Protocol Design: Redefining The State Of The 'Art'

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We are only beginning to discover that the problem of defining unambiguous and logically consistent protocol standards is uniquely challenging and should be considered fundamental. The problem is to devise a method that would allow us to draft protocols that *provably*, instead of arguably, meet their design criteria.

So far, protocol design has mostly been considered a mere programming problem, where skill in design is only related to programming experience. (And where experience is, of course, a nice word for the history of mistakes that a programmer is not likely to make more than once.) Missing in that view is a set of objective construction and measuring tools. As in any engineering discipline, design skill should be teachable as experience in the usage of design tools, rather than mere experience in the avoidance of mistakes.

Before we can solve the protocol design problem in this way, we will need two things:

(1) Adequate specification languages to formalize protocol definitions and design requirements

(2) Effective validation techniques to check requirements against definitions

This paper is about this fundamental problem of protocol design, and it discusses how far we have come in solving it.

An abbreviated version of this paper appeared in IEEE Software − Jan '92

October 20, 1991

Protocol Design: Redefining The State Of The 'Art'

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

An important part of the early work on formal methods in software design has been focused on protocol design problems. One of the reasons for that is that the design of even small protocols can be deviously complicated. The benefit of this early work is that the most impressive results of the application of formal methods have been achieved in precisely this field. There is, however, much confusion about the proper usage of the term 'formal method.' The term has been used to refer to almost any notation that is merely unambiguous and formally defined. Before discussing the protocol design problem specifically, let us therefore first state more precisely what we understand a formal design method to be and how it differs from a more traditional one.

A Traditional Design Method

A traditional design cycle, as used in industrial environments, is usually split into at least three phases

1. High level design
2. Low level design
3. Coding and Testing

In the first phase, the problem requirements are explored and tentative solutions are written down informally, e.g. in English or pseudo−code. The code is reviewed and discussed among peers, often in a very disciplined manner in code walk−through sessions attended by experienced designers.

In the second phase of the design, some of the details are filled in until a fairly complete framework for the final code has been constructed. Then the work is divided and the high and low level design decisions are converted into a programming language and tested for correctness.

Every error uncovered in the testing, or 'debugging,' phase forces a return to coding and perhaps low level design. It is no real surprise that most large scale design efforts tend to spend the majority of the time in this third and last phase. Coding and debugging thus become the focus of the design process, a phenomenon that must be familiar to anyone who has written a computer program.

Why is this the case? There are two main reasons, strictly related to the specification style used and the validation tools that are available. Notice that the third phase of the traditional design process is the first where a formal and *unambiguous notation* is used for representing design decisions: the target programming language. The third phase is also the first one where *tools* are available to verify that a description complies with design requirements. A compiler is used to check the syntactical requirements of the formal notation. Testing runs are used in an attempt to check the semantical requirements of the design.

It is also well−known that the later in the design process an error is detected, the more expensive it is to fix. A logical inconsistency that can be trivially corrected during a high level design phase can cause unwelcome delay if discovered in the debugging phases, and serious financial loss if discovered after the installation of a software product in the field. To make matters worse, the errors that are introduced in the first design phase tend to be the ones that are the most difficult to detect or repair in the last phase.

A More Formal Design Method

The effort to formalize the design process can be understood as an effort to introduce methods that allow the designer to verify the correctness of design decisions at the point where they are made: starting with the first design phase. We want to allow designers to verify the correctness of a high level design before it is committed to (and obscured in) low level code. To realize that goal we need to give the high level design

phase the same two characteristics that are now unique to the coding and debugging stages:

1. An unambiguous notation, and
2. An effective validation tool

The validation tool can be used to validate the syntactical and semantical correctness of the descriptions. In more familiar terms: the validator checks the logical consistency of high level specifications; it checks designs against design criteria. A design method that has these characteristics is what we shall call a 'formal method.'

Design Criteria

The design criteria that a user can define as part of a formal design method can include

- Absence of any unwanted behaviors of a system, such as the infinite repetition of actions that do not meet a progress criterion defined by the user
- Guaranteed observance of a specific functionality, e.g., within finite time after lifting the receiver the telephone subscriber must receive a dial−tone.

These first two types of design criteria are called logical correctness criteria. They are often referred to as safety and liveness conditions, after Lamport [1977]. Some logical correctness criteria can be predefined. Examples of such criteria are absence of system deadlock scenarios, the absence of unspecified receptions, of buffer overflow, etc. In general these safety criteria state that the design may not be either overspecified or underspecified. Underspecification, or incompleteness, causes such errors as unspecified receptions: a scenario in which a protocol process receives inputs for which no responses were defined. Overspecification is responsible for unexecutable code segments in a protocol definition: dead code. Clearly, both under− and over−specification is undesirable, quite independent of the specific protocol being designed.

The second class of correctness criteria is protocol specific. This class contains the particular design criteria that must be formalized explicitly before any reasonable correctness check can be performed. The notation in which such correctness criteria are expressed is not necessarily the same as the notation in which the actual protocol behavior is defined. We will return to this issue in more detail below.

There is also a third class of design criteria.

- Guaranteed observance of real−time performance requirements

A requirement of this third type could state, for instance, that a dial−tone may not be delayed more than a specific length of time. The methodology available for predicting and/or measuring real−time performance of a protocol, however, is quite different from the methodology that can be used for establishing logical consistency of design rules, see e.g., Schwartz [1987]. Here we focus on just the framework that has been developed for proving logical consistency, not because it is more important, but mainly because the advances that have been made in this area are not nearly as well−known.

Protocol Engineering

In a well engineered system a design *provably* meets the criteria that were set for its logical correctness. This means that validation is very much a corner−stone of the new formal design methods.

Rather oddly, this is not a generally accepted principle in protocol design just yet, not even among those who study formal methods. There are basically two approaches to the design problem, that have been called the ''single language'' approach and the ''dual−language'' approach (Ostroff [1989]).

The first approach includes formal specification languages such as CCS (Milner [1980]), LOTOS (Van Eijk, Vissers, and Diaz, [1989]), Z (cf. Bjorner et al. [1990]), and VDM (cf. Jones [1986]). In this approach only the protocol specification proper is formalized. There is no formal notation for expressing design requirements. The design cycle starts with a high−level abstract view of the solution which is transformed in a stepwise manner into a more detailed solution, proving at each step the formal equivalence of each new specification to the one immediately preceding it. The rationale is that it is relatively easy to write down a logically consistent high−level abstract solution to a design problem, but that errors tend to get introduced when more detailed designs are produced.

The second approach is the one we have described earlier. It is based on the availability of a formal notation for both specifications and correctness requirements, complemented by an effective proof method for

showing that the requirements hold for the specifications, largely independent of the level of abstraction at which that specification is defined.

If both the specification and the requirements language are carefully defined, it can be guaranteed that all requirements that can be expressed are also formally decidable within the proof system. This is, for instance, the approach taken in (Ostroff [1989]) and in (Holzmann [1991a]). In some cases, it is also possible to enhance an existing specification language with a new requirements language and a proof method, even when the original specification language was not designed with the prospect of validation in mind.

An example of this is the CCITT specification language SDL.[1] In SDL, by definition, message channels between processes are unbounded. Even simple safety properties of systems with unbounded channels were proven to be formally undecidable, see for instance Cunha and Maibaum [1981], Brand and Zafiropulo [1983], Apt and Kozen [1986], Reif and Smolka [1988]. However, since no final implementation of an SDL specification could ever be given true access to unbounded message channels (alas, even modern computers have only finite resources), it is quite acceptable to drop the unboundedness assumption when a proof system for SDL is constructed. This is the approach we have taken in the construction of a validation system for SDL that has been in use at AT&T since 1989. SDL was enhanced with a notation for requirements specification that includes, for instance, temporal logic formulae, and it sets bounds on all message channels before correctness proofs are rendered, cf. Holzmann and Patti [1989], Chaves [1991], Holzmann [1991b].

In the remainder of this paper we follow the ‘‘dual−language’’ approach. Critical to the success of this approach are the specification and the validation procedures used.

Specification and Validation

In the last ten years, significant progress has been made in the development of both new formal specifications for high level protocol designs and effective validation procedures for these specifications. Let us briefly look back at the most important milestones that were reached in this development process.

The first automated validation of a communications protocol was attempted by a group at the IBM Research Labs in Zurich in the late seventies, cf. West and Zafiropulo [1978], Zafiropulo [1978], West [1978]. They applied a technique, called a ‘perturbation’ or ‘reachability’ analysis, that had been described earlier by Carl Sunshine in his PhD thesis, see Sunshine [1975]. In a perturbation analysis, a given system state is perturbed systematically to derive all possible successor states, which in turn become the target of further perturbations in a systematic search for error states that continues until all reachable states have been explored.

No high level notation was available at this time of these first experiments to describe a design formally, but even in the absence of such a notation this effort demonstrated convincingly that and automated validation could reveal design errors that expert designers would miss, even after years of study. The victim in this first experiment was a particularly simple protocol, CCITT Recommendation X.21, which made the results of the formal validation all the more poignant. The early implementations of the perturbation analysis, however, suffered from several drawbacks that have since largely been overcome. In the remainder of this article we will discuss the progress that has been made in both formal specification and in the formal validation of protocols, with an emphasis on what this progress can mean to the protocol designer today.

2. Specification

The protocol validation systems of the early eighties required a great deal of human ingenuity in preparing a protocol specification for validation. For every new validation problem that presented itself a substantial amount of protocol−specific validation software had to be written. Since only a handful of people were able to write such validation software, formal validation was a personal art, rather than an engineering skill.

A first major step forward came when experiments with special purpose languages that could be used to specify the input for the personal validation software that was used. Since these were again personal

1. SDL, or Specification and Description Language, was developed within de CCITT over a period of more than fifteen years. Overviews of the language can be found in e.g., Rockstrom and Saracco [1982], CCITT [1988], Saracco, Smith, and Reed [1989].

efforts, the languages that were created had a limited goal and audience. These little languages would be parsed by a pre−processor, translated into the internal format of a validation system, and used to trigger, for instance a perturbation analysis. It dramatically sped up the effectiveness of validations, and prepared the way for the development of more serious formal notations with a more universal appeal.

The three most prominent higher−level specification languages of this type in use today are LOTOS (Van Eijk, Vissers, and Diaz, [1989]), ESTELLE (Budkowski and Dembinski [1987]), and SDL (Saracco, Smith, and Reed [1989]). As noted earlier, the effective verifiability of specifications written in these languages has not been high on the list of concerns of the language designers. Formal design in LOTOS is based on the single language approach, ESTELLE and SDL can in principle be adapted to a dual−language approach, given slightly modified language definitions and an appropriate notation for requirements specification.

Not only the input format (i.e., the protocol specification), but also the output format of the early validation systems could be very hard to interpret correctly. The verdict of these validators could, for instance, be a statement of the type:

> "composite state (12,384,76) is **unstable**"

What is a designer to do with such a result? Fortunately this is an easier problem to solve, once the validation system has been taught to interpret a high−level specification language for its input. It then becomes a mere programming chore to map the notation that the validation system understands to the one the the user understands, i.e., to the formal specification. The output of most validation software today typically consists of a trace−back or events through the high−level formal specification, leading from a user−defined initial system state to an error state discovered by the validator.

Behavior and Requirements

In the dual−language approach, there are two parts of the design that must be formalized: the protocol behavior and the corresponding correctness criteria. There are many formal notations for behavior specification, ranging from extended finite state machine formats to full−fledged programming languages. In many cases, even notations that are not specifically designed with validation in mind can fairly easily be adapted to support a formal design method. We will therefore not dwell on the remaining differences between the specification languages that exist. The specific choice made will largely be a personal preference of the designer, as it should be.

Temporal logic is probably the best example of a notation that is ideally suited for requirements specification. The basic concepts were first explored generally in Rescher and Urquhart [1971], and first applied to the design of distributed systems in Pnueli [1977]. Almost all recent dual−language formal design methods are based in one form or another on notations derived from temporal logic, including the two systems that we have developed, cf. Holzmann [1991a, 1991b].

Here the choice of an appropriate notation is necessarily more based on the capabilities of the proof system than on personal preferences of a designer. It is, of course, attractive to develop a notation in which any correctness criteria whatsoever could be expressed, irrespective of the computational complexity involved in the proof of such requirements. A wiser choice, however, is to allow only a smaller, carefully chosen, subset of correctness requirements that can be effectively verified, and discard the ones that predictably would require computationally intractable proofs. The approach is eloquently illustrated in Manna and Pnueli [1990].

3. Validation

The most serious problem of the original perturbation analysis from the early eighties was precisely its computational complexity. So far, all automated validation methods based on the dual−language approach, are based on an exhaustive inspection of reachable system states, referred to as ‘perturbation’ or ‘reachability’ analysis. Alas, it has been proven that the computational complexity of such a method is also the best one that can be achieved (in formalese: the protocol validation problem is PSPACE−complete, cf. Reif & Smolka [1988]).

The number of reachable states that must be generated can rise very quickly with the problem size. Clearly, the hardware that we can use today for formal validation problems has become more powerful. There has been an increase of at least a factor of ten in both the speed of CPU’s and in the amount of main memory

that is available for automated validation runs. But apart from these very noticeable improvements, we can recognize at least three more fundamental improvements in the design of validation algorithms that have been achieved, within the constraints of what is theoretically possible.

1. On the fly verification
2. State space compaction
3. Partial order semantics

We discuss them one by one.

3.1. On the fly verification

The early implementations of the perturbation analysis were first based on multi−pass algorithms (e.g. the duologue matrix analysis of Zafiropulo [1978]), and later on breadth−first graph traversal algorithms. It was soon discovered, however, that a depth−first graph traversal algorithm, e.g. as described in Holzmann [1985,1990], lends itself to a more efficient implementation. The depth−first graph traversal method has the advantage that all validation work can be performed on the fly, during the generation of the reachable states. The advantages of a one−pass algorithm are clear. First, validation does not have to be postponed until all reachable states have been generated, as was customary in the early implementations. Secondly, and most importantly, the validation of correctness properties need not depend on our ability to effectively generate a full state space. It will probably always be the case that very large problems can only be sampled in an attempt to establish their logical correctness. On−the−fly verification methods have provided a good test−bed for experimenting with sampling techniques. These techniques are known as ‘‘scatter searching’’ or ‘‘partial searching’’ techniques and were first described in Holzmann [1985, 1987a].

More recently, the technique of on the fly verification was also found to be advantageous in the construction of software for circuit analysis problems (Courcoubetis, Vardi, Wolper, and Yannakakis [1990]). This area usually referred to as ‘‘model checking’’. The validation problems are very similar to those in protocol design. The main difference is that in the validation of a hardware circuit the assumption of multiple interacting asynchronously executing processes is often too strong. The circuits is often synchronized by a common clock, allowing an important simplification of the validation algorithms. One of the early papers on model checking is Clarke et al. [1983].

3.2. State space compaction

In 1987 it was discovered that both the speed of a traditional perturbation analysis and its capacity to store reachable states could be improved considerably by a relatively simple programming trick. The method, called ‘supertracing’ or ‘bit−state storage,’ was introduced in Holzmann [1987b]. The main drawback of the technique is a loss of the certainty that all reachable states will have been explored during a search. The statistical probability that any state would be missed can be pushed arbitrarily close to zero, though, even for state spaces that are many orders of magnitude larger than those that can be validated with a classic perturbation analysis. It is especially the latter result that makes the supertracing technique valuable.

The technique has the property that all problems that can be validated exhaustively with a traditional technique (up to roughly 10^5 reachable system states) can be validated one to two orders of magnitude faster with a supertrace algorithm, with an extremely high probability of exhaustive coverage. For problems that are too large for the traditional perturbation techniques (from 10^5 to 10^9 reachable states) the supertrace technique can still provide a sufficiently high probability of exhaustive coverage with the same speed advantage over the traditional perturbation analysis. The effect of the introduction of this technique have been very noticeable. In my own work environment, it made it possible for the first time to experiment with the application of formal validation tools to industrial size problems (cf. Holzmann [1991b]).

Meanwhile the bit state storage technique has been implemented, at least as an option, in virtually all existing validation systems (e.g. in one of the oldest and most inspiring systems, Xesar, developed at the University of Grenoble in France). It is the method employed in AT&T’s SDL validation tool (Holzmann and Patti [1989]), and one of the search options in the SPIN software described in Holzmann [1991a].

3.3. Partial order semantics

The most recent, and potentially the most exciting, improvement that has been achieved in automated validation is the discovery of a formal foundation for the application of partial order semantics for the perturbation analysis (Godefroid [1990], Probst [1990], Valmari [1990]).

In a traditional reachability analysis, reachable states are generated over all possible execution paths of a concurrent system. To generate all these execution paths, the search ‘‘shuffles,’’ or interleaves, the actions of all asynchronously executing processes in every feasible way. When the actions affect the state of shared objects, for instance by sending to or receiving from shared message channels, the results of each different interleaving can be different, and may or may not lead to errors. When two actions are independent, however, the classic shuffling can be woefully redundant. As an extreme example, consider two non−interacting, completely independent processes. A classic perturbation analysis of such a system will blindly explore all possible interleavings of the independent actions, all leading to the same result.

This may seem like gross overkill, but the problems are very subtle. It is fairly hard for a validator to detect accurately where processes are independent and where interleavings can safely be suppressed. Several approximate methods, based on simple heuristics to restrict the number of interleavings that have to be explored, were implemented before sound systems for partial order semantics were described (cf. Gouda and Yu [1984], Gouda and Han [1985], Holzmann [1985], West [1986b]). These heuristics always provably or unprovably carried with them the risk of incompleteness of the validation results. The new theories for partial order semantics that have now become available can be used to reduce the number of interleavings that must be inspected in a completely reliable manner, *provably* without the risk of any incompleteness. The early experiments we have performed indicate that a reduction in the amount of computation that has to be performed during validations of in the order of 60% to 80% can now be achieved.

4. The State Of The Art

We have come far in the development of formal specification methods and formal validation algorithms for protocol design problems. Validation, however, is still by many regarded to be an esoteric art, not the practical engineering discipline it has grown to be. It is certainly not common knowledge that the validation of a small to medium size protocol is a matter of seconds. Nobody needs to open himself up to the embarrassment of publishing or communicating a half−understood algorithm, certainly not professionals in the field of distributed systems. The following anecdote can illustrate this.

An Anecdote

One of the main computer manufacturers in the U.S., that shall blissfully remain unnamed here, recently recommended to a customer to use the following algorithm to enforce mutual exclusion among processes in a distributed system that could try to access a shared resource (in the absence of an indivisible test−and−set instruction.)

The solution is described below in the validation language PROMELA (Holzmann [1991a]). Most language concepts used here will be familiar. The `if` clauses are used for selections, here between two alternatives, based on the truth−value of boolean expressions. The assert statement is meant to verify that the shared resource can only be used by one process at a time (the fundamental requirement of a mutual exclusion algorithm).

```
byte in, x, y, z;    /* shared globals, initial value 0 */

proctype user(byte me)
{
L1:  x = me;
L2:  if
     :: (y != 0 && y != me) −> goto L1    /* try again */
     :: (y == 0 || y == me) −> goto L3    /* proceed   */
     fi;
L3:  z = me;
L4:  if
     :: (x != me)  −> goto L1       /* try again */
     :: (x == me)  −> goto L5       /* proceed   */
```

```
     fi;
L5:  y = me;
L6:  if
     :: (z != me) -> goto L1             /* try again */
     :: (z == me) -> goto L7             /* proceed   */
     fi;
L7:                            /* success */
     in = in+1;                /* count competitors */
     assert(in == 1);               /* access resource   */
     in = in - 1;
     goto L1                        /* repeat */
}
```

The solution was shown to me by that customer, who had trouble getting it to work properly (sic). Those readers who doubt the importance of automated validation tools are encouraged to find the bugs in this design manually. Those who can find all its flaws within five minutes win this argument.

The mutual exclusion problem was first described by Dijkstra in 1965, Dijkstra [1965], but it continues to generate research papers. The problem should be considered solved, ad infinitem even, and its solution should really be part of the working knowledge of everyone working in distributed systems (it generally is).

It takes less then five minutes to type in the PROMELA specification above, and less than five milliseconds to produce all the counter examples that prove its incorrectness. It is not just incorrect, it is woefully incorrect. There are just 223 reachable states in this sample problem (assuming two user processes), which makes it trivial for a machine to verify all possible execution scenarios. For a human being it is virtually impossible to imagine all possible ways in which any one of those 223 states may or may not be reached from any one of the others.

The morale of this anecdote is twofold. The first part is that it can be very difficult to spot the flaws in a distributed algorithm by inspection alone. Even reputable organizations, and very smart people, can make design errors. Since a protocol is no more or less than an algorithm that is to be run on a distributed system, to verify the correctness of a protocol of any size is virtually impossible by human inspection alone.

Secondly, it is sheer folly not to validate a protocol design, or in general any newly design distributed algorithm, with the existing state−of−the art tools, as a matter−of−fact part of the normal design process. The designer who does not use these tools is simply planting time bombs in his or her designs.

Unfortunately, anecdotes like this one abound. It is remarkably easy to pick up any recent journal that contains papers on formal methods, type in some of the examples that are reasoned correct with manual methods, and prove their incorrectness by purely mechanical means. Most people writing protocol validation software also know from experience that virtually no freshly designed protocol can survive the scrutiny of the new automated validation systems. It is still a safe bet that every protocol standard produced by the international committees in the last ten years will likely contain errors of incompleteness or logical inconsistency.

5. Conclusion

The main goal of this paper is to show that there now exist effective formal design methods that can be used to tackle hard protocol design problems. The methods have been made possible by improvements in both the specification of high level designs and of basic validation algorithms. Given that the new methods exist, the only reason that they are not used on a much larger scale is that not enough is known about them yet. Though this is quickly changing, there is some hesitance to adopt formal validation and formal protocol design as full fledged engineering disciplines that can be taught as part of the normal university curricula.[2]

2. For those who shy away from the construction of complete validation systems, general purpose protocol validation software is also available for experimental and educational purposes. The validation software SPIN, described in Holzmann [1991a], for instance, can be obtained free of charge, by sending a one−line message ‘‘send index’’ to the internet destination: netlib@research.att.com Commercial applications require a license from AT&T for a small fee. Contact the author, gerard@research.att.com, for more details.

The term ‘validation engineer’ was recently introduced as a tentative new job−title for the designers at AT&T responsible for the formal validation of new protocol implementations. It is a very encouraging sign that design based on formal methods, is quickly becoming a realistic, common−sense option for even industrial size projects.

Acknowledgement

I share many of the ideas expressed in this paper, and all of the work done on the introduction of formal design and validation methods at AT&T’s 5ESS International Switching Division, with three important people: John Chaves, Joe Lin, and Joanna Patti. It is a pleasure to acknowledge their influence and contribution.

6. References

Apt, K.R., and Kozen, D.Z. [1986], ‘‘Limits for automatic verification of finite state concurrent systems,’’ *Inf. Processing Letters*, Vol. 22, No. 6, May 1986, pp. 307−309.

Bjorner, D., Hoare, C.A.R., and Langmaack, H. (eds). [1990], *VDM and Z: formal methods in software development*, Proc. Third Int. Symp. of VDM Europe, Kiel, FRG, April 1990. Lecture Notes in Computer Science, Vol. 428, ISBN 0−387−52512−0, 579 pgs.

Brand, D., and Zafiropulo, P. [1983], ‘‘On communicating finite state machines,’’ *Journal of the ACM*, Vol. 30, No. 2, pp. 323−342.

Budkowski, S., and Dembinski, P. [1987], ‘‘An introduction to Estelle: a specification language for distributed systems,’’ *Computer Networks and ISDN Systems*, Vol. 14, pp. 3−23.

CCITT [1988], *Blue Book*, Recommendation Z.100, International Telecommunications Union (ITU), Geneva.

Chaves, J.A., [1991], ‘‘Formal Methods at AT&T − an industrial usage report,’’ Proc. Forte’91 Conference on Formal Techniques, November 1991, Sydney, Australia.

Clarke, E.M., Emerson, E.A., Sistla, A.P. [1983], ‘‘Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach,’’ *Proc. 10th ACM Symposium on Principles of Programming Languages*, Austin, Tx.

Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. [1990], ‘‘Memory efficient algorithms for the verification of temporal properties,’’ *2nd Workshop on Computer−Aided Verification*, Rutgers University, New Brunswick, N.J., June 18−20, 1990.

Cunha, P.R.F., and Maibaum, T.S.E. [1981], ‘‘A synchronization calculus for message oriented programming,’’ *Proc. Int. Conf. on Distributed Systems*, IEEE, pp. 433−445.

Dijkstra, E.W. [1965], ‘‘Solution of a problem in concurrent programming control,’’ *Comm. of the ACM*, Vol. 8, No. 9, p. 569.

Eijk, P. van, Vissers, C.A., and Diaz, M. [1989], *The Formal Description Technique Lotos*, North−Holland Publ., Amsterdam, 1989. 451 pgs.

Godefroid, P. [1990], ‘‘Using partial orders to improve automatic verification methods,’’ *Proc. 2nd Workshop on Computer−Aided Verification*, R.P. Kurshan, and E.M. Clarke (Eds.), Rutgers University, Springer Verlag, New York.

Gouda, M.G. and Yu, Y.T. [1984], ‘‘Protocol validation by maximal progress state exploration,’’ *IEEE Trans. on Communications*, Vol. COM−32, No. 1, pp. 94−97.

Gouda, M.G. and Han, J.Y. [1985], ‘‘Protocol validation by fair progress state exploration,’’ *Computer Networks and ISDN Systems*, Vol. 9, pp. 353−361.

Holzmann, G.J. [1985], ‘‘Tracing protocols,’’ *AT&T Technical Journal*, Vol 64, December 1985, pp. 2413−2434.

Holzmann, G.J. [1987a], ‘‘Automated protocol validation in ‘Argos,’ assertion proving and scatter searching,’’ *IEEE Trans. on Software Engineering*, Vol. 13, No. 6, June 1987, pp. 683−697.

Holzmann, G.J. [1987b], ‘‘On limits and possibilities of automated protocol analysis,’’ *Proc. 7th IFIP*

WG 6.1 Int. Workshop on Protocol Specification,, Testing, and Verification, North−Holland Publ., Amsterdam, pp. 137−161.

Holzmann, G.J., and Patti, J. [1989], ‘‘Validating SDL specifications: an experiment,’’ *Proc. 9th IFIP WG 6.1 Int. Workshop on Protocol Specification*, *Testing, and Verification*, North−Holland Publ., Amsterdam.

Holzmann, G.J. [1990], ‘‘Algorithms for automated protocol validation,’’ *AT&T Technical Journal*, Special issue on Protocol Testing and Verification. Vol 69, No 1, pp. 32−44.

Holzmann, G.J. [1991a], *Design and Validation of Computer Protocols*, Prentice Hall, Englewood Cliffs, NJ, 512 pgs, ISBN 0−13−539925−4.

Holzmann, G.J. [1991b], ‘‘Practical methods for the formal validation of SDL specifications,’’ *Computer Communications*, November 1991, Special Issue on ‘Practical uses of FDT’s.’

Jones, C.B. [1986], *Systematic software development using VDM*, Prentice Hall, London, 1986.

Lamport, L. [1977], ‘‘Proving the correctness of multiprocess programs" *IEEE Trans. on Software Engineering*, Vol SE−3, No. 2, pp 125−143.

Lamport, L. [1986], ‘‘The mutual exclusion problem — parts I and II", *Journal of the ACM*, Vol. 33, No. 2, April 1986, pp. 313−347.

Manna, Z., and Pnueli, A. [1990], *Tools and Rules for the Practicing Verifier*, Stanford University, Report STAN−CS−90−1321, July 1990, 34 pgs.

Milner, R. [1980], ‘‘A calculus for communicating systems,’’ *Lecture Notes in Computer Science*, Vol. 92.

Ostroff, J.S. [1989], *Temporal logic for real−time systems*, Research Studies Press Ltd, Wiley & Sons, New York, ISBN 0−86380−086−6, 209 pgs.

Pnueli, A. [1977], ‘‘The temporal logic of programs,’’ *Proc. 18th IEEE Symposium on Foundations of Computer Science*, Providence, R.I., pp. 46−57.

Probst, D.K. [1990], ‘‘Using partial−order semantics to avoid the state explosion problem in asynchronous systems,’’ *Proc. 2nd Workshop on Computer−Aided Verification*, R.P. Kurshan, and E.M. Clarke (Eds.), Rutgers University, Springer Verlag, New York.

Reif, J.H., and Smolka, S.A. [1988], ‘‘The complexity of reachability in distributed communicating processes,’’ *Acta Informatica*, Vol. 25, pp. 333−354.

Rescher, N., and Urquhart, A. [1971], *Temporal Logic*, Springer Verlag, Library of Exact Philosophy, ISBN 0−387−80995−3, 273 pgs.

Rockstrom, A., and Saracco, R. [1982], ‘‘SDL — CCITT Specification and Description Language,’’ *IEEE Trans. on Communications*, Vol. COM−30, No. 6, pp. 1310−1318.

Saracco, R., Smith, J.R.W., and Reed, R. [1989], *Telecommunications Systems Engineering using SDL*, North−Holland Publ., Amsterdam, 633 pgs, ISBN 0 444 88084 4.

Schwartz, M. [1987], *Telecommunication networks: protocols, modeling, and analysis*, Addison−Wesley Pub., 1987, 749 pgs., ISBN 0−201−16423−X.

Sunshine, C.A. [1975], *Interprocess Communication Protocols for Computer Networks*, Ph.D. Dissertation, Dept. of Computer Science, Stanford Univ., Stanford, CA.

Valmari, A. [1990], ‘‘A stubborn attack on state explosion,’’ *Proc. 2nd Workshop on Computer−Aided Verification*, R.P. Kurshan, and E.M. Clarke (Eds.), Rutgers University, Springer Verlag, New York.

West, C.H., and Zafiropulo, P. [1978], ‘‘Automated validation of a communications protocol: the CCITT X.21 recommendation,’’ *IBM J. Res. Develop.*, Vol. 22, No. 1, pp. 60−71.

West, C.H [1978], ‘‘General technique for communications protocol validation,’’ *IBM J. Res. Develop.*, Vol. 22, No. 3, pp. 393−404.

West, C.H. [1986b], ‘‘Protocol validation by random state exploration,’’ *Proc. 6th IFIP WG 6.1 Int. Workshop on Protocol Specification*, *Testing, and Verification*, North−Holland Publ., Amsterdam, pp. 233−242.

Wolper, P. [1986], ‘‘Specifying interesting properties of programs in propositional temporal logic,’’ *Proc. 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Fla., January 1986, pp. 148−193.

Zafiropulo, P. [1978], ‘‘Protocol validation by duologue−matrix analysis,’’ *IEEE Trans. on Communications*, Vol. COM−26, No. 8, pp. 1187−1194. Angela − with bleeding heart (...) i’ve cut the list of references in half now; the ones that remain are below. of course, the appropriate parts of the sentences that refer to the other 20 should be deleted as well from the main text. if possible, we could still try to salvage some of the 20 victims in a history sideline, but you’ll have a better feeling for the feasibility of that than i. hope this helps! −gerard ==========reduced refs=====

7. References

Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. [1990], ‘‘Memory efficient algorithms for the verification of temporal properties,’’ *2nd Workshop on Computer−Aided Verification*, Rutgers University, New Brunswick, N.J., June 18−20, 1990.

Dijkstra, E.W. [1965], ‘‘Solution of a problem in concurrent programming control,’’ *Comm. of the ACM*, Vol. 8, No. 9, p. 569.

Godefroid, P. [1990], ‘‘Using partial orders to improve automatic verification methods,’’ *Proc. 2nd Workshop on Computer−Aided Verification*, R.P. Kurshan, and E.M. Clarke (Eds.), Rutgers University, Springer Verlag, New York.

Gouda, M.G. and Han, J.Y. [1985], ‘‘Protocol validation by fair progress state exploration,’’ *Computer Networks and ISDN Systems*, Vol. 9, pp. 353−361.

Holzmann, G.J. [1985], ‘‘Tracing protocols,’’ *AT&T Technical Journal*, Vol 64, December 1985, pp. 2413−2434.

Holzmann, G.J. [1987a], ‘‘Automated protocol validation in ‘Argos,’ assertion proving and scatter searching,’’ *IEEE Trans. on Software Engineering*, Vol. 13, No. 6, June 1987, pp. 683−697.

Holzmann, G.J., and Patti, J. [1989], ‘‘Validating SDL specifications: an experiment,’’ *Proc. 9th IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, North−Holland Publ., Amsterdam.

Holzmann, G.J. [1990], ‘‘Algorithms for automated protocol validation,’’ *AT&T Technical Journal*, Special issue on Protocol Testing and Verification. Vol 69, No 1, pp. 32−44.

Holzmann, G.J. [1991a], *Design and Validation of Computer Protocols*, Prentice Hall, Englewood Cliffs, NJ, 512 pgs, ISBN 0−13−539925−4.

Holzmann, G.J. [1991b], ‘‘Practical methods for the formal validation of SDL specifications,’’ *Computer Communications*, November 1991, Special Issue on ‘Practical uses of FDT’s.’

Lamport, L. [1986], ‘‘The mutual exclusion problem — parts I and II", *Journal of the ACM*, Vol. 33, No. 2, April 1986, pp. 313−347.

Manna, Z., and Pnueli, A. [1990], *Tools and Rules for the Practicing Verifier*, Stanford University, Report STAN−CS−90−1321, July 1990, 34 pgs.

Ostroff, J.S. [1989], *Temporal logic for real−time systems*, Research Studies Press Ltd, Wiley & Sons, New York, ISBN 0−86380−086−6, 209 pgs.

Pnueli, A. [1977], ‘‘The temporal logic of programs,’’ *Proc. 18th IEEE Symposium on Foundations of Computer Science*, Providence, R.I., pp. 46−57.

Reif, J.H., and Smolka, S.A. [1988], ‘‘The complexity of reachability in distributed communicating processes,’’ *Acta Informatica*, Vol. 25, pp. 333−354.

Saracco, R., Smith, J.R.W., and Reed, R. [1989], *Telecommunications Systems Engineering using SDL*, North−Holland Publ., Amsterdam, 633 pgs, ISBN 0 444 88084 4.

West, C.H., and Zafiropulo, P. [1978], ‘‘Automated validation of a communications protocol: the CCITT X.21 recommendation,’’ *IBM J. Res. Develop.*, Vol. 22, No. 1, pp. 60−71.

West, C.H [1978], ‘‘General technique for communications protocol validation,’’ *IBM J. Res. Develop.*, Vol. 22, No. 3, pp. 393−404.

West, C.H. [1986b], ‘‘Protocol validation by random state exploration,’’ *Proc. 6th IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, North−Holland Publ., Amsterdam, pp. 233−242.

Wolper, P. [1986], ‘‘Specifying interesting properties of programs in propositional temporal logic,’’ *Proc. 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Fla., January 1986, pp. 148−193.