Practical Methods for the Formal Validation of SDL Specifications

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Formal design and validation methods have achieved most of their successes on problems of a relatively modest size, involving no more than one or two designers and no more than a few hundred lines of code. The serious application of formal methods to larger software development projects remains a formidable challenge.

In this paper we report on some initial experience with the application of a formal validation system to SDL design projects involving more than ten people, producing tens of thousands of lines of high–level code over several years. The problems encountered here are large enough that most formal methods break down, for both technical and non–technical reasons.

August 29, 1991

Practical Methods for the Formal Validation of SDL Specifications

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

A traditional design cycle often starts with a rather loosely defined problem statement. The more subtle aspects of the problem and its requirements are often only discovered over time, for instance during the final coding and debugging phases. When new automated validation tools are introduced into the traditional design cycle they run the risk of being treated as a sophisticated form of debugging, and they are placed at the end of the design cycle. A formal design method, however, can only truly impact the quality of designs if it succeeds in moving validation from the *end* of the design cycle to the *start*. This, of course, places demands on the nature of the formal method and on the nature of the design tools that can be build to support it.

The purpose of the introduction of formal methods in the design process is two−fold. It can provide the designer with more powerful tools to produce unambiguous specifications (1) and to perform rigorous validations (2).

1. Formalization requires a precise and a complete specification of the nature and constraints of a design problem. The precision avoids problems of ambiguity in the problem statement, and of miscommunication between the designers whose job it is to produce a solution.
2. The designer should be able to validate the logical consistency of a problem statement with aid of automated tools, and should be able to prove with automated tools that a tentative solution to the problem meets the design requirements.

In almost all formal methods work, only the first goal is taken seriously. The importance of the second goal, however, should not be underestimated. After all, a precise and unambiguous specification can still be logically inconsistent, e.g., include deadlock. Validation has therefore become the focal point of our efforts to introduce formal methods in an industrial environment.

In this paper we report on our experience with the introduction of formal validation tools in a 5ESS® Switch development at AT&T. The specification language used in this environment is the CCITT language SDL, with some small local adaptions. We will indicate under which conditions SDL specifications can be used within a formal method, and we briefly describe some characteristics of the SDL validation tool that we have developed. We will then discuss some technical and non−technical difficulties we have encountered in an ambitious attempt to integrate formal validation tools into a large−scale software development effort.

2. An SDL Validation Tool

The definition of the CCITT's Specification and Description Language (SDL) was developed in study groups SG−XI and SG−X of the CCITT over a period of more than 20 years. The study originally started in 1968. A first version became CCITT Recommendation Z101−Z104 in 1976, and revised versions were published in 1980, 1984, and 1988. A description of the language SDL itself can be found in, e.g., [SDL '87] and need not be repeated here. The language is specifically intended for use in telephone switching applications [e.g., Jilek '84, Lippold '85, Bennett '86], so it is understandable that it has become the first choice of many telephone companies for a formal description technique [e.g., Saracco '89].

It is easy to see that SDL, specifically as it is used in our development environment, does not fully meet the requirements for a formal method. For one thing, validation problems for SDL are not necessarily tractable. Asynchronous message buffers in SDL, for instance, are by definition unbounded. This makes finite

state verification impossible, unless strict upper bounds on the length of each message buffer can be proven or assumed. There is also an SDL language feature that allows task statements to contain arbitrary informal text, from C to poetry. Unfortunately, this feature is heavily relied on in our development area. Clearly there cannot be a method for validating the contents of such arbitrary SDL statements unless some further rules are imposed. The simplest rule that we have imposed is that the code in declarations, tasks, and decisions is compatible with the host language of our SDL validator, a subset of the programming language C. Other rules are meant to guarantee that the SDL systems that are subjected to validation are bounded in the number of processes and the length of message buffers, and are completely specified (there are no hidden traffic sources or sinks).

Though these issues are important, the way in which we have addressed them is fairly standard. The interested reader is referred to [Holzmann & Patti '89] for a more detailed discussion. To avoid confusion, the full definition of SDL does allow for the formal definition of data types, and operations on data objects in task statements [e.g., Saracco '89]. It is expected that in the long run these rules will be adopted and integrated into our validation system.

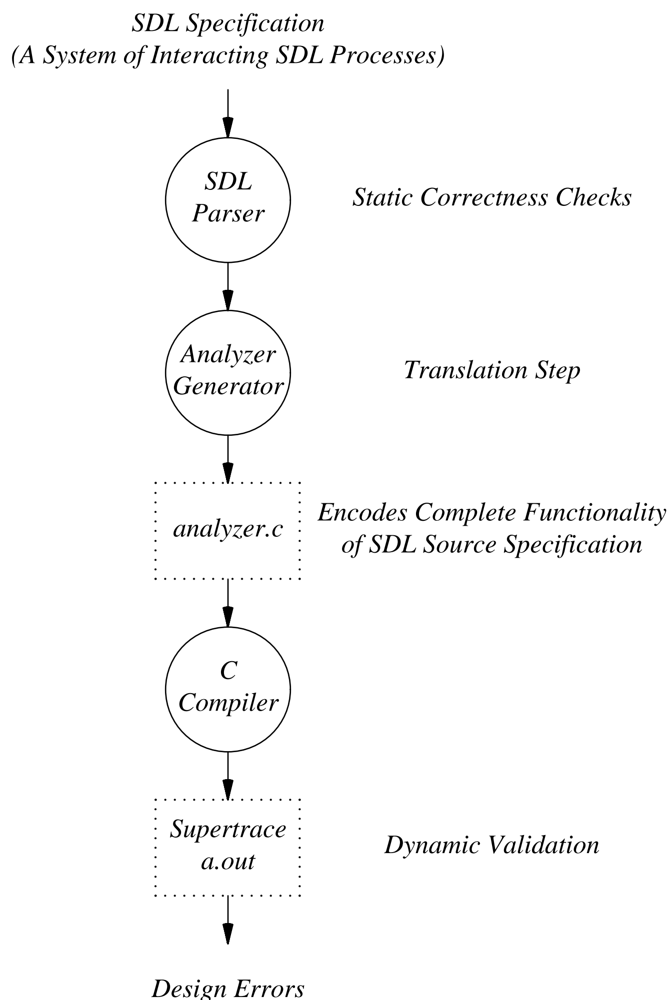

Figure 1 − Structure and Usage of Sdlvalid

Sdlvalid

A first version of our validation tool for SDL specifications was written in 1987. It was introduced into AT&T's switch development environment as an experimental tool in the middle of 1988, and was first described in [Holzmann & Patti '89]. The experimental version consisted of a simple preprocessor connected to a fast reachability analysis tool that had been developed independently [cf. Holzmann '88]. In 1989 the experimental version of the validator was replaced with a more robust tool that directly parses and analyzes SDL specifications. The structure and usage of this tool, named *sdlvalid*, is illustrated in Figure 1.

Some of the design errors caught by this validation tool are

- Incompleteness (e.g., that a message sent by one process cannot be received by at least one process, or that a message received by one process is not sent by at least one process)
- Usage of potentially uninitialized data (using a standard data−flow check)
- System Deadlock (also called deadly embrace, or circular waiting)
- Buffer Overrun (an attempt to send messages to a full queue)
- Unspecified Reception (reception of a message in a state that has no response defined for it)
- Race Conditions (timing or timer errors, hidden assumptions about the relative speeds of concurrent processes)
- Non−progress Cycles (potentially infinite execution sequences in which no, user−specified, progress mark is passed)
- Unreachable Code, Unreceivable Inputs (dead code segments)
- Violations of (user specified) Correctness Assertions (including system or process invariants)
- Violations of (user specified) Temporal Logic Formulae, that formalize claims about correct or incorrect behaviors of the protocol.

The last two validation features of the tool were added last, and are also the most rarely used. With very few exceptions, SDL validation runs are searches for the standard, non protocol−specific, design errors. The first two items are checked statically, during the parsing phase. The remaining items require dynamic checks by reachability analysis.

The tool was installed in January 1990 and has been used on two design projects so far. These design projects are small by engineering standards; they are large from a formal validation point of view. Both projects produced in the order of 10,000 to 20,000 lines of SDL source text, and both involved 40 to 50 people over a period of roughly two years. Only a small number of people of the project teams were involved specifically with the validation aspects of the SDL that was produced. More recently this has lead to the introduction of the appealing new job title ''validation engineer.''

Two overriding concerns in the design of the final validation tool have been user−level simplicity and effective complexity management.

- □ Simplicity: we required that the usage of the validation tool requires minimal training. No knowledge of formal methods or of proof theories is required to perform simple validations, e.g., for absence of deadlock.
- □ Effective complexity management: we required that the tool automatically scales to the maximum capabilities of any hardware that is available for validation, to provide optimal performance for given hardware and problem size constraints, without user intervention.

These two points seem obvious, but they are violated by virtually all other validation methods in use to date.

User−Level Simplicity

Sdlvalid by default checks a protocol design on its completeness and logical consistency. This means that it does not require any guidance from the user for the validator to perform sophisticated searches for absence of deadlocks, unspecified receptions, buffer overflow, and absence of unreachable code. Each error found is reported to the user as a trace−back of events, leading from an initial system state to the state in which the error was detected.

A typical display of a deadlock error sequence is shown below.

```
error: deadlock

process states:
==============
proc phone_system, state: wait_for_first_hangup, line 55, file phone_sys.sd
proc bob         , state: converse, line 25, file bob.sd
proc vickie      , state: converse, line 41, file vickie.sd

queue history:
==============
step 8:  phone_system  ->> vickie       : { answer, 0 }
step 7:  bob           ->> phone_system : { off_hook, 0 }
step 7:  phone_system  ->  vickie       : { answer, 0 }
step 6:  bob           ->  phone_system : { off_hook, 0 }
step 6:  phone_system  ->> bob          : { ring_phone, 0 }
step 5:  phone_system  ->  bob          : { ring_phone, 0 }
step 5:  vickie        ->> phone_system : { dial_digits, 1 }
step 4:  vickie        ->  phone_system : { dial_digits, 1 }
step 4:  phone_system  ->> vickie       : { dial_tone, 0 }
step 3:  phone_system  ->  vickie       : { dial_tone, 0 }
step 3:  vickie        ->> phone_system : { off_hook, 0 }
step 2:  vickie        ->  phone_system : { off_hook, 0 }
```

The first few lines of the error report defines the state of every process at the time of the error, the line number in the source file for that process and the name of the source file. Then, under the default option, the error−trace steps backwards in time, one event at a time, showing the interactions that took place. A single headed arrow means a message transmission here, a double headed arrow means a message reception. At the end of each line, in curly braces, the message type and the value of all message parameters is listed.

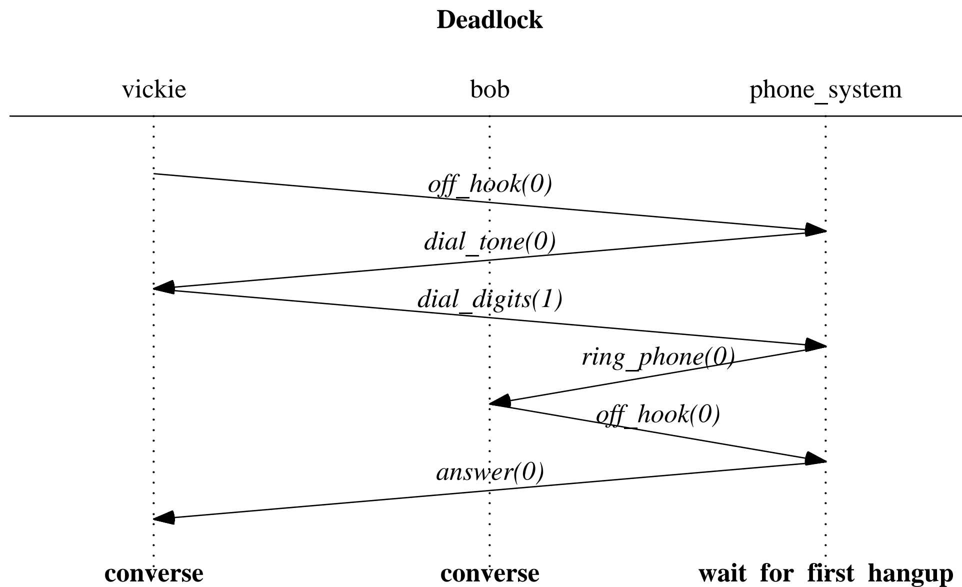

Figure 2 − Graphical Form of Error Trace produced by Sdlvalid

The output format above is the most general version, requiring no special display capabilities. Optionally, however, the same deadlock sequence can also be rendered in graphical form, with events represented by arrows from process to process carrying message identifiers, as shown in Figure 2.

If a default search does not produce anything valuable, the user can choose to perform more detailed validations by ‘‘priming’’ a specification with more protocol specific correctness assertions, for instance to prove preservation of system invariants or absence of non–progress cycles. The final release of *sdlvalid* also added facilities for checking three basic and intuitively clear types of temporal logic formulae introduced in [Manna and Pnueli ’90] as ‘‘... three classes of properties we [...] believe to cover the majority of properties one would ever wish to verify.’’ The three types are named

- Invariance
- Response and
- Precedence

Three simple examples of such claims are, in the format recognized by *sdlvalid*:

```
CLAIM;
    ALWAYS p;                /* Invariance */
ENDCLAIM;

CLAIM;
    p IMPLIES EVENTUALLY q;  /* Response   */
ENDCLAIM;

CLAIM;
    p IMPLIES q UNTIL r;     /* Precedence */
ENDCLAIM;
```

where all capitalized words are keywords, and `p`, `q` and `r` are composite expressions on the state of running processes, message queues, and local or global variables. For example, `p` could be defined as:

(bob INSTATE idle || FIRSTMESG bob IS off_hook && Cnt == 4)

etc.

All validations are rendered in a similar, quiet, unobtrusive fashion with minimal interaction from the user. When the validator finds a violation of a correctness requirement it provides the user in all cases with a counter–example to his (implicit or explicit) correctness claims, in the form of a textual or graphical back–trace.

Of course, none of *sdlvalid*’s capabilities would mean anything to a designer if the validation tool cannot tackle the complexity of his or her applications. Performance was crucial to the success and the adoption of the validator. A few words are therefore in order on how this performance was realized.

Complexity Management

Almost all validation tools in use today are based on some form of reachability analysis. In a standard reachability analysis of a concurrent system, all system components (processes, message channels, variables) are placed in a user–defined initial state and all system states reachable from that state in one or more steps are systematically generated by an exhaustive symbolic execution of the specification [Holzmann ’90]. For the algorithm to terminate, clearly the number of reachable states must be finite and execution cycles must be recognized. These conditions are readily met in the systems we are studying. Sadly, this algorithm is exponential in the number of executing processes, but no better method is known or even likely to be found [cf., Garey and Johnson ’79].

Reachability analysis then is basically a full shuffle of all possible executions of concurrent processes to derive all reachable composite system states. Reachability analysis sometimes goes under different names. For instance, it is known as parallel expansion in CCS [e.g., Milner ’80] or LOTOS [e.g., Eijk, Vissers and Diaz ’89], or as the shuffle product in early process algebras [e.g.., Holzmann ’82]. It is important to recognize, however, that all these methods are based on the same notion of exhaustive searching. The

computational complexity of all these methods is, inescapably, the same.

The bottleneck in these validations has two components: the CPU−time and the amount of memory that is minimally required to complete a full validation. The performance bottlenecks are well−understood and documented [Holzmann '90]. On large machines a standard reachability analysis algorithm can manipulate state−spaces of, say, 64 Megabytes of memory, analyzing in the order of 1,000 to 10,000 reachable system states per second of CPU time.

The typical size of a single reachable system *state* for the types of problems we are considering here, however, is in the order of 10 Kbytes of data (variables, buffers, processes, etc.). As always in validation work, it is the responsibility of the designer to validate the smallest possible representation of a design problem. But it can require considerable human ingenuity to achieve reductions of the problem specifications in this environment, and achieving it can jeopardize the accuracy of a validation. We are therefore interested in minimizing the demand on the designer to simplify designs just to make them verifiable.

The number of effectively reachable system states in a full design of the size we are considering is typically in the order of 10^9 and up. A traditional validation methods that requires full storage of all 10^9 states of 10^4 bytes each can clearly analyze no more than $64 \cdot 10^6 / 10^4$ or 6400 states out of a state space of 10^9 states, giving an unacceptably low coverage:

$$64 \cdot \frac{10^6}{10^4} \; : \; 10^9 \;\; \rightarrow \;\; 0.001\% \;\; \text{coverage/run} \qquad \text{[standard]}$$

On secondary storage, perhaps up to 1 Gigabyte of storage could be claimed, at the expense of a reduction of the search speed to in the order of 10 to 100 states per second [Holzmann '91a]. In that mode, in theory, up to $10^9 / 10^4$ states could be inspected, or roughly 10^5 out of 10^9 states, but even with the best known methods such a search would take in the order of 10^7 to 10^8 seconds of CPU time and thus combines an unacceptable coverage (0.01 %) with an unacceptable performance (10 to 100 days).

To solve this problem, a different type of validation algorithm was introduced in 1987 [Holzmann '87]. This algorithm, which was named *supertrace*, when run on the same validation problem (same hardware, same memory requirements) can analyze in the order of $8 \cdot 64 \cdot 10^6$ or $5 \cdot 10^8$ reachable system states in a single run, at roughly 10^4 states per second.

$$8 \cdot 64 \cdot 10^6 \; : \; 10^9 \;\; \rightarrow \;\; 50\% \;\; \text{coverage/run} \qquad \text{[supertrace]}$$

With multiple runs using different hash methods the coverage of the validation can be brought arbitrarily close to 100%. With this algorithm incorporated into *sdlvalid* the coverage of even a single validation run provides a coverage well above that of any other state space searching method known to date. The design of the supertrace algorithm is explained in [Holzmann '91a]. A sample implementation of the algorithm, though not *sdlvalid* itself, is freely available for educational use [cf. Holzmann '91b].

3. The Construction of Fire−Walls in Designs

We have now accumulated a few years of experience with the a formal validation tool in an industrial design environment. The problems encountered here have traditionally been large enough that most formal methods break down, for both technical and non−technical reasons.

The technical reasons have to do not only with the size and the complexity of the systems being designed, but also with the way in which new designs must be integrated with the older ones. As one example, an ordinary telephone switch is controlled by several million lines of high level code, which constantly undergoes revision by many hundreds of programmers. Any new code has to be compatible with the code that already exists and is usually based on a direct revision of existing code. Since a thorough redesign of the entire system using formal methods is seldom a viable alternative, any new formal method that is introduced must be able to cope with the existing code−base. Very few of the existing formal methods do.

The approach we have taken here is to require designers to formalize precisely what their assumptions are about the behavior of the existing code. We thus build *fire−walls* between a new design and the existing code base. This fire−wall encapsulates the minimal set of assumption one has to make about the environment in which a new design is placed. It would be impossible to formally validate the accuracy of those assumptions, since the existing code base is not only large, but also produced before formal methods were

introduced. The best one can do here is to build run–time checks into the system that check the validity of all the assumptions that were necessary to construct the fire–wall. All violations of these assumptions are reported at run–time. For a given set of assumptions, however, it can be formally validated that any addition to the existing system either preserves or contributes desirable correctness properties.

The non–technical reasons that formal methods can quickly break down when applied in industrial settings have to do with the training and motivation of the designers who have to use them. Engineers typically work on strict deadlines, and are unavoidably rewarded sooner for promptness than for thoroughness. We expect that this trend can only change over time, when sufficient data can be gathered to support the claim that the designs produced with the benefit of a formal method and formal validation tools are more reliable, easier to maintain, and easier to extend than traditional designs.

The result of the application of *sdlvalid* has been carefully documented. The first application, for instance, was in the implementation of an ISDN protocol in the International Switching Division of AT&T Bell Laboratories. Over a period of one year, 97 different types of incompleteness and 6 different types of deadlock were discovered in the growing design, with the help of the validation tool. The log for individual invocations of *sdlvalid* records 1,636 runs in that period, giving approximately one error discovered for every 16 validation runs.

Based on these first results we have substantially extended our efforts to build new tools to support the introduction of formal methods into the design environment at AT&T. Our current focus is on the development of new tools that can guide the designer through a controlled refinement process that is integrated with formal validation. The two overriding concerns of this new effort are again: user friendliness, and effective complexity management.

4. Conclusion

The reason that formal methods are introduced is sometimes described as follows: *''because they are intellectually exciting to use, and thus give daily delight in the professional pursuit of systems development''* (quoted verbatim from a recent conference announcement). Though this is a worthy feature, it is not the immediate goal of the adoption of formal methods in an industrial environment. The purpose of the adoption of formal methods here is ultimately to increase the feasibility for formal *validations*, and through validation increase the reliability of a design. A formal method therefore requires the integral support of strong validation tools before it can be of value in the design process. There are many non–technical issues that can prevent the large scale adoption of formal methods in industrial environments. Ideally, the method must fit smoothly into the existing design environment, the supporting tools must require minimal training, and they must deal effectively with the unavoidable complexity of industrial size designs.

We have experimented with a formal validation tool in an industrial development environment since the middle of 1987. To date, close to one hundred designers have taken tutorials in the usage of the tool, and two design teams have incorporated some form of formal validation into their projects. (At the time of writing, August 1991, a third team has adopted the validation tool, but not applied it yet.) By AT&T standards, the experiments have been of a modest scale. For a formal method, and for the application of a formal validation tool, the size of the applications is unprecedented. Validations routinely involve state–spaces of up to a billion reachable states, searching for correctness violations that would be impossible to spot manually. Every type of design error found in the validation runs could have been very costly indeed if it had not been caught.

The problems that had to be solved to give our tools the characteristics outlined in this paper have been great, though many of these have been of a non–technical nature, e.g., how to provide the right user–interface that is powerful and yet simple to use. There is much work that remains to be done to fully integrate formal validation with design, but we have good reason to believe that we are on our way towards realizing that goal.

5. Acknowledgements

The experiments with formal validation in an industrial development environment would not be possible without the enthusiastic support of many people in AT&T's SDL tools group and in the 5ESS® International Switching Division. I specifically want to mention the support and the pioneering work of Joanna

Patti, John Chaves, and Joe Lin. It is also a true pleasure to notice that this work is strongly supported, facilitated, and encouraged by both the research and the development organizations within AT&T .

6. References

[Bennett '86], Bennett, R.L., Lindner, J.A., Michelsen, R.W. and Rypka, D.J., "SDL in 5ESS Switching System Development," *Proc. 6th Int. Conf. on Software Eng. for Telecomm. Switching Systems*, Eindhoven, the Netherlands, 14–18 April, 1986.

[Eijk, Vissers and Diaz, '89], Eijk, P. Van, Vissers, C.A., and Diaz, M. *The Formal Description Technique Lotos*, North–Holland Publ., Amsterdam, 1989. 451 pgs.

[Garey and Johnson '79], Garey, M.G., and Johnson, D.S. *Computers and Intractability: a Guide to the Theory of NP−completeness*, Freeman, San Francisco.

[Holzmann '82], Holzmann, G.J., ''A Theory for protocol validation,'' *IEEE Trans. on Computers*, Vol. C−31, No.8, pp. 730–738.

[Holzmann '87], Holzmann, G.J., ''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 '88], Holzmann, G.J., ''An improved protocol reachability analysis technique,'' *Software, Practice and Experience*, Vol 18, No. 2, February 1988, pp. 137−161.

[Holzmann and Patti '89], Holzmann, G.J., and Patti, J., ''Validating SDL specifications: an experiment,'' *Proc. 9th IFIP WG 6.1 Int. Workshop on Protocol Specification*, *Testing, and Verification*, North–Holland Publ., Amsterdam, pp. 317–326.

[Holzmann '90] Holzmann, G.J., ''Algorithms for automated protocol validation,'' *AT&T Technical Journal*, Special issue on Protocol Testing and Verification. Vol 69, No 1, pp. 32–44.

[Holzmann '91a], Holzmann, G.J. *Design and Validation of Computer Protocols*, Prentice Hall, 1991, 512 pgs.

[Holzmann '91b], Holzmann, G.J., ''Tutorial: design and validation of computer protocols,'' *11th IFIP WG 6.1 Int. Workshop on Protocol Specification,*, *Testing, and Verification*, Stockholm, Sweden. To appear in Computer Networks and ISDN Systems 1992.

[Jilek '84], Jilek, E. R., "Implementation of SDL/PR in a Digital Switching System," *Proc. IEEE Global Telecomm. Conf.*, Atlanta, Ga. November 26–29, 1984.

[Lippold '85], Lippold, T. T., "Programming Call Processing Directly in SDL," *Proc. National Communication Forum*, 1985.

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

[Milner '80], R. Milner, ''A calculus for communicating systems,'' *Lecture Notes in Computer Science*, Vol. 92.

[Sarracco, Smith and Reed '89], Robert Saracco, J.R.W. Smith, and Rick Reed, *Telecommunications Systems Engineering using SDL*, Noth−Holland Publ, 1989, 632 pgs.

[SDL '87], "Special issue on the CCITT language SDL," *Computer Networks and ISDN Systems*, Vol. 13, No. 2, pp. 65−134, 1987.