Validating SDL Specifications: an Experiment

Article · May 2003

Source: CiteSeer

CITATIONS
27

READS
148

2 authors, including:

Gerard Holzmann
Nimble Research
238 PUBLICATIONS **18,117** CITATIONS

SEE PROFILE

Validating SDL Specifications: an Experiment

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Joanna Patti

AT&T Bell Laboratories
Naperville, Illinois 60540

ABSTRACT

This paper describes a method for validating specifications written in the CCITT language SDL. The method has been implemented as part of an experimental validation system. With the experimental system we have been able to perform exhaustive analyses of systems with over 250 million reachable composite system states. The practicality of the tool for the analysis of substantial portions of AT&T's 5ESS® Switch code is now being studied.

Proc. 9th Int. Workshop on Protocol Specification, Testing and Verification, Twente University, The Netherlands, June 1989. North−Holland−Publ. Amsterdam.

1. Introduction

A particularly effective and well understood technique for validating systems of extended finite state machines is reachability analysis. With this method a validation program exhaustively forces the system into all states that are reachable by a series of execution steps, starting from a single known initial system state. For each reachable system state the program then verifies the observance of a predefined set of correctness criteria, such as the absence of deadlocks and unspecified receptions. Variants of this technique have been in use for over ten years [e.g. Zaf '77].

Without too much programming effort this technique can be used to analyze systems generating in the order of 100,000 reachable system states. Larger systems are problematic for almost all existing validation methods. The CPU−time and storage requirements quickly become prohibitive [Holz '90]. The basic complexity of the search algorithms, of course, can not be changed. The complexity can be expressed as the number of system states that has to be generated, inspected and stored to perform a completely exhaustive search. The amount of time that must be spent in each basic operation of the algorithm, however, can be reduced substantially with a few, relatively simple, programming tricks.

In a well−tuned system the most expensive operations of the search are all storage related: storing states, comparing two states, computing a hash value for states. With moderate programming effort other operations can be optimized sufficiently to have a smaller impact on the run time. An attempt to replace the traditional storage techniques with less time consuming ones proves to be exceptionally rewarding. The exhaustive search technique can gain one to two orders of magnitude in power, measured as the maximum number states that effectively can be analyzed on a machine with a given size and speed. The method and its rationale are fully described in [Holz '88] and [Holz '90].

We now have almost two years of experience with this new method, which was named *supertrace*. The scope and performance of supertrace have been documented and the software has meanwhile been made available by AT&T, for non−commercial use. A mild challenge to protocol designers, posed by the first author after the development of supertrace, has been to find protocols of true practical significance that,

because of their size, still can not be validated exhaustively. This challenge, of course, did not go unanswered for very long.

In this paper we will focus on the application of supertrace to a large problem of unquestionable practical significance: SDL code that is being developed for a telephone switching system. Where applicable we refer to other documents for more information about supertrace, the validation languages *Argos* and *Promela* used by supertrace, and the specification language SDL.

2. Telephone Switching: the 5ESS Switch

AT&T's 5ESS Switch contains over three million lines of code. The largest part of the software is written in the programming language C. The switch, however, contains not only a substantial amount of code, it also uses a large amount of concurrency. A substantial portion of the switch software (approximately 10%) is therefore written in the CCITT specification language SDL. SDL was specifically designed for the specification of distributed systems, in particular for telephone switching systems [e.g. Jilek '84, Lippold '85, Bennett '86].

SDL is used in the 5ESS Switch to implement the call processing aspects of the business and residential customer features (BRCS), ISDN call processing, and it is used for the implementation of the operator administration features. About 10% of the 5ESS Switch software developers at AT&T write their code exclusively in SDL. Before the SDL code is run in the switch it is translated into C, which is then run under the control of a special purpose real−time executive system called FEX (Feature Execution system) that provides the proper SDL semantics.

2.1. Reasons for Using SDL

SDL is well suited to handle the multiplicity of features that is required to implement BRCS services. The BRCS software is structured into a series of independent SDL processes, with each feature corresponding to one or more processes. The interactions between the different features are simple intraprocess message communications [Bennett '86]. The decision to use SDL has several additional benefits:

- The structure of the language mimics the way people tend to think about how telephone service works. For this application it is more natural to use SDL than a procedural language like C.
- The language can support stepwise refinement: the code passes easily from a system engineer to a code designer, without changes in format. The system engineer, for instance, can concentrate on the message passing level of feature development, and pass the design on to the code developer for refinement.
- The formalism of SDL supports a mechanical translation of specifications into Finite State Machine (FSM) models coded in C. The code generated in this way implicitly enforces a structure that makes the feature logic more evident.
- SDL is an internationally supported language, often used and required by AT&T's customers.
- Since the feature code can be sub−divided into a series of separate FSMs, teams of code designers can conveniently work in parallel.

2.2. Existing Support Tools

Two versions of SDL are supported for the 5ESS Switch development: SDL/GR and SDL/PR. SDL/GR is the familiar graphical form of SDL; SDL/PR is the textual, program form, equivalent. The existing SDL tools support mainly the front−end of the development process. There is an editor for creating SDL/GR graphs, with some syntax checking built in. There are tools for translating specifications from SDL/GR into SDL/PR and back, and there is a tool for translating SDL/PR into C. There is also a report generator and an SDL simulator. So far, however, each tool still accepts a slightly different set of SDL constructs. The ultimate goal is of course to provide a tool set supporting one consistent set of language constructs, and providing a range of capabilities.

The SDL simulator is the most recent addition to the support tools. It has proven invaluable in debugging fresh designs. But simulation, like testing, can never prove the absence of errors with any measure of reliability.

SDL is an extended FSM language, enhanced with general primitives for exchanging data through asynchronous message passing. It can in principle be translated into any other FSM based language. One such language is *Argos*, developed as a standard protocol validation language, which has been used since 1983 to implement a series of protocol analyzers of increasing power and performance [Holz '87a]. 'Supertrace,' is merely the latest entry in this series. The question we are trying to answer is: Can the method employed by supertrace be used for the analysis of SDL code, rather than *Argos*, and in particular can it cope with the complexity of the SDL code that runs in the 5ESS Switch?

3. The Experimental System

3.1. Overview

Figure 1 illustrates the experimental system that is being studied for the validation of SDL code.

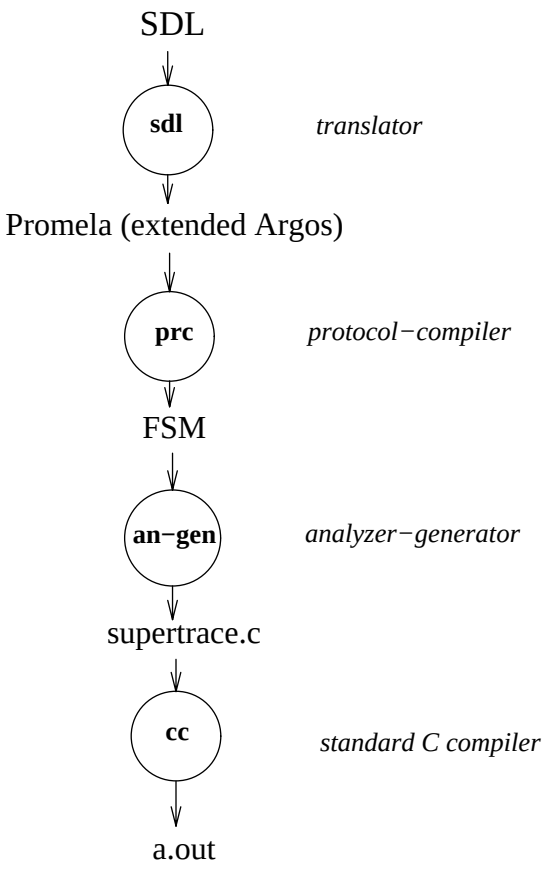

Figure 1 – Experimental Validation System

After a series of translations, this system derives an executable protocol analyzer that can perform an exhaustive validation of the original SDL specification. At present, the validation is based on an intermediate model in a validation language. The construction of the experimental validation system required substantial changes in the validation language *Argos*. Rather than patch the changes into the existing language, it was decided to design a new language that can support SDL features, while maintaining the elegance and simplicity of the initial design. The new language is called *Promela*, short for Protocol Meta Language. Features supported by *Promela*, that were not in *Argos*, include:

- The dynamic creation of processes and message channels.
- An arbitrary number of message parameters.
- Multiple data types (bit, byte, short, int).
- Facilities for using externally defined C functions, linked in by the C compiler.

With the new language, a new supertrace analyzer−generator was written that also supports some features not found in the original system:

- Detailed source level tracebacks.
- A rubber state vector.

The rubber state vector is a logical extension of the original state vector model [Holz ’87b, ’88]. A rubber, rather than a static, state vector allows for dynamically growing or shrinking systems. Most other features of the original supertrace software have been preserved. The support of a rubber state vector, however, did lower the effective performance by a factor of approximately two.

3.2. Supertrace Performance, a brief review

By measurement we can determine that the speed of supertrace software is dominated the calculation of hash values for state vectors: the only fundamental storage operation that cannot easily be $roman replaced. sup 1$ The hash function used in the current implementation can process approximately 1 million bytes (Mbytes) per second on a DEC−VAX−8550 (running the Unix® Operating System), or about 5 Mbytes per second on a large IBM 3090 computer (running Amdahl’s UTS® operating system).

The state vector in supertrace contains compacted runtime values of all state information relevant to the system being validated: contents of queues, values of all local and global variables, control flow states for all running processes etc. The above performance figures mean that a system with a state vector of 100 bytes can be analyzed at a constant, and predictable, speed of 60,000 reachable states per second of CPU time on the larger machine, independent of the total size of the state space being constructed. Since the runtime requirements are with good approximation a linear function of the length of the state vector, the performance can also be expressed as: 6 million state−bytes per second.

The algorithm that achieves this performance was published [Holz ’88]. Source code for the complete validation system shown in Figure 1 is available for non−commercial use, i.e. to universities. This revised implementation of the supertrace software, incurs a time penalty (a factor of roughly two) for the usage of the rubber state vector. Nevertheless, we know of no other algorithm with a better measured performance and coverage of the target state space for large systems. Supertrace, then, seems an appropriate tool for tackling the job of formally validating large protocol systems.

4. First Results

The system illustrated in Figure 1 has been in use since the middle of 1988. At the time of writing the system is used on a daily basis by a team of developers, and it works as advertised: it quickly finds the bugs in the interactions of concurrent modules. The errors found correspond to unexpected interleavings of events, unspecified receptions, unexecutable code, or even complete system deadlock. The largest modules analyzed so far contain over 10,000 lines of SDL. It translates into a validator of over 100,000 lines of C, which produces a state space of over 250 billion reachable system states when run. The initial choice for a pre−processor and a redefined validation language does, however, have some drawbacks that will have to be remedied. The first drawback is that error tracebacks produced by the analyzer generated are source level tracebacks through the *Promela* source, not the original SDL sources. The second drawback is that a translation from SDL source to analyzer in three separate steps can introduce clumsy code that may unnecessarily complicate and slow down the final validations.

The new system we are developing, to remedy these drawbacks, is depicted in Figure 2 below.

---

1. The calculation is necessarily part of all validators based on reachability analysis, but in traditional systems there are also other, more time−consuming, operations to perform.

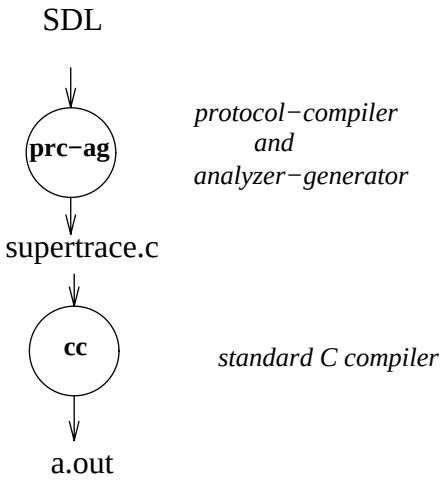

Figure 2 – Target Validation System

In the target system, the first three programs of the experimental system will be replaced by a single program that reads SDL specifications and produces an optimized analyzer generator, with source level traceback capability directly into the original SDL files.

5. Problem Areas

The problems we have encountered in the derivation of an analyzable model from SDL specifications in most cases have to do with the possibilities that SDL offers to circumvent the language proper. We will discuss those cases in the next few sections. One other problem is temporary. It is caused by an incompleteness in the version of the SDL language currently supported by our tools. The destination of a message in SDL can be any process. If the process is unambiguously named, there can be no confusion. There is, however, also a facility for defining a hierarchy of SDL processes in SDL blocks. Within that context it is possible to define message 'paths,' a sequence of processes that can act as a processing pipeline, transferring messages along the path towards their destination. A message, for instance, can be sent to a generic destination 'NEXT,' where it depends on the hierarchy, and the current process's place within that hierarchy, what the destination of the message is. Hierarchy specifications and message destinations such as 'NEXT' are not yet within the scope of the experimental system.

5.1. Open Coded Tasks

The contents of some SDL statements, such as tasks and decisions, is not subject to any language rule. A task may contain C, Pascal, Fortran, or a James Joyce novel. Clearly, there cannot be any method for validating the contents of SDL tasks unless some further rules are imposed. The simplest rule is that the code in declarations, tasks, and decisions is compatible with the target language of the SDL translator, which in our case is *Promela*. In the experimental system the SDL translator can also optionally hide code that is not compatible with the target language, and produce a non−deterministic model that will be validated exhaustively (i.e. for all possible effects of the hidden code on the visible code).

An example can illustrate this process. Consider the following specification in SDL/PR format.

```
BLOCK B2;
    PROCESS P1;
        STATE ST1;
            INPUT M2;
                TASK 'x = y+1';
                NEXTSTATE ST2;
            INPUT M5;
                DECISION 'x<y';
                (TRUE): NEXTSTATE ST2;
                (FALSE): NEXTSTATE ST1;
                ENDDECISION;
        STATE ST2;
            INPUT M5;
                OUTPUT M4;
                NEXTSTATE ST1;
    ENDPROCESS;
ENDBLOCK;
```

Example 1 – SDL/PR fragment.

The task and decisions in this example conform to both C and *Promela* syntax, so it can be translated into a literal *Promela* model, which may look like as follows.

```
proctype P1() of { byte[2] }
{   byte _msg;
ST1:
    if
    :: SELF?M2 ->
        x=y+1;
        goto ST2
    :: SELF?M5 ->
        if
        :: (x<y) -> goto ST2
        :: (!(x<y)) -> goto ST1
        fi
    fi;
ST2:
    SELF?M5 ->
        ENVIRON!M4(SELF);
        goto ST1
}
```

Example 2 – Promela version.

The message buffer through which processes of type P1 receive their input is accessible through the (local) name SELF. The destination of the output generated is by default ENVIRON. Note that within SDL the identity of a sender (i.e. SELF) is tagged onto each message transmitted and is available to the receiver, in a local variable named SENDER.

If the user of the validator chooses to hide the contents of the tasks and decisions, the following model would be produced.

```
proctype P1() of { byte[2] }
{   byte _msg;
ST1:
    if
    :: SELF?M2 −>
        goto ST2
    :: SELF?M5 −>
        if
        :: (1) −> goto ST2
        :: (1) −> goto ST1
        fi
    fi;
ST2:
    SELF?M5 −>
        ENVIRON!M4(SELF);
        goto ST1
}
```

Example 3 – Promela version – hidden tasks and decisions.

The decision after the reception of a message M5 in state ST1 is now a nondeterministic one: both branches are considered executable and the execution paths that can follow either choice will be traced. Note that in Examples 1 and 2 the decision is forced by the possible values of variables x and y. Even in those cases though, it is still possible for x and y to have different relative values, for different execution paths that lead to the decision point. In that case the two possible decision clauses may still be validated, depending on the execution histories. With the model from Example 3, however, coverage of both clauses is guaranteed (though maybe not always desired). Hiding tasks therefore increases the complexity of the systems to be validated: though the number of statements per process is reduced, the number of permissible paths increases substantially. It does however allow us to perform validations even for incompletely specified systems, or for systems that rely heavily on portions of code that are written in a target language other than SDL (or *Promela*).

It is still possible to write SDL code that cannot be hidden or translated. It is, for instance, valid in SDL to use non−SDL code to dynamically evaluate the destination of a message transmission:

```
OUTPUT message TO 12**3 + Fortran_routine()*6;
```

The expression that produces the destination is not compatible with C or *Promela*. In cases like these, therefore, the translator will modestly have to admit defeat and flag an error.

5.2. Uninterpreted Decisions

There is another problem associated with the conversion of explicit SDL into non−deterministic *Promela* decisions in the validation models. Consider the following SDL code fragment.

```
DECISION (x);
(3):NEXTSTATE ST2;
(y):NEXTSTATE ST1;
ENDDECISION;
```

Example 4 – SDL Decision, implicit 'ELSE'

If 'x==3' the first clause is to be executed, if 'x==y' the second. If, however, 'x!=3 && x!=y' the decision clause is to be skipped. The general translation of this code into *Promela* therefore looks as follows.

```
if
:: (x == 3) −> goto ST2
:: (x == y) −> goto ST1
:: (x != 3 && x != y) −> skip
fi
```

Example 5 – Promela Decision, explicit 'ELSE'

The 'if' statement in *Promela* blocks if none of its clauses is $roman executable. sup 2$ If the decision is hidden from view, this becomes:

```
if
:: (1) −> goto ST2
:: (1) −> goto ST1
:: (1) −> skip
fi
```

Example 6 – Promela Decision, explicit 'ELSE', uninterpreted decision

From the SDL specification it can not be determined if the implied 'else' clause can ever be executed. In Examples 1 and 2 the two clauses in the decision form a complete set: '(x<y && !(x<y))' is always unexecutable. However, Examples 4 and 5 show that it is not always that clear. If tasks are hidden from view the analyzer may report errors caused by choices that would be unexecutable in reality. In the last example it may even cause a violation of implied SDL syntax: there is a path through the transition that does not specify a next state. To see how the problem can be avoided, consider the following example.

```
DECISION (a);
(0):NEXTSTATE ST2;
(1):NEXTSTATE ST1;
ENDDECISION;
```

Example 7 – SDL Decision, implicit but unintended 'ELSE'

Only the programmer knows in this case if variable 'a' can ever have any other value than 0 or 1. The only prudent choice for the analyzer generator is to consider also the third possibility where '(a<0||a>1)', which is in accordance with SDL semantics. The designer can avoid the confusion by using the SDL keyword 'ELSE' in all cases where it is known that the 'implicit else' should be ignored:

```
DECISION (a);
(0):  NEXTSTATE ST2;
ELSE: NEXTSTATE ST1;
ENDDECISION;
```

Example 8 – SDL Decision, explicit 'ELSE'

5.3. Open Systems

Another problem that had to be dealt with in the experimental system was that many SDL applications are not self−contained. They are meant to run in an environment of other processes that is not specified. Supertrace, however, was designed to perform an exhaustive validation for all possible behaviors of a 'closed' system, i.e. a system in which all processes are specified. Performing a validation of an open system for all possible behaviors of the environment is almost always undesirable: a malicious environment can disrupt the best designed system. Note for instance that a phone system need not be designed for customers who can generate two off−hook signals in sequence, without an intermediate on−hook. Not only does it increase the complexity of the analyses to generate and validate all possible nonsensical behavior of the environment, the results of such an analysis can not be of much interest to the designer.

The best solution to this problem, from the validator's point of view, is to ask the designer of the partial system to supply a dummy SDL specification of the (expected) behavior of the environment, for instance as

2. The *Promela* 'if' statement is mainly used as a synchronization mechanism.

a special SDL process definition. The validator can than exhaustively analyze the system under the given constraints to the environment. The validation in these cases, however, will be no more reliable than the accuracy of the environment specification, which is usually no more than a quick guess.

Another solution is to extract the expected behavior of the environment directly from the model that the designer provides. This is risky, but at least self−contained: it assumes that the assumptions inherent to the original SDL design are correct, and validates for any logical inconsistencies that remain. Precisely those errors are of primary interest to the designer. In a validation of this type, all message interactions with the environment are labeled 'free.' The label means that those interactions (i.e. the reception or the transmission of a message) are always considered to be executable and have no effect. In other words: wherever the original design claims that a message from the environment may arrive, supertrace will validate for the possibility that such a message will arrive, and it will check the consequences of that event. If in Example 2 message M5 would be labeled 'free,' the model generated would change as follows.

```
proctype P1() of { byte[2] }
{   byte _msg;
ST1:
    if
    :: SELF?M2 ->
        x=y+1;
        goto ST2
    :: (1) ->   /* input of free message M5 */
        if
        :: (x<y) -> goto ST2
        :: (!(x<y)) -> goto ST1
        fi
    fi;
ST2:
    (1) ->
        ENVIRON!M4(SELF);
        goto ST1
}
```

Example 9 – Free messages

Note that in the above example (following the simulated reception of free message 'M5') both paths through the decision in state 'ST1' are always validated, while the path following the reception of message 'M2' is only validated if there is at least one scenario in which the message 'M2' can arrive in state 'ST1.' The translator can determine, without user assistance, for some messages that they should be set 'free.' This is the case for messages that are sent but not received and for messages received but not sent. Other messages will have to be provided by the user in a special file that contains a list of message names. This so−called 'freelist' is then used by the analyzer generator to label the additional messages.

5.4. Modeling Timeouts

SDL has a facility for specifying any number of timers. A timer can be set or reset, thus enabling or disabling a timeout. Timers are traditionally a difficult concept to handle in a validation system. The validation, after all is meant to establish the self−consistency of a specification *irrespective* of timing considerations, that is independent of the relative speeds of processes and independent of possible transmission delays of messages. That means that in its full generality a validation of this type must consider the timeout as a possibility that is either present or absent, never as a true probability or even as a conditional probability that depends on the relative timings in the system.

It is quite feasible, though also quite pointless, to consider all possible consequences of an ill−set timer, for instance a timer that fires immediately when set. The purpose of the timer, however, is to recover from potential deadlock, for instance a deadlock caused by message loss. In the experimental system, therefore, a 'timeout' is an action that only becomes executable in, what otherwise would be, a deadlock state. This heuristic has proven to be very successful in avoiding the tracing of non−errors caused by unlikely timer behavior, and finding the remaining logical errors. A timer 't0' in an SDL process, for instance, is modeled

in the experimental system by a *Promela* process as follows:

```
proctype t0() of { byte[2] }
{   byte SENDER;

    do
    :: SELF?SET,SENDER
    :: SELF?RESET -> SENDER = 0
    :: timeout ->
        if
        :: ( SENDER) -> SENDER!l1,SELF; SENDER = 0
        :: (!SENDER) -> skip
        fi
    od
}
```

Example 10 – Model of an SDL Timer in Promela

The timer can be set or reset, in accordance with the SDL specification. When a timeout becomes executable (using the heuristic rule above) the appropriate timeout message is sent only if the timer is currently set. The analysis is still independent of timing considerations, but does take a reasonable subclass of timer behaviors into account.

5.5. Non−Standard Correctness Criteria

By default, the validation system will check for the completeness of the specification submitted to it: absence of unspecified receptions, absence of deadlocks, absence of improper terminations (i.e. reaching an endstate with not all processes in their final state or in the final state with one or more residual messages in their input buffers). On request (a command line option) the system can also check for the possibility of buffer overrun, or for the presence of non−progress loops, or unexecutable code. With a little more work the system can also verify the observance of any user−specified correctness criterion, phrased as the reachability or unreachability of system states, or the observance of specific system properties. At present we have no formalism for describing such correctness criteria. Though this is potentially the most valuable type of validation that the system can perform, it is also, at least in the experimental system, the least utilized. In practice, in most cases we have seen so far, the basic test for the absence of deadlocks suffices.

6. Non−Technical Considerations

Until now, validation by any other means than simulation or code−walkthroughs by committees of experts was infeasible in 5ESS Switch development. The supertrace software, used in tandem with the existing SDL Simulator, can improve the quality of the code, by finding also the errors that can not be found by manual inspection.

Still, it is not a foregone conclusion that a seasoned SDL designer will accept the new tool, however promising it might appear. Programmers are accustomed to the traditional coding cycle, (edit, compile, run, debug). However, not all SDL designers are necessarily programmers. It is perhaps a strength of SDL that designs can be comfortably created using just pen and paper. The use of sophisticated support tools for the design and analysis of SDL designs needs to be well publicized, formally taught, and promoted, before it can be truly successful. Our aim is to change the coding cycle into: editing, formal analysis, compilation, and conformance testing. At first this may seem needlessly clumsy and involved to the user. It requires, for instance, a rigorous use of SDL and a well−developed abstraction of the environment in which a new application, or feature, is to be used. The body of code that makes up the environment in a telephone switch is very large and difficult to comprehend in any detail. The effort involved may therefore be substantial at first. Part of our job is therefore to show potential users that the extra overhead to the design process can pay off.

7. Summary

The experimental system for validating SDL specifications has been in operation since the middle of 1988. It supports a large fraction of the language, e.g. including SAVE, and CREATE, but not process hierarchies and procedures. The experience we have gained with this system will guide us in the development of the target system: a one step analyzer generator for SDL specifications. Experience so far shows that we cannot expect to be able to validate any arbitrary SDL specification. With some reasonable precautions, however, validation will be feasible. The precautions should guarantee that the non−SDL portions of a design do not redefine or affect the interactions that are written in SDL proper.

With the experimental system we have been able to complete analyses of over 250 million reachable composite system states. This gives reason to be moderately optimistic about the capability of the system to tackle the tasks for which it is being designed, again if some reasonable constraints on the size of the models to be validated are placed.

The target SDL validation system is scheduled to be operational by the end of 1989.

8. References

[Bennett ’86], Bennett, R.L., Lindner, J.A., Michelsen, R.W. and Rypka, D.J., "SDL in 5ESS Switching System Development," *Proc.*, Sixth International Conference on Software Engineering for Telecommunication Switching Systems, Eindhoven, the Netherlands, 14−18 April, 1986.

[Jilek ’84], Jilek, E. R., "Implementation of SDL/PR in a Digital Switching System," *Proc.*, IEEE Global Telecommunications Conference, Atlanta, Ga. November 26−29, 1984.

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

[Holzmann ’87a], "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 ’87b] Holzmann, Gerard J. "On Limits and Possibilities of Automated Protocol Analysis," *Proc.* VII−th Workshop on Protocol Specification, Testing, and Verification, Zurich, 1987, North−Holland Publ. Co., Amsterdam, pp. 339−346.

[Holzmann ’88] Holzmann, Gerard, J., "An Improved Protocol Reachability Analysis Technique," *Software, Practice & Experience*, pp. 137−161, Feb. 1988.

[Holzmann ’90] Holzmann, Gerard, J., "Algorithms for Automated Protocol Validation" , to appear, *AT&T Technical Journal*, Jan/Feb. 1990.

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

[Zafiropulo ’77] Zafiropulo, P., *A new approach to protocol validation*, Int. Conf. on Communications, June 1977, Chicago.