The Theory and Practice of A Formal Method: NewCoRe

Gerard J. Holzmann
AT&T Bell Laboratories, Murray Hill, New Jersey 07974, USA

We discuss what the ideal characteristics of a formal design method should be, and evaluate how the existing methods measure up. We then look at a recent attempt within AT&T to apply formal methods based on design verification techniques, and evaluate it in the same context.

Keyword Codes: D.2.0, D.2.10, F.3.1
Keywords: Software Engineering, Design, Program Specification and Verification

1. INTRODUCTION

A new discipline of *Formal Methods* promises to provide a solid theoretical foundation for software design. Most professionals involved in software design will agree that there is a need for such a foundation. The complexity of the software that controls even the most mundane aspects of our world can be overwhelming. A modern telephone switch, for instance, is controlled by several million lines of control software, that are developed and maintained by hundreds of skilled programmers. Any single change to the code could potentially cause havoc in large fragments of a nationwide telephone network. To prevent this, extremely high requirements must be placed on the quality of the software that goes into a system like this.

Ultimately, the quality of a design is determined by the number of defects that remain after the standard design cycle has been completed. Quality can be improved by a positive or negative feedback principle, as it was done among typesetters in 15th Century Korea:*

> The supervisor and compositor shall be flogged thirty times for an error per chapter; the printer shall be flogged thirty times for bad impression, either too dark or too light, of one character per chapter.

A perhaps more constructive approach is to provide the designers with

- A *design discipline* that minimizes the occurrence of errors, and
- A *design tool* that helps them to detect the errors that remain.

Referring to the analogy of typesetting, a logical first step would be to make sure the compositor can read and write the language. The second step would be to provide him or her with a dictionary, or, better still, with an automatic spelling checker.

Clearly, a good methodology can reduce the reliance on tools. Similarly, an effective tool−set can make up for the flaws of a design methodology, or for the way in which it is applied. In a mature engineering discipline, methodology and tools are merged into a single skill, where the first provides the guidelines and intuition that lead to a good design, and the second provides the means to verify that the required quality standards are being met.

So far, work on formal methods has focused primarily on methodology, while neglecting the tools that would be required to enforce it. The most popular methods offer an unambiguous and formal notation for specifying design decisions, but they lack the tools that would be required to verify objectively if those design decisions are themselves logically complete and consistent, and if, when taken together, they actually solve the original design problem.

The goal of this paper is to demonstrate that it is possible to develop a formal method that offers both an unambiguous notation for expressing design decisions, and automated tools to check the soundness of those decisions. The need for verification, however, comes first, and should guide the development of the methodology. A formal method that does not support

---

* *Early Movable Type in Korea*, Kim Won−Yong, 1954. The quote appears in Daniel Boorstin's book *The Discoverers*, Vintage Books, 1983, Ch. 62.

verification, in our view, fails to meet one of its most important objectives.

2. THE CRITERIA TO BE USED IN EVALUATING FORMAL METHODS

A mature engineering discipline minimally has the following three characteristics:

- It discriminates between requirements and implementations.
- It uses engineering models (prototypes) to verify design decisions.
- It can predict the essential characteristics of a product before it is build.

The first bullet item says that we should be able to make explicit formal statements about the correctness requirements for a new design that are independent of the design itself.

The next item says that we should be able to build an abstraction of the behavior of a large design in a smaller model, which is susceptible to formal analysis.

The last item says that we should be able to derive the essential characteristics of a design from its model. Once the model has been verified, it acts as a blueprint for the final construction.

How well do the existing formal methods measure up to these three requirements? Formal Description Techniques (FDT's) such as Z, VDM, and LOTOS, allow us to formalize an abstraction of the design, and to refine such an abstraction in the direction of an implementation. They provide no separate notation for expressing correctness requirements (first bullet item), and no tools for the formal analysis of the logical soundness of an abstraction for a given set of requirements (second and third bullet item).

Missing from most of the formal methods developed to date is the capability to perform verification: a notation for expressing correctness requirements and a tool for verifying that those requirements inescapably hold for a given prototype design. It can well be argued that a verification methodology should provide the core of the formal method. Without it, the formal method reduces to mere notation.

There are many quality standards that must be applied to the verification methodology itself, if it is to fulfill the critical role in the design cycle that is suggested here. A good verification methodology should provide the designer with a solid scientific method for the analysis of new designs. The method captures designs in a formal framework that allows the user to make formal statements about the design that can be proven or disproven through controlled and reproducible experiments. Any capable user should be able to perform, or to repeat the experiments. Furthermore, it should be possible to state explicitly which physical or theoretical limitations exist to the applicability of the tool or the method. That is, the verification methodology provides:

- A sound formal framework (or theory) for capturing design prototypes.
- The capability to make statements about the design that are either verifiable or falsifiable.
- The capability to perform controlled experiments with reproducible results.
- A clear identification of its constraints.

We are far from realizing these ideals. It appears that the first generation of formal methods that we have developed does not live up to the promise of either an engineering discipline or a scientific method. The existing formal description techniques provide an unambiguous formal notation, but no guidelines for its usage, no means for expressing formal correctness requirements, no means to build prototypes, and no means for verification. The existing formal verification tools, on the other hand, rarely provide a coherent notation for expressing correctness requirements on design prototypes, and (for understandable reasons) they rarely identify their own limitations.

Considering the above, it is probably fair to say that we have not yet succeeded in developing a mature new discipline of design. That is the bad news. The good news is that many people are working hard to change that. In the following sections I will consider one such attempt, and consider how we may approximate the ideals sketched above.

3. THE NEWCORE PROJECT

The NewCoRe* project was started in early 1990 as a collaboration of a small number of people in the research and development organizations within AT&T to trial the viability of formal verification techniques on a regular industrial−size design project. [2] To make the trial meaningful, it was decided that the NewCoRe team would not be allowed to hand−pick a suitable project, but would simply work on whichever project happened to be scheduled next for the target organization. This application turned out to be the design and implementation of a portion of Signaling System 7 in the 5ESS® switching system: the ISDN User−Part protocol defined by the CCITT.

The application was estimated to require a few thousand lines of source code, to be developed by a team of between 40 and 50 designers over a two year period. From the developers point of view, this was only a regular−size project. For formal verification, however, it provided a unique challenge. To the best of our knowledge, a design of this scale, integrally based on automated formal verification techniques, had not been attempted before.

3.1 Goals

To avoid bias in the interpretation of the result of the project, we decided that the NewCoRe team would work in parallel with a conventional design team (the control group), adopting precisely the same requirements, constraints, and deadlines. The aim for both teams was the full development of a completely deliverable product so that, at the end of the design cycle, the two products could be compared on a range of quality measures. The NewCoRe team voluntarily took on a few extra handicaps. First, it decided early on that no more than four to five dedicated designers would be needed to complete the project (a ratio of one designer in the NewCoRe team for every ten in the conventional team). The NewCoRe team also undertook to develop or adapt the tools to support the formal design process in addition to the development of the product. The team adopted three strict requirements for the design tools:

- The tools fit smoothly into the existing design environment
- The usage of the tools requires minimal training, and
- The tools deal effectively with the complexity of the application.

Each of these requirements had a significant impact on the approach that was taken by the NewCoRe team. The first item was probably the hardest to adhere to. The existing design environment is based on the CCITT Specification and Description Language (SDL). [1,7] To be able to perform verifications effectively, however, we needed a notation that can be used to express both design decisions and correctness requirements in a formal framework that can be subjected to rigorous analysis. SDL does not fully meet those requirements. For instance, SDL implicitly defines the addition of an unbounded message buffer to each running process. The unboundedness of the buffer immediately makes all correctness properties of an SDL system formally undecidable. Another SDL language feature allows task statements to contain arbitrary informal text, anything from Fortran to poetry. The feature, alas, is heavily relied upon in our development area. Clearly there cannot be a method for validating the contents of the informal code fragments.

* The name NewCoRe summarizes one of the goals of the project: to explore new techniques for controlled refinement. It is also a play on the internal name of design specification documents: Core documents.

3.2 Methods

(a) SDL Restrictions

To make statements about SDL specifications formally decidable, we had to impose three restrictions on its use. The first is that the designer is required to specify a bound on the size of message channels, the maximum number of running processes, the maximum number of fields per message, etc. The second is that all the code in declarations, tasks, and decisions is expressed in a well–defined (small) subset of the programming language C. The third, and last, restriction is that the SDL system is always completely specified before it is subjected to verification (i.e., there are no hidden traffic sources or sinks).

(b) SDL Extensions

We also had to extend the language with the minimally required formal notation for expressing correctness requirements about an SDL system. Following Zohar Manna and Amir Pnueli's work [6], we defined three classes of properties based on temporal logic: *Invariance*, *Response*, and *Precedence*.

The properties are expressed in a simple SDL–like syntax, as shown in Table 1, where `p`, `q`, and `r` are arbitrary boolean propositions on the system state.

Table 1

LTL Formula	Is Written As
$\square$ p	`ALWAYS p`
$\square$(p $\rightarrow$ $\Diamond$q)	`p IMPLIES EVENTUALLY q`
$\square$(p $\rightarrow$ q U r)	`p IMPLIES q UNTIL r`

In addition to the above format for expressing three classes of temporal logic claims, we introduced extra semantics for existing SDL syntax. For instance, all state–names beginning with the three–character prefix "End" were taken to define SDL states where the system could legally terminate execution. A termination in any other state was defined to be a design error. Similarly, state–names beginning with the prefix "Progress" were defined to identify the effective progress states of the SDL system. Any infinite execution (i.e., cycle) of the system that did not traverse at least one of the progress states thus defined, was again taken to be a violation of the correctness requirements.

Another extension to the SDL language was made to enable the designer to make design abstractions more easily. It supports the specification of non–deterministic decisions, with the help of a new keyword `NONDET`, which can be used as a prefix to SDL decision statements. For instance, in

```
NONDET DECISION 'anything';
(1):... code for first choice...
(2):... code for second choice ...
    etc.
ENDDECISION;
```

the names of condition and decision labels are arbitrary, and each choice is claimed to be equally likely for (or irrelevant to) verification purposes.

A final extension of the SDL syntax served to simplify adherence to the requirement of complete specification of all traffic sources. An extra keyword, `FREE`, was introduced to be used to as a placeholder for messages that are assumed to be always receivable. The sender of the `FREE` messages need not be specified. The optional parameters to a `FREE` message are used to constrain more precisely the circumstances under which the `FREE` message is receivable. For instance, the SDL statement

```
INPUT FREE( QLEN (switch_proc) <= QSZ);
```

specifies an input that is receivable if and only if the length of the message queue for the process named `switch_proc` contains fewer than `QSZ` messages. Everything in upper–case is either an SDL keyword, or a predefined constant. The user–defined bound on the length of a message buffers (i.e., input queues), for instance, is stored in the predefined constant `QSZ`.

Most existing specifications written in SDL can be modified with relatively minor effort into this newly defined, decidable, dialect of SDL. In a first approximation of an analyzable model, the messages sent by all environment processes, for instance, can be mapped to `FREE` messages to simulate completely random behavior. Guided by the requirements of the formal validation (i.e., which specific properties need to be provable), the model can then be refined by adding more realistic assumptions about the environment.

(c) Assessment

It is worthwhile to pause here for a while to consider what purpose the methods summarized above have in view of the ideal characteristics of formal method discussed earlier.

First, the addition to SDL of specific syntax and semantics for expressing correctness requirements provides one of the essential characteristics of an engineering discipline: the ability to distinguish requirements from implementations. Behavior, and requirements on behavior, are specified in distinct portions of the SDL specification, in the dialect of SDL that was adopted by the NewCoRe team.

Second, the extensions and restrictions taken together provide us with a modeling language that allows us to built formal prototypes of a new design. These prototypes allow us to make statements of which the truth value is formally decidable via controlled experiments.

Third, the formal system thus defined fits smoothly into the existing design environment, and requires minimal training. The basic language requirements can be taught within a few hours.

Before we look at the methodology of design that was developed by the NewCoRe team, two issues remain to be discussed to justify our use of the term ‘formal method’ for the work of the NewCoRe team. First, we need to explain how a verification tool for performing the required *controlled experiments* can be constructed. Second, we need to explain explicitly what the *constraints* of such tools are. The next sub–section is devoted to these two issues.

3.3 Controlled Experiments

We have already noted that for finite state systems all correctness properties that can be expressed as properties of states, or sequences of states, are formally decidable. The reachability graph of the system, which is itself necessarily finite, always contains sufficient information to prove or disprove any such property. The traditional way of establishing the truth or untruth of a correctness property is therefore to construct the smallest possible portion of the reachability graph that is sufficient to verify a given set of properties. [4]

For a well–defined formal system, such as our modified version of SDL, the reachability graph can be constructed mechanically from a given specification. This means that there are algorithms that can in principle construct the graph for any given system completely automatically. Our ability to perform controlled experiments rests on this capability.

Constraints

Does this mean that there are no constraints to this procedure? Unfortunately, this is far from true. The reachability graph consists of nodes and edges, where each node represents a unique system state, and each edge represents a single atomic execution step of the system. From any given system state, typically many different execution steps can be made, such as the independent actions of concurrent processes that may be interleaved arbitrarily in time. The graph will be a directed graph, possibly cyclic. In general, the complete graph will be one large strongly connected component, that is: every node in the graph is reachable from every other node.

If there are R reachable system states (R is always finite), and the node that encodes each such state requires S bytes of memory, the complete graph will consume $R \times S$ bytes. But, not only the graph is finite, most computer memories are too. If we have M bytes of main memory available for the construction of the reachability graph, the constraint to this type of search is $R \times S \leq M$ *or* $R \leq M/S$.

For any given design problem the parameters S and M are usually fixed. As an example, if S is $512 = 2^9$ bytes and M is 64 Mbyte = 2^{26} bytes, the maximum value for R is 2^{17}, or about 131,000, reachable system states. Unfortunately, secondary storage (e.g., disk) is much too slow to be used for constructions such as the above, so the constraint is hard to relax.

Two questions must now be asked: (1) will we ever hit this constraint in practice, and (2) what happens if we do? Fortunately, small designs will comfortably fit within the constraints of this verification method. They typically consume fewer than a few hundred bytes of memory to store one node in the graph, and also generate less than a few hundred thousand reachable system states. For a full industrial–size design, as we are studying in the NewCoRe project, however, a single node may take up to thousands of bytes of memory to store, and the number of reachable system states can be in the billions. That is, with S ˜ $O(10^3)$ and R ˜ $O(10^9)$ we would require a machine with M ˜ $O(10^{12})$ to still be able to apply the traditional verification methods. So far, however, machines with more than $O(10^8)$ bytes of main memory (e.g., 128 Mbyte) are hard to find.

This strict limitation to the standard verification techniques is rarely spoken about, other than as an obscure and unavoidable theoretical limitation. But the effect can be quite plainly calculated. If we run the verification algorithm for a problem with 10^9 states on a machine with 10^8 bytes of memory, the algorithm *stops* after generating $10^8/10^{12}$ or 0.01% of the reachability graph: it runs out of memory. Not only do we hit the constraint head–on, hitting it can make the verification attempt itself rather futile. Can't we do better?

The ideal way of avoiding the constraint is to reduce the complexity of the basic model, so that it can be described by a reachability graph that fits the limits of the available hardware. One of NewCoRe design strategies is aimed at this (the controlled refinement strategy to be discussed). Another approach is, however, to organize the verification tool itself in such a way that with the same problem size, and with the same memory bounds, a substantially larger fraction of the reachability graph can be handled, boosting, for instance, the expected coverage from 0.01% to close to 100%. Using the, so–called, 'supertrace' (or bitstate hashing) technique [3,4], nodes can be encoded in just 1 bit of memory, while preserving all the capabilities of a conventional verification algorithm. The 10^9 nodes from the last example can now be encoded in 10^9 bits of memory, or 10^6 bytes = 1 Mbyte, which is less than 1% of the assumed available storage of 128 Mbyte. The expected coverage of the verification run can now be increased to arbitrarily close to 100%.

Of course, the supertrace technique has constraints as well. In return for the increase in coverage, it now becomes unknown and unknowable what the coverage of each run precisely is. We only know that in cases where $R >> M/S$ the *expected* coverage of a supertrace run will be several orders of magnitude larger than that of a traditional algorithm. This means that the technique should not be used when the expected coverage of a traditional run would already be 100%, that is when $R \leq M/S$. In that case we only lose information by using supertrace (i.e., the certainty that 100% coverage was realized). The problem domain of the NewCoRe project, however, makes traditional verification rarely a viable option.

The Tool: Sdlvalid

The SDL verification tool that implements the supertrace algorithm, and supports the extensions to SDL that we discussed earlier, is called *sdlvalid*. The requirement on the tool itself is that it requires virtually no training, and works much like a spelling checker. No knowledge of formal methods, temporal logic, or proof theories is required to perform verifications. If *sdlvalid* detects a violation of a correctness requirement, either pre–defined or or user–defined, it produces a complete counterexample that demonstrates along which execution sequence the property is violated. Examples of the pre–defined types of error are:

- Logical 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
- System Deadlock (also called deadly embrace, or circular waiting)
- Buffer Overrun (attempts 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)

Any violation of these implicit properties, as well as any violation of explicit temporal logic requirements, is reported as a counterexample to the claim in the form of a complete trace-back, leading from the initial system state to the first state where the violation becomes evident. Figure 1 gives an example of an execution scenario produced by *sdlvalid*, for instance as part of a counterexample demonstrating the violation of a correctness claim. Counterexamples can also be produced in textual form if, for instance, no graphical output device is available. The tool *sdlvalid* automatically delivers the highest possible coverage of the reachability graph, regardless of problem size.

We have now outlined the design of a tool that, within specific constraints, can be used to perform controlled experiments, aimed at proving or disproving the validity of correctness claims for new designs. In the next section we discuss the design methodology that was adopted in the NewCoRe project based on this tool.

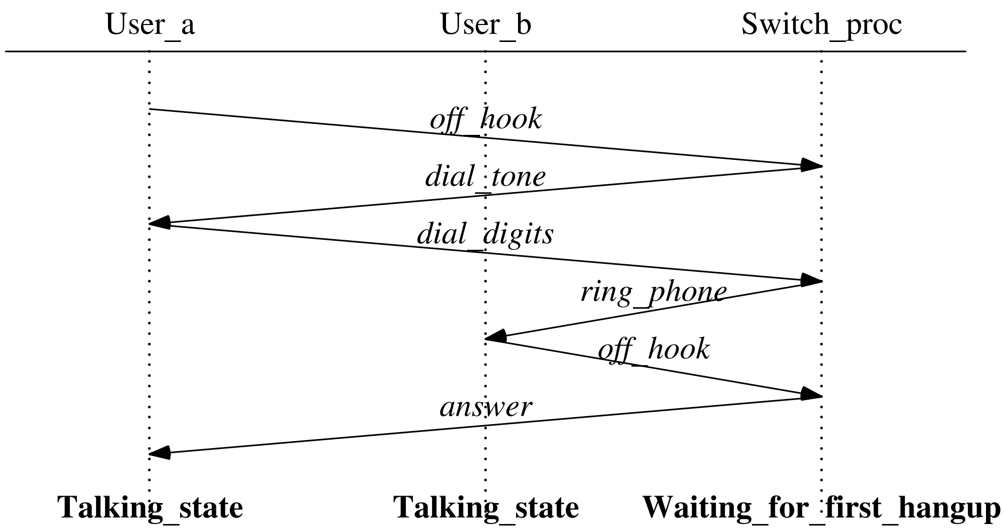

Figure 1 − Sample Execution Scenario produced by Sdlvalid (Graphical Form)

4. THE NEWCORE DESIGN DISCIPLINE

4.1 The Verification Process

The proof obligations for the new design were derived from the original design requirements, specified in a so–called *Feature Specification Document* (FSD). As illustrated in Figure 2, the requirements formed the basis of both the SDL specification, defining the actual protocol to be implemented, and a series of temporal logic assertions about that protocol.

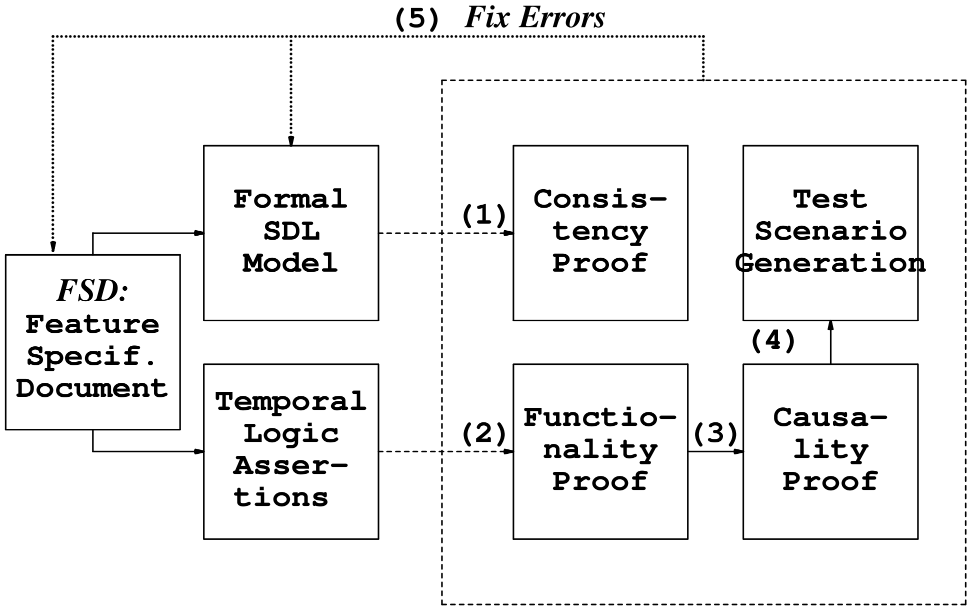

Figure 2 − NewCoRe Verification Process

The first step, called a *Consistency Proof*, was to verify the validity of all pre–defined correctness criteria, and of all user–defined *Invariance* claims, for the SDL specification itself.

The second and third step were used to prove the validity of the *Response* and the *Precedence* claims. The latter two temporal logic claims are of the 'trigger–response,' where a trigger condition (a boolean proposition) must lead to (imply) the truth of a response condition, which can either be a single boolean proposition, or a combination of boolean propositions. The second step was used to show that the trigger condition of each Response and Precedence claim could not remain invariantly false and thus make the corresponding claim vacuously true. This step was called the *Functionality Proof*. The third step was used to prove that *if* the trigger condition of the claim became true, then inevitably the response condition also became true. We called this step the *Causality Proof*.

The verification was completed by using the test–sequence generation capability of *sdlvalid*, to generate sample execution scenarios where trigger and response were visible, to demonstrate adherence to each claim. Though, tis fourth step was not required for verification purposes, psychologically it was perhaps the most important of all steps.

4.2 Controlled Refinement

A design that is ultimately captured in several thousand lines of source text consists of many small design decisions. Groups of decisions can be collected into modules, and combined into a larger structure that delivers the required global functionality. Design is always an iterative process. No single decision is immune to change once it has been added to the design. It is therefore wise, and common practice, to attempt to confine the effect of each design decision to the smallest module possible.

The paradigm that was adopted in NewCoRe to control the refinement process, and to keep track of changing proof obligations, is that of the UNIX® tool `make`. When a design decision is changed, it suffices to type the NewCoRe equivalent of `make` to rebuild the system for the new specifications, *and* to review the validity of all previously verified properties.

The designers, in effect, do not develop code, but the means to build the code, and the means to verify it mechanically. All knowledge about the design is encapsulated into modules that function much like abstract data types in a traditional object–oriented design. A dependency file records the dependencies between all design modules. Each module, furthermore, can have several different encodings, each representing a different level of abstraction.

The designer can choose to replace any given module from the system with another one, that is perhaps more detailed, but equivalent in behavior to the old module under a given set of explicit correctness criteria. In the worst case, to prove the validity of previously verified properties, it may be necessary to repeat all earlier verifications for the new system. In many cases, however, this worst case can be avoided. It will, for instance, be sufficient if a behavioral equivalence proof of the old and the new module alone can be given. [5,8] Since the modules are much smaller than the system, the equivalence proofs are often much simpler to render than the global verifications.

4.3 The Fire Wall

The correctness of a new piece of code that is added to an existing code base in the 5ESS switch is not independent from that environment. Any fault in the existing code, or even a mere unjustified assumption about the behavior of the existing code, can undermine the validity of the most thorough verification efforts for the new code.

A substantial effort in the NewCoRe project was therefore to identify the minimal set of interface points to the remainder of a 5ESS switch that was required for the new code. The assumptions about the possible behaviors of each interface point were them formalized, and integrated into an abstract environment model that became part of the verified design. Formally, therefore, the correctness of the new design was subject to the validity of our assumption about the interface points. Since a formal proof of correctness for the latter (i.e., a formal verification of the remainder of the 5ESS switch) was certainly unfeasible, we decided to include runtime traps in the code produced by the NewCoRe team. The runtime traps encode all formal assumptions about the behavior of the interface points that affect the verifications performed. If any of these assumptions proves, at runtime, to be false, the corresponding trap fires, and issues an appropriate warning.

5. RESULTS

The NewCoRe project was started in 1990 and completed in 1992. The team did not achieve one of its most important original objectives: the production of an independent design that could be compared against an independent design of the mainstream team. The reason for that was so trivial that it is almost embarrassing that we did not foresee it.

About six months into the project, the formal verifications performed by the NewCoRe team started uncovering a first series of logical inconsistencies in the design requirements. At this point, the independence between the mainstream team and the NewCoRe team proved impossible to maintain, since the mainstream team could not be kept unaware of the errors. At this point the mainstream team was asked to adopt the verified formal models produced by the NewCoRe team as the basis for the remainder of their design, and with that the comparison had lost its basis. The role of the NewCoRe team therefore changed from a parallel effort into a supporting effort, assuming the role of a *verification task–force* for the mainstream team, charged with the application of formal methods to verify the further development of the design, instead of producing an independent design, as originally planned.

The NewCoRe effort, nonetheless was considered successful. The final verified model of the design that was produced by the NewCoRe team contained 7,500 lines of non–comment SDL source. A total of 145 correctness properties was formalized in the three classes of temporal logic supported by *sdlvalid* and verified for the final design model. Close to 10,000 verification runs with *sdlvalid* were performed by a team of four to five people over a two year period: an average of more than 100 verification runs per week.

A total of 112 serious design errors were uncovered in the design requirements, by automated verification runs alone, and could be prevented from reaching the product, without requiring field–tests. The most notable result of the NewCoRe effort was, however, the demonstration that almost 55% of all requirements from the original design requirements (i.e., one of the feature specification requirements document given to the two design teams at the start of the project) were proven to be logically inconsistent (unrealizable) by formal verification.

It is unknown, and unknowable, how many of the design errors uncovered by the NewCoRe team after the first six months of the project would have been found by the mainstream team independently at a later stage, without using formal verification. But, we believe to have shown that even if verification would serve only to help designers locate inconsistencies in the requirements earlier in the design cycle, it would be worth the effort.

6. CONCLUSION

We have sketched the characteristics of a formal method, first as an ideal, and then within the practical constraints of a real verification effort that was performed in an industrial environment. Not one of the popular formal methods that we reviewed at the outset of this project was found to live up to the requirements of a scientific method.

For practical reasons, we chose the CCITT specification language SDL as a starting point for the development of a method. The language had to be modified to give it minimally the characteristics of a decidable formal system. The modifications include the addition of a separate syntax for defining a design and for formalizing the correctness claims about that design.

Next, we developed a tool to assist the designer in performing controlled experiments, without imposing or requiring knowledge of formal proof systems. The tool's main task is to produce counterexamples to correctness claims with the highest possible probability of success. It is not always possible to prove the absence of correctness violations with mathematical certainty, at least not for systems of the complexity we have studied.

The original aim of the NewCoRe project was to perform a comparison between two independent design techniques, both subject to the same design requirements and deadlines. It quickly proved impossible, for fairly trivial reasons, to keep the two design efforts separate. Nonetheless, we were able to show that, even for an industrial size design effort, a design based on formal verification techniques is feasible. Our current effort is to develop additional methodology to allow designers to express the original design requirements immediately in a format that can be subjected to verification, before the design cycle itself is begun.

Acknowledgements

The NewCoRe project benefitted from the pioneering efforts of John Chaves and Joanna McCaffrey from AT&T Switch Development, and of Fuchun Joe Lin, now with Bellcore Research. Other people who contributed to the work of the NewCore team include Eric Cheng, James Cheng, and Fred Ng from Switch Development, and researchers Sean Dorward and David Lee.

7. REFERENCES

[1] 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.

[2] Chaves, J.A., ''Formal Methods at AT&T – An Industrial Usage Report,'' *Formal Description Techniques IV – 1991*, North–Holland, Amsterdam, 1992, pp. 83–90.

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

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

[5] Lam, S.S., and Shankar, A.U., ‘‘Protocol verification via projections,’’ *IEEE Trans. on Software Engineering*, Vol 10, No. 4, pp. 325−342.
[6] Manna, Z. and Pnueli, A., ‘‘Tools and Rules for the Practicing Verifier,’’ Stanford University, Report STAN−CS−90−1321, July 1990, 34 pgs.
[7] Saracco, R., Smith, J.R.W., and Reed, R., *Telecommunications Systems Engineering using SDL*, North−Holland Publ, 1989, 632 pgs.
[8] Sifakis, J., ‘‘Property preserving homomorphisms of transition systems,’’ *Lecture Notes in Computer Science*, Vol. 164, 1983, pp. 458−473.