; retransmit           */
        :: B?r → break      /* success: break loop        */
        od;                 /* end of loop                */
        B!e                 /* end of transmission        */
end;                        /* end process module         */
proc B                      /* spec of process B          */
        example             / * reference task call      */

end;                        /* end process module         */
ref. B: example             /* reference task spec        */
        A?m;                /* receive message m          */
        if                  /* nondeterm. select.         */
        :: A!a              /* positive acknowl.          */
        :: A!r              /* negative acknowl.          */
        fi;                 /* end selection task         */
        A?e                 /* end of transmission        */
end.                        /* end of specification       */
```

The design error discussed in section 3 is traced by the protocol analyzer 'pan' as a sequence of message exchanges, starting in the initial system state (all processes idle, all message queues empty):

A → B: m
B → A: r
A → B: m

The last message is responsible for the error. The 'B' process is expecting and 'e' message from 'A' but it receives an 'm'.

References

[1] K.A. Bartlett, & D. Rayner, The certification of data communication protocols. Proc. IEEE Symp. on Computer Network Protocols, Washington, DC, (Jan. 1980), pp. 12–17.

[2] A. Rockstrom, & R. Saracco, SDL – CCITT Specification and description language. IEEE Trans. on Comm., Vol. COM-30, No. 6, (June 1982), pp. 1310–1318.

[3] CCITT Yellow Book, Vol. VI, Fascicle VI.7, (SDL) Recommendations Z.101 – Z.105, Int. Telecomm. Union, Geneva, Switzerland, (1982).

[4] P. de Chazal, Necessary enhancements to SDL for use as a functional description technique. SDL Newsletter (C-CITT), No. 3, (May 1982), pp. 51–59. (contact Ed. R. Saracco, CSELT, V. Reiss Romoli 274, 10147 Turin, Italy.)

[5] G.J. Holzmann, A theory for protocol validation, IEEE Transactions on Computers, Vol. C-31, No. 8, (August 1982), pp 730–738.

[6] G.J. Holzmann, Concise description of a protocol validation algebra, Report nr. 38, AVS Lab 10.28, Dept. Electrical Engineering, Delft University of Technology, The Netherlands, (June 1982).

[7] G.J. Holzmann, The Pandora System: reduction techniques, Report nr. 43, AVS Lab 10.28, Dept. Electrical Engineering, Delft University of Technology, The Netherlands, (Dec. 1982).

[8] T. Ideguchi, T. Mizuno, & H. Matsunaga, Automatic implementation of communication protocols. ACM Computer Communication Review, Vol. 11, No. 1, (Jan. 1982), pp. 40–56.

[9] H. Marxen, B. Muller-Zimmerman, & S. Schindler, The OSA project: the RSPL-Z compiler, Proc. IEEE Int. Conf. on Communication, Denver Co., USA, (June 1981), Vol. 1, pp. 9.1.1–5.

[10] P.M. Merlin, Specification and validation of protocols, IEEE Trans. on Communications, Vol. COM-27, No. 11, (Nov. 1979). pp. 1761–1780.

[11] L. Pouzin, & H. Zimmerman, A tutorial on protocols. Proceedings of the IEEE, Vol. 66, No. 11, (Nov. 1978), pp. 1346–1370.

[12] J. Rudin, & C.H. West, An improved protocol validation technique. Computer Networks, Vol. 6, No. 2, (May 1982), pp. 65–73.

[13] G.D. Schultz, D.B. Rose, C.H. West, & J.P. Gray, Executable description and validation of SNA. IEEE Trans. on Communications, Vol. COM-28, No. 4, (April 1980), pp. 661–677.

[14] C.H. West, General technique for communication protocol validation. IBM J. Res. Develop., Vol. 22, No. 4, (July 1978), pp. 393–404.