THE PANDORA PROTOCOL DEVELOPMENT SYSTEM

Gerard J. Holzmann

Rob A. Beukers

Delft University of Technology
Dept. EE, Mekelweg 4, Delft
The Netherlands

ABSTRACT

In a joint project with the Netherlands PTT, the Delft University of Technology is developing an interactive protocol design and analysis system named 'Pandora'. The system provides users with a controlled environment for protocol synthesis and formal analysis, and offers both software and hardware tools for protocol assessment.
Pandora can assist the user in the documentation of protocol designs by autonomously extracting SDL−diagrams, and has a set of tools for the generation of executable protocol implementations from abstract specifications.

Proc. 3rd IFIP PSTV Conf, Zurich, Sw., June 1983, North−Holland Publ, pp 357−369.

1. Introduction

We usually define a 'protocol' as set of procedures that controls the exchange of information between concurrent processes. In this view, protocol development is a control problem, and as we all know the protocol designer needs all the help that he or she can get to solve it properly.

There may, however, be a better way to design protocols. The design of a protocol is nothing more or less than the design of a 'language' for information exchange. And, the implementation of a protocol, i.e. the design of a protocol machine that interprets this language, is very much like the design of a 'compiler' for the protocol language. So, in an attempt to reduce the relatively new problem of protocol design to a 'previously solved problem' we may try and learn something from compiler designers.

A protocol can be defined [10] as a triple (I, S, V) where

- I is the protocol 'interaction grammar,' (the 'procedure rules' of a message exchange),
- S is a specification of 'message format,' (a syntax definition), and
- V is the 'protocol vocabulary,' (the finite set of message types that can be transmitted or interpreted if received).

For each new level in a protocol hierarchy we can define a new triple (I, S, V). In a character−oriented protocol, for instance, at a low level of abstraction we can define the message vocabulary V as a set of precisely 258 elements: the 256 8−bit character codes, plus two elements that represent character 'headers' (start bits) and character 'trailers' (stop bits). Subsequently, we can define the encoding of all message elements in a syntax definition S. A complete, though trivial, 'protocol' definition at this level would then also specify an interaction grammar I, which in this case could stipulate the right order in which the data, start, and stop codes are to be transmitted and interpreted.

The grammar, syntax and vocabulary defined by a protocol together make up the definition of a 'protocol language': the language to be interpreted by the communicating processes.

Fortunately, there are some very good tools that can help a compiler writer to construct an efficient lexical analyzer − parser combination, and it may well be that we can use the same tools for protocol design.

These observations form the basis of the Pandora protocol development system. In the Pandora system protocol designing is treated as a special case of compiler designing and the tools for protocol development are derived from the known tools for automated compiler development.

First of all, we can use tools like the Unix−programs YACC and LEX [6,8] to quickly build an efficient compiler for a new protocol specification language. But, we can also try to generate YACC and LEX input specifications for the automatic implementation of protocols as such. The protocol itself is then seen as a compiler for the 'protocol language' as defined above.
To avoid confusion, note that the protocol 'specification language' is the language that describes the protocol; the 'protocol language,' however, is the language described by the protocol itself.

A general overview of the Pandora system is presented in [5]. In this paper we will discuss some new features of the system now being developed at the Delft University of Technology. First, in section 2, we will give an overview of the tools that make up the programming environment of the Pandora system. In section 3 we discuss the construction of the 'network simulator,' a device being built for the evaluation and comparison of protocol designs. Section 4 illustrates the techniques discussed by some examples, and section 5 concludes the paper.

2. Programming Environment

The software for the Pandora system is embedded in a Unix programming environment (Unix is a trademark of Bell Laboratories). Special care has been taken to maintain portability across Unix systems. All software is written in C, and all system files, including those used in formal analyses, are clear−text files that can be accessed and edited by users without restrictions.

Figure 1 gives an overview of the tools available on the system. Everything below the dotted line is visible to users: most notably the plotter (HP−P), the network simulator (NESI), and the graphics display. Everything above the dotted line is internal and may be ignored by the user. Examples of 'internals' that are normally of little concern to the user are the files created by the protocol compiler 'Proco' with conversions of the protocol meta description for the background processes 'Pan', 'Plot' or 'Pass' (see below).

The user's main entry point to the Pandora system is the protocol editor of new protocols into the system, and subsequently have them analyzed, documented and, if all goes well, have them implemented. In the process of designing a new protocol the user can get feedback from the system in the form of different types of reports, such as the proco−, pan− and pass report.

2.1 Prosy: the protocol editor

The protocol editor is programmed as an extension of the standard Unix text editor 'ed.' The extension, named 'Prosy,' is invoked as a simple 'front end' process, that communicates with the Unix editor via pipes. The front end filters user−commands, expands them where abbreviations are used, and interprets them independently if a special protocol synthesis command is recognized. These protocol synthesis commands are run as background processes in the editor. In the Prosy editor these special commands normally result in a call to the protocol compiler 'Proco,' and via Proco to one of the following programs: 'Pan,' 'Plot' or 'Pass.' The function of these programs is discussed below.

2.2 Proco: the protocol compiler

The protocol compiler of the Pandora system has a number of important tasks. Other than a conventional compiler, its main job is not to convert programs into code, but to provide the user with an elaborate compile time analysis of his partial or complete protocol designs. The protocol compiler, therefore, is mainly used as a tool for 'synthesis guidance.'

Apart from the usual search for syntax and grammar errors, the compiler can give an overview of the structure of a protocol (in numbers of processes, procedures, states, and messages exchanged), it can provide a signal list, and it will perform extensive checks for protocol completeness.
The protocol compiler can be used in three modes of operation: warnings for incompleteness), and 'silent' (suppress warnings and list only errors).

More or less as a side−effect, Proco can also convert the protocol specification into alternative representations. Instead of just a single option for code generation, the compiler can offer a range of conversions of

the input language.

The main objective of the system is to use the protocol compiler to generate a compiler−like formal description of the protocol, which can be picked up by a compiler generator such as YACC [6] (section 1). For the time being, though, we use Proco to convert the protocol specification directly into C−programs that implement the protocol [cf. 9]. To accomplish this Proco uses default library routines that can link the protocol into a specific hardware environment. The user can override the defaults by extending the meta−specification with portions of 'included code' (in the language C).

For the logical analysis of a protocol Proco can convert a specification to the protocol expressions used by Pan (section 2.3). Finally, Proco can also convert specifications into flowcharts (section 2.4). An extension of Proco, for the translation of abstract specifications into transition tables, is in preparation.

2.3 Pan: the protocol analyzer

The analysis methodology used in the Pandora system is based on a validation algebra derived from the theory of regular expressions [3,4].

Pan implements and automates the analysis of a number of general correctness criteria. Clearly, the disadvantage of a general tool like Pan is that we can only analyze protocols for properties to which they should all conform, such as absence of deadlock or completeness. Protocol specific properties will have to be analyzed by hand, for instance with the same algebraic method as used by Pan, or with the aid of special−purpose programs.

The protocol expressions used in the analysis are generated by the protocol compiler Proco. Again, the input and output files of Pan are clear−text files, which can be inspected by users, e.g. to debug the analyzer or the compiler.

The main function of the validation is to find inconsistencies in the interaction grammar of the protocol definition (section 1). Pan will trace deadlocks and potential timing problems, will tag unexecutable protocol parts, and report all cases of incompleteness that cannot be detected at compile time.

For an introduction to the algebra and the validation process we refer to [3].

2.4 Plot: protocol documentation

The program 'Plot' can be used to document a design on either a hardcopy plotter (we use an HP−7221C plotter) or a graphics display. When the user calls this command while in the editor, 'Prosy' (section 2.1) will first call 'Proco' (section 2.2) to convert the description in the protocol specification language to a description in a 'plot language' (see below). Then the program 'Plot' is called to interpret this description and draw the chart.

Extracting a flowchart, or an SDL−like [1,2] description of a protocol from the specification is a normal conversion of one language to another. The input language in this case is the protocol 'specification language.' The output language specifies plot figures. In the Pandora system we use a simple language, comparable to 'PIC' [7] that was extended with a standard library for flowcharts and SDL diagrams.

At the lowest level in this plot language we have commands such as with defaults for size and direction if none are given. One level higher there is a small set of predefined symbols such as 'decision,' 'task,' and 'state,' and patterns like 'fork N' (to split a control flow into N branches) and 'join' (the reverse of 'fork').

Within certain limits, the scale of the generated figure can be adjusted to the size of the paper (hard−copy output) or the size of the computer display. Fortunately, it is good practice to divide program specifications into modules (procedures) that can be plotted separately. With the available tools on Pandora it is certainly possible to specify a protocol that cannot completely be plotted, even at the smallest readable scale. The documentation tools will force the designer of such a monolith to divide his design into parts that can be documented.

Most problems with the layout of the figures can be avoided if a few simple rules are followed. For instance, control−flow lines implied by goto−jumps are not plotted; instead jumps and labels are indicated by name in the charts. Branches, for selections or loops, are plotted alternatively to the right or to the left of the main control−flow line. In cases where these rules cannot prevent conflicts in the layout the user can either edit the protocol specification language description, e.g. add a dummy statement to change the

proportions, or edit the plot language description of the figure generated by proco.

2.5 Pass: protocol assessment

Protocol assessment, the evaluation and comparison of protocol behavior in networks, is done with the aid of a 'network simulator.' This device can be programmed to imitate the characteristic errors of a given data network. The interaction with the network simulator is dedicated to the program 'Pass.' For the assessments 'Pass' uses the protocol implementations generated by 'Proco.' A description of the simulator being developed for inclusion in the Pandora system is given in section 3.

3. The network simulator

3.1 Purpose of the simulator

For the evaluation of protocol behavior a hardware device is being built that can be programmed to imitate the error characteristics of large data networks. Unlike the networks modeled, though, the network simulator's behavior is completely reproducible. In this section we will discuss some of the design goals we have set for this network simulator.

One of the main requirements for the design of the simulator is that it be both protocol and hardware independent.
It may seem odd to require that a hardware device (the simulator) that is meant to simulate other hardware devices (networks) be hardware independent. The simulator, however, is meant to imitate behavior rather than implementations. In other words: it mimics 'symptoms,' not 'causes.'

The simulator is freely programmable, which means that we can program it for the simulation of any network, and for the format of any protocol subjected to that network. It is interfaced to the real world via a small set of standard medium speed ports. In simulation experiments each port has the function of a logical channel. Since we do not model implementations but behaviors, we circumvent all problems that would exist if we wanted to reproduce the working of the almost infinite range of physical channels used in data networks.

The simulator can introduce several types of errors in a sequence of transmitted 'data objects' (see section 3.2). It can introduce the delay, distortion, duplication, and deletion of objects, and it can model the systematic mapping of data objects from one class to those of another. The error distribution functions are under software control. The least interesting case is, of course, when all all error functions are turned off. In that case the simulator acts as a transparent link, modeling an error−free channel.

3.2 Data representation

The 'objects' transmitted through the network simulator can be any kind of predefined structure: bits, headers, CRC−checks, frames, packets, etc.

As seen by the simulator, the input stream is a sequence of objects interpreted according to a user−defined hierarchical structure: the message format definition. Each level in this structure consists of data objects constructed from the objects defined at the lower levels in the hierarchy. The lowest level objects are the signals to and from the I/O ports. Errors in the input stream can be introduced at two distinct levels: in the 'raw' uninterpreted data (e.g. at bit or byte level), and at a higher level of abstraction per user−defined entity.

The simulator receives the incoming stream of data objects, and decides, with the help of programmable error distribution functions, if an error is to be introduced. The stream of objects is then modified and passed on to an output port.

3.3 Implementation

An experimental version of the network simulator has been coded as a normal Unix process, linking the communicating processes via a complex of 'pipes.' Eventually, though, the network process will have to be run on dedicated hardware, connected to the Unix host machine via standard I/O ports, as described above. The network simulation process itself again consists of an automatically generated interpreter program. This interpreter is used to recognize the objects of each level in the predefined hierarchy (the user−

defined ‘message format’) and to effectuate the required modifications of the objects at each level.

The control of the simulator is dedicated to the program Pass (section 2.5). Pass will also take care of building a script to generate the interpreter, for which it uses a conversion of the protocol specifications that is generated by Proco (section 2.2).

4. Discussion

4.1 Current Status

At the time of writing, March 1983, most of the tools for protocol analysis and protocol synthesis described above have been completed. Most tools for protocol implementation and protocol assessment, including the network simulator, are still in an early stage of development.

The protocol analyzer Pan has been extended for application in the Pandora system with a small number of features that cannot all be founded easily within the validation algebra [3], but do enhance the value of an analysis considerably. Three of these changes will be discussed below: the use of multiple buffers, the addition of goto’s to the specification language, and the inclusion of a method for loop detection in the analyzer.

Multiple buffers

In early versions of the protocol analyzer Pan, protocol specifications were restricted to the use of a single message queue per process. The algebra of protocol expressions, as presented in [4], does not have this restriction. So, to allow for a more faithfull modeling of also layered protocols, with at least two distinct communication channels (e.g. one to each of the two layers N+1 and N−1 surrounding it), the analyzer was rewritten to remove this restriction. The maximum number of independent message queues per process has now, quite arbitrarily, been set to 10.

Goto’s

Many existing protocol definitions are written in a state, rather than control−flow, oriented manner. Such protocol definitions are most naturally formalized in transition tables, or equivalently in program form with the aid of state labels and goto jumps. There is no operator in the validation algebra that corresponds to a jump, but clearly, these jumps are just another way to specify time orderings.
To allow for a concise specification of transition−table−like protocol specifications we have therefore included the goto statement in the meta−specification language and adapted the analyzer to interpret it properly.

Loops

It was noted in [3] that infinite cycles in protocol executions can present problems in an automated analysis. Especially with the addition of the goto the creation of loops is hard to avoid.
In the protocol analyzer Pan, these cycles are encountered as uncontrolled expansions of protocol expressions [4]. Unfortunately, to recognize all different types of such loops solely by algebraic means turns out to be a hard theoretical problem.
Little was needed, though, to detect the infinite cycles at runtime in the analyzer. Therefore, pending the further development of the theory, the program Pan has been extended with these runtime traps.

4.2 Specification Language

A number of important features of the protocol specification language used in the Pandora system can be illustrated by some small examples.
Consider the following partial description of an imaginary protocol:

```
state6B:
        if
        :: dce?q
        :: dce?q −> goto state07
        :: dce?m −> goto state19
        :: dce!b −> goto state21
        fi;
state10:
        ...
```

First a word about notation. The term ‘dce?q’ indicates that the process described here can receive a message type the process can send message ‘b’ to process ‘dce’.
The if ... fi construct specifies a range of options, one of which will have to be selected. As in the above case, the choice may be indeterministic: there are two entries labeled ‘dce?q’. The first statement after the ‘::’ flag that separates the options is called a ‘guard’. The option that follows the guard is selectable only if the guard is executable. Other than send statements, the receive statements can, of course, only be executed if the appropriate type of message has arrived.

The following protocol fragment shows a loop construct:

```
state01:
        do
        :: timeout −> dce!b; dce!a
        :: dce!i −>
                if
                :: dce?m −> goto state19
                :: dce!a
                fi
        :: skip −> break
        od;
```

The options of the do .. od statement are selectable as in the if .. fi statement discussed above. After an option has been completed, however, the do statement is reentered from the top. The execution only stops when a

The above example illustrates a proper nesting of statements, and also shows two new types of guard: ‘timeout’ and ‘skip’. The ‘skip’ statement is a null statement: it has no effect and can always be executed. In the example it was used to indicate that the loop can be broken at any time. The ‘timeout’ statement can be executed only when the message queue of the process described is empty.

Included Code

In the following fragment the use of deterministic selection and included code is illustrated:

```
%%
msg.type = extract(lastmsg)
%%;
if (msg.type)
:: skip        /* ignore */
:: other!accept −> CALL
:: other!reject −> CLOSE
fi;
other!done
```

Included code, any length of program code preceded and followed by a ‘%%’ sign, is ignored by the analyzer pan, but linked into the protocol implementations generated by proco.
Included code may appear anywhere where a statement in the Pandora specification language may appear, and it is syntactically equivalent to a statement (hence the semicolon after the second %%). In this case the included code consists of a single assignment of the return value of procedure call ‘extract’ to variable

‘msg.type.’ The same variable ‘msg.type’ is also used in the optional condition field of the if statement that follows. Again the analyzer ignores the condition (as it ignores all local computations) and proceeds as if none was given. The compiler, however, uses the condition for the generation of a deterministic selection in the implementation.

The condition may be any statement that returns a value. The value returned is used to index the range of options, using a simple switch statement. In the above case: if the value is anything other than 1 or 2 the selection is skipped entirely (the skip for value 0 is explicit). If the value returned (in this case the value of variable msg.type) is 1 a macro CALL is invoked; if the value is 2 another macro CLOSE is called. In either case execution continues with the transmission of message ‘done’ to process ‘other.’

Specification of multiple queues

The specification of the message queues from which messages are to be collected, or to which messages are to be sent, is again an optional part of the interaction statements. By default, messages are addressed to a queue labeled ‘0’. The statement ‘dte!mesg:2’ would specify that queue ‘2’ is addressed instead of ‘0’. Similarly ‘dce?mesg:2’ specifies that message type ‘mesg’ from process ‘dce’ is to be collected from the queue labeled ‘2’.

4.3 Synthesis Guidance

If we would boldly enter the above protocol fragments into the Prosy editor, we may expect the following guidance. At first, the only thing the protocol compiler Proco can do is to flag syntax errors: the specification is not in the format expected. At the very least the specification should begin by giving the name of the process being specified.

Our reaction to this information from Proco can be to add the line ‘proc dte’ at the top of the fragment, and a line with a term ‘end.’ at the bottom. Now the specification is syntactically correct, but clearly incomplete. In the report compiled by Proco a number of warnings will appear:

1. process ‘dte’ is tagged as an ‘isolated process,’ since no other process in this specification is communicating with it;
2. all messages to be sent by process ‘dte’ are listed as ‘sent but not received’, and similarly
3. all messages to be received by process ‘dte’ are listed as ‘received but never sent’;
4. process ‘dce,’ which is named in the specification as the potential source and destination of ‘dte’s’ messages, is listed as ‘missing.’ ... etc.

Especially for larger, more complex, protocols, these hints and overviews prepared by Proco can quickly direct the user towards flaws and isolated spots of incompleteness in a design, even before the more profound analysis of Pan is required.

5. Conclusion

All programs used in the Pandora system, including the analyzer, need less than 64k of memory to run. The Pandora system itself is implemented on a small mini−computer (a PDP 11/24) without a virtual memory system. The protocol compiler and the analyzer are medium−size programs, of about 1200 lines of C−code, which can be understood with relatively little effort. All other programs are much smaller than this. The Prosy editor, for instance, could be restricted to a mere 100 line program, simply because it enhances rather than replaces the standard Unix editor.

Still, all tools discussed in this paper would be quite useless if they were not accessible to even casual users. To realize this goal, much attention has to be given, not to hiding details, but to confining them to places where they are really needed. Our effort will be to preserve this characteristic of the system when it is completed.

Acknowledgements

The design and construction of the Pandora system was made possible by a grant from the Netherlands PTT, under contract number 45155 OCA.

References

[1] A. Rockstrom & R. Saracco, SDL − CCITT Specification and description language. IEEE Trans. on Comm., COM−30, 6 (June 1982) 1310−1318.

[2] CCITT, Yellow Book VI.7, Recommendations Z.101 − Z.105 (Int. Telecomm. Union, Geneva, 1982).

[3] G.J. Holzmann, A theory for protocol validation, IEEE Transactions on Computers, C−31, 8 (August 1982) 730−738.

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

[5] G.J. Holzmann, The Pandora System: an interactive system for the design of data communication protocols. To appear in: Computer Networks (1983).

[6] S.C. Johnson, YACC − yet another compiler compiler, Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA (1975).

[7] B.W. Kernighan, PIC − a crude graphics language for typesetting, Computing Science Technical Report 85, Bell Laboratories, Murray Hill, NJ, USA (1981).

[8] M.E. Lesk, LEX − a lexical analyzer generator, Computing Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, USA (1975).

[9] H. Marxen, B. Muller−Zimmerman & S. Schindler, The OSA project: the RSPL−Z compiler, Proc. IEEE Int. Conf. on Communications 1 (Denver, Col., USA, June 1981) 9.1.1−9.1.5.

[10] J. Puzman & R. Porizek, Communication control in computer networks (Wiley & Sons, New York, 1980).