Specification-based Testing of Reactive Software: A Case Study in Technology Transfer

Lalita Jategaonkar Jagadeesan *, Adam Porter†, Carlos Puchol‡, J. Christopher Ramming§, and Lawrence G. Votta*

February 28, 1997

Abstract

We describe a case study in which we tried to transfer a specification-based testing system from research to practice. We did the case study in two steps: First we conducted a feasibility study in a laboratory setting to estimate the potential costs and benefits of using the system. Next we conducted a usability study, in an industrial setting, to determine whether it would be effective in practice.

The case study illustrates that technology transfer efforts can benefit from a greater focus on practitioners' needs, and that this focus helps identify some of the open problems that limit formal methods technology transfer.

We also found that there is often a tension between the scope of the problem to be solved and the specificity of the solution. The greater the scope of the problem, the more general the formal method solution and, thus, the more customization that must be done to use it in a particular environment.

We suggest that researchers limit the scope of the problems they try to solve to minimize the risk of technology transfer failure.

1 Introduction

Formal languages research has had an enormous effect on the practice of software development. Specifically, it has laid the foundation for the tools developers use daily – compilers, case tools, configuration management and module interconnection systems.

Formal methods research, however, has not had the same success. This seems odd since formal methods research is based on general formal language research. This problem has been widely discussed and many articles have pointed out possible causes. We argue that many of these articles are "outward-looking". That is, they suggest that the biggest barriers to transfer lie outside the technology itself.

For example, Sam Redwine and William Riddle [25] claim that it takes about 20 years for technology to get into use. Tom Allen [1] suggests that social factors can lengthen or reduce this interval. We also hear that that practitioner's resist change and that they lack the required mathematical training to use formal methods [13].

Surely, these and other factors play important roles in technology transfer. Nevertheless, we believe that even if all these issues disappeared, formal methods would still be difficult to put into practice. This is because there is a gap between the solutions offered by research and the needs of practitioners.

Our experience in attempting to use a formal method in an industrial setting has convinced us that we must also take an "inward-looking" view. That is, that we must think harder about whether technology really solves practitioners' needs.

We came to this conclusion while trying to introduce a formal specification-based testing method into a Lucent development setting. We found that the greatest barriers to transfer did not come from the developer community

* Software Production Research Department, Bell Laboratories, 1000 E. Warrenville Rd., Naperville, IL 60566 (USA), {lalita,votta}@bell-labs.com

† Department of Computer Science, University of Maryland at College Park. aporter@cs.umd.edu. This work is supported in part by a National Science Foundation Faculty Early Career Development Award, CCR-9501354

‡ Department of Computer Sciences, The University of Texas at Austin, cpg@cs.utexas.edu. This work was partially supported by a Fulbright fellowship at The University of Texas at Austin . This work was performed while the author was visiting Bell Laboratories

§ Innovative Services Research Department, AT&T Laboratories, jcr@research.att.com

(which was motivated to use our work), but came from mismatched between the system and the users' needs. Moreover, we believe that these problems are characteristic of many efforts to transfer formal methods technology.

There are several common ways in which formal methods can fail to meet practitioners' needs.

- Inappropriate simplifications. Formal methods must abstract certain aspects the system. Sometimes, however, these details are very important. For example, a system's inputs are constrained by the physical hardware over which a software system runs. Sometimes, ignoring this detail can create enormous numbers of spurious input sequences.
- Scale issues. As systems get larger they present problems and opportunities that are absent in the small systems usually used in research. For example, state space explosion makes many types of formal methods infeasible in practice. On the other hand many interesting problems don't fully appear in the lab but do so in larger systems (verifying properties of module interfaces).
- Requiring complete and/or stable specifications. Many formal methods assume relatively complete, stable specifications. This is often impossible in practice. Even if it is possible, it would require enormous changes in the way work is done today, and it is risky because small changes in requirements can force large changes to the specifications.

In this article we describe a case study to transfer a specification-based testing method into industrial practice. Our approach was to first conduct a small laboratory study to test out our ideas and then to conduct an industrial case study to exercise the tools in a more realistic setting. From these two studies we draw several lessons about formal methods technology transfer, point out the technical problems that make it hard, and suggest ways to improve the situation.

1.1 Historical Perspective

As often happens in research we built a tool for one reason and then discovered other applications for it. We had been experimenting with software inspection methods. To do these experiments we needed a cheap and accurate way to determine the number of defects in a piece of software. We realized that if we formally specified the code to be inspected, we might be able to create oracles and automatically test it. Therefore, we built a system to do this [18].

We soon realized that this system might be applicable in an industrial setting. We grabbed the nearest development manager and tried to sell our idea to her. She didn't buy it! Instead she suggested that we collect data to identify the strengths and weaknesses of the system, and to estimate the cost of adopting the system and its impact to her group.

1.2 Strategy of the Case Study

To do this we needed to explore the effectiveness of the system, while keeping down our costs and the risk to our industrial partner. Most of our previous empirical work involves controlled experiments of well-established technology. However, since this study involved new technology, we felt that a more descriptive, less controlled approach was needed [19]. Also it was important to find a good compromise between the cost to perform the study, the information learned, and the degree to which we intruded on the projects we would be working with. Therefore, we decided to perform a case study (See Figure 1).

We conducted the case study in two phases [29]. First, we ran a feasibility study in a laboratory setting. To do this we formally modeled the requirements of a small, but real, telephony application, asked several graduate students in computer science to implement them, and then tested the resulting applications using our system. Next we tried to study the usability of the system by deploying it in a live, large-scale software development (the one from which we drew the telephony application used in the first study).

2 Background and Related Work

Reactive systems are those that must respond continually to stimuli from their environment: computation and outputs to the environment are driven by inputs received from the environment. Examples of reactive systems include elevators, traffic controllers, and avionics controllers; most real-time systems are also reactive in nature.

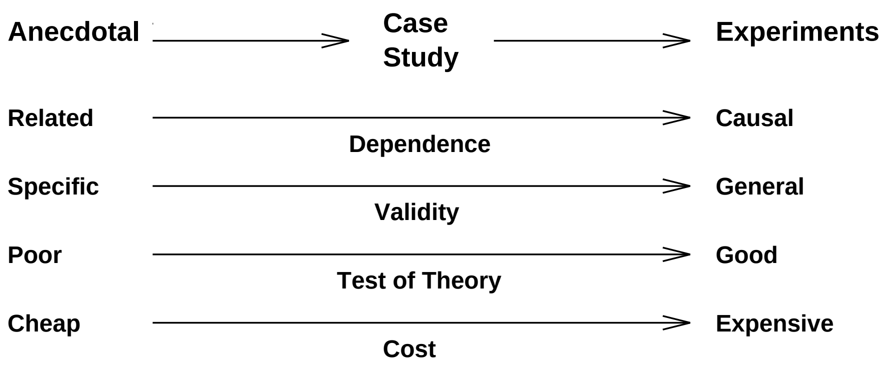

Figure 1: **Spectrum of Empirical Work** The plot depicts the range of empirical work along four important axes; dependence, validity, test of theory, and the cost to perform the study. As you move from left to right these properties change as indicated.

Reactive systems are also ubiquitous in the software for Lucent Technologies' 5ESS® telephone switching system [21], which provides telecommunications services.

Reactive systems are often safety-critical and must be thoroughly tested to ensure that they meet stringent requirements. Since the number of potential input sequences that a reactive system must handle is infinite, much testing is needed to establish confidence in the system. The testing of a typical 5ESS feature, for example, consumes a significant portion of its development resources. Although many other factors contribute to this situation – such as the need to use sophisticated hardware labs and the need to regression test the core system after new features are added – the cost of having people select tests and evaluate test data figures prominently among them.

Despite advances in testing that have lessened the dependence on human effort, we believe that reactive systems have some special characteristics that inhibit the use of these advances and force manual performance of many testing activities:

- Time-dependent behaviors.

  A reactive system's output often depends not only on its current input, but also on the system history. This makes it difficult to calculate the input-output relations needed to evaluate test results.

- Multiple acceptable outputs.

  Many techniques assume that test results are unique. This is untrue for nondeterministic systems and when an application's requirements are underspecified.

- Incomplete specifications.

  In practice it is rare for an entire feature to be formally specified. Consequently, testing techniques that develop oracles from a complete specification of a system are often impractical.

- Low failure rates.

  To gain confidence in a reactive system's reliability and availability it is often necessary to run a large number of tests. As failures become less frequent, the efficiency of having people evaluate test results drops dramatically. Testing techniques should alert developers only when a failure has occurred, rather than require developers to evaluate test results by hand.

2.1 Specification-Based Testing

Informally, specification-based testing refers to a class of techniques in which specifications are analyzed (depending on the specific approach) to develop test cases, execute them with an application, and validate the results.

In our approach requirements are specified as a restricted class of temporal logic safety properties [20]. From these specifications we automatically generate finite state machines (FSMs) that accept the language of input-output traces that violate the safety properties. The resulting FSM's are used to generate test inputs, which are fed to the actual system to determine whether or not its output violates one of the safety properties. If a violation occurs, our tools automatically alert the user and indicate which safety property has been violated. Furthermore, our tools provide an execution trace leading to the violation.

Temporal logic is the basis of our approach, so by definition it supports time-dependent behaviors. Temporal logic naturally describes non-determinism and, therefore, multiple acceptable outputs are easily accommodated. We do not assume that specifications are complete; any well-formed temporal logic safety property can be tested. Most importantly, our approach completely eliminates human involvement in the selection of test data, the development of test harnesses, and the evaluation of test results.

This work was inspired by Dillon&Yu [9], who present a method for testing reactive software against specifications written in a version of temporal logic called Graphical Interval Logic [8]. Properties written in this logic are translated into FSM's whose language is the set of executions that violate the given property; the resulting FSM's are then used to generate test inputs. Dillon and Yu indicate that they are currently developing tools to support this method, and that they will be integrated with Richardson's TAOS [26] test management system.

Parissis&Ouabdesselam [23] present a technique for testing whether reactive software satisfies specifications written in LUSTRE [12], a synchronous data-flow language that can also be viewed as a temporal logic.

We have used standard temporal logic – rather than Graphical Interval Logic or LUSTRE – mainly to take advantage of some temporal logic tools that we had developed in the course of earlier work [16].

Richardson et al. [27] present an approach for deriving oracles from formal multi-paradigm specifications. Our approach is focused on temporal logic safety properties, and oracles are derived automatically.

Our approach is a form of conformance testing – black-box testing for determining whether an implementation exhibits the behavior prescribed by its specification. Many approaches to conformance testing have been proposed, corresponding to a variety of specification languages. For example, Gaudel [10] presents a framework for the testing of algebraic specifications. Brinksma et al. [5, 6] present a theory of testing based on labeled transition systems, and applications to the specification language LOTOS are shown in [6, 24].

3 The Testing Framework And Tools

3.1 The Computation Model

Our tools test reactive applications [11]. Specifically, applications must conform to the *synchrony hypothesis* [4], which implies that applications must appear to operate in discrete "steps." The application receives a set of inputs (input signals), reacts to the inputs by computing and producing a set of outputs (output signals), and then quiesces, waiting for new inputs.

The synchrony hypothesis also implies that the computations performed by the application are atomic with respect to their environment. In particular, no new inputs should arrive from the environment while the application is computing or alternatively, if inputs do arrive, they are registered for processing in the next step.

3.2 The Specification Language

Informally, safety properties stipulate that "something bad never happens." Since reactive systems usually have to respond in a bounded amount of time, liveness properties – which stipulate that "something must eventually happen" – are reduced to safety properties. Consequently, safety properties are sufficient to describe the intended behavior of most reactive systems.

Temporal logic is a well-known formalism for specifying safety properties, and our specification language is based on its propositional linear-time variant [20]. The specifications used by our tool are written with a specially designed notation. As an example, consider the property "the elevator's door is never open while the elevator is moving, and if someone pushes the third floor button then the elevator will reach the third floor in 10 ticks or less". Figure 2 shows a specification of this property[1].

Figure 2 also shows a special *relation* directive that is a feature of our testing system. This directive is used to help the system derive more compact oracles by indicating that two or more signals are mutually exclusive. In our example the directives state that the elevator will never be moving and stopped at the same time, nor will the door be both open and closed simultaneously. The input language also includes an *implication* directive, which indicates that the presence of one signal implies the presence of another.

[1] The special signal `TICK` models the passage of discrete units of time, or steps, not necessarily real time.

```
Inputs: MOVING, STOPPED, OPEN, CLOSED,
        F-3, GO-3;

Relation: MOVING # STOPPED;
Relation: OPEN # CLOSED;

S0 := { (OPEN -> not MOVING)
       and (MOVING -> not OPEN) }
S1 := { F-3 RespondsTo GO-3 In 10 TICK }
P := Always { S0 and S1 }
```

Figure 2: Input Syntax for Temporal Logic

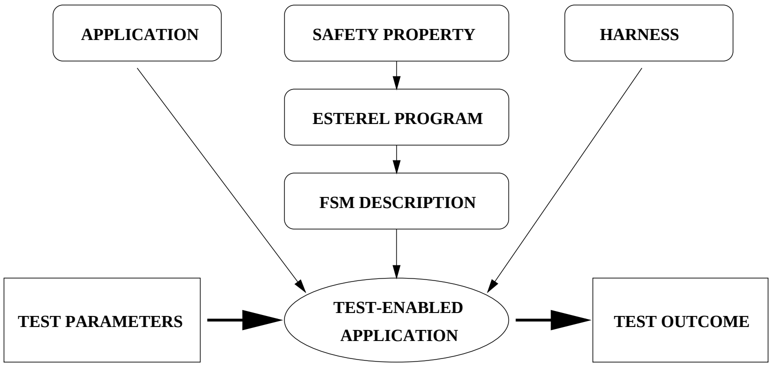

Figure 3: Architecture of the testing system.

These directives allow the compiler to omit some of the oracle's states and transitions, reducing its size. This also prevents the test harness from generating unnecessary test cases.

The safety properties themselves are composed of signals, the standard boolean operators, simple temporal operators: *previous*, *since*, *has-always-been*, *once*, and *back-to*, and a *bounded-response* operator (property `S1` in Figure 2 is an example of a bounded response property). See [20, 16] for formal definitions of the operators.

3.3 The Toolset

We have developed techniques and tools that automatically test whether a software application satisfies temporal logic safety properties. Testing whether an application satisfies a safety property is equivalent to observing whether it has any finite executions that violate the safety property. Thus, our testing system has two goals: to generate test cases that lead to violations, and to identify violations as quickly as possible and without human intervention. To achieve these goals, the testing system has three components (see Figure 3): the *application under test*, the *test harness*, and the *oracle state machines*. These components are automatically assembled to produce an executable object called the *test-enabled application*. This application can then be run with various parameter settings to adjust the number of test runs, the number of reactive cycles per test run, and the format of the test output.

Below we describe the test harness, the oracle state machines, how the test-enabled application is produced and optimizations to the testing process.

3.3.1 Test Harness

The job of the test harness is to drive the testing process and to coordinate the behaviors of the oracle state machine and the application. The test harness is automatically generated from the safety property and a description of the input and output signals.

During the testing process the test harness repeatedly exercises the application. For this to be possible, the application must be designed to conform to the *harness interface*. This interface enables the test harness to

```
structure { /* ... */ } ELEVinputs;
structure { /* ... */ } ELEVoutputs;
void ELEV_set<SIGNAME>(BOOL, ELEVoutputs *);
void ELEV_set<SIGNAME>(BOOL, ELEVinputs *);
void ELEV_test<SIGNAME>(BOOL, ELEVoutputs *);
void ELEV_test<SIGNAME>(BOOL, ELEVinputs *);
void ELEV_RESET();
void ELEV_CLEANUP();
void ELEV(ELEVinputs *inputs,
         ELEVoutputs *outputs);
```

Figure 4: Interface functions generated for an elevator application.

observe the application as well as to influence its behavior.

One aspect of the harness interface is a data structure that the test harness sets and that the application queries to transmit inputs between them. The interface contains a similar data structure for output signals by which the application returns data to the test harness. Both of these structures come with a set of functions for querying and modifying them. The final interface component is a set of functions for initializing, executing, and shutting down the application.

As long as this interface is respected, the application can be linked with the the test harness to create an executable system. A portion of the interface functions (written in C) appears in Figure 4.

3.3.2 Oracle State Machines

In order to generate test cases, our system uses the following important fact about safety properties [30]:

> For any safety property, there exists a finite-state machine whose language is the set of all possible finite executions that violate the property.

We refer to these finite-state machines as *oracles*, and they are the mechanism by which an application's flaws are revealed. In our toolset, oracle state machines are constructed through the following chain of events. First, safety properties are specified by the system engineer using the temporal logic syntax described earlier [16]. Next, as an engineering convenience, our toolset automatically translates the temporal logic formulae into ESTEREL [4] programs. These programs express deterministic finite-state machines, which we extract easily by invoking the ESTEREL compiler. The resulting information is then automatically analyzed and eventually linked with the test harness and the application.

The state machine information includes a list of states, the start state, the accepting states, and a set of transitions labeled with both input and output signals. Each state transition is labeled with a pair $< I, O >$, where I is a set of simultaneous input signals to be provided to the application under test, and O is a possible set of simultaneous output signals produced in response by the application. Therefore, state transitions are based on a combination of the inputs given to the system and the outputs received from it.

The language of the generated state machine is the set of all sequences $< I_1, O_1 >< I_2, O_2 > \cdots < I_n, O_n >$ that violate the safety property. Thus, accepting states of the state machine indicate a violation — the machine is driven into a accepting state if and only if a safety property has been violated.

3.3.3 The Test-enabled Application

The oracle, the test harness, and the application are automatically linked to produce the test-enabled application. This application operates in a simple stimulus-response cycle. First the harness queries the oracle to determine which inputs should be given to the application. Next, the oracle randomly selects a set of inputs from its current state. The harness then invokes the application with these inputs, and waits for the application to produce a set of outputs in reaction. Once these outputs are received by the harness, they are combined with the inputs and are sent to the oracle. From its current state, the oracle then takes the transition that is labeled with this set of signals (this transition exists and it is unique, since the oracle is deterministic). The oracle then moves to the target state of the transition, and the cycle is repeated.

If the oracle reaches an accepting state, the safety property has been violated. Otherwise, the cycle repeats until a violation is detected or the maximum test sequence length has been reached, after which the test is deemed inconclusive and is aborted. The user can set at run time the maximum test sequence, the number of sequences to generate and the format and content of the test results.

As a convenience, the system can be made to report entire test traces. In the event that a violation is detected this allows users to reproduce and analyze the violation using a debugger.

3.3.4 Optimizations

Automated analysis of the FSM allows us to optimize the testing process. One optimization involves separating the oracle's states into *safe* and *unsafe* states. Unsafe states are those from which an accepting state is reachable: all the rest are safe. If the oracle reaches a safe state during testing, the test is aborted. This avoids useless cycling, for instance, when an initialization property is being tested and the application initializes successfully.

Another optimization is to actively avoid safe states by selecting only input signals that have a chance of driving the machine into an unsafe state. However, because each transition depends both on the input signals (controllable by the harness) and the output signals (not controllable by the harness), it may not be possible to avoid all safe states. This optimization is most useful when the application contains an "exit" signal, or contains other signals that change the system's mode in such a way that the safety properties can no longer be violated. The effect of this optimization is to generate longer, more useful test sequences.

Oracle state machines can be quite large. One size-reducing optimization is to use mutual exclusion and implication directives within the specification. This information is passed to the Esterel compiler, which uses it to construct more space-efficient oracles.

3.4 A Small Example

As a small example of this technique, suppose we have a simple elevator in a building with three floors. The inputs to the elevator are `GO-1`, `GO-2`, and `GO-3`, corresponding to request buttons for each floor. The outputs from the elevator are `OPEN` and `CLOSED` – corresponding to the state of the door, `MOVING` and `STOPPED` – corresponding to the motion of the elevator, and `F-1`, `F-2`, and `F-3` – corresponding to the floor the elevator is currently on. We assume that the elevator is on exactly one floor at any given time (if it is between floors, it outputs the number of the floor it last visited). We also assume that it is either moving or stopped (but clearly not both), and its door is either open or closed (but clearly not both). It is initially stopped on the first floor with its door open.

A very basic safety property of most elevators is that when the door is open, the elevator is stopped: that is, there is no execution of the elevator in which both `OPEN` and `MOVING` are simultaneously output.

The set of all possible finite executions (over the elevator's inputs and outputs) violating this property consists of sequences of the form $< I_1, O_1 > \cdots < I_k, O_k >$, where the I_i are any combinations of the inputs, and at least one of the O_j contains both `OPEN` and `MOVING`. This is also the (infinite) language of the oracle finite state machine corresponding to the safety property.

If our tool were testing an elevator application, it would randomly generate a sequence of input sets. For example, the first set of inputs might be $\{\texttt{GO-2}, \texttt{GO-3}\}$, corresponding to people getting on the elevator and requesting floors 2 and 3. This set of inputs is automatically provided to the application. Suppose that, in response, the application generates the set of outputs $\{\texttt{F-1}, \texttt{CLOSED}, \texttt{MOVING}\}$, corresponding to the doors closing and the elevator starting to move. For the next step, the tool for example may (automatically) provide an empty set of inputs to the application under test, corresponding to the lack of any new floor requests. Suppose that the application in response generates the set of outputs $\{\texttt{F-2}, \texttt{OPEN}, \texttt{MOVING}\}$, corresponding to the elevator arriving at floor 2, opening its doors, and continuing to move with its door open, clearly an undesirable situation! Since the output set contains both `OPEN` and `MOVING`, the safety property has been violated. The sequence consisting of the pair $< \{\texttt{GO-2}, \texttt{GO-3}\}, \{\texttt{F-1}, \texttt{CLOSED}, \texttt{MOVING}\} >$ followed by the pair $< \{\emptyset, \{\texttt{F-2}, \texttt{OPEN}, \texttt{MOVING}\} >$ is in the language of the oracle finite-state machine and leads it to an accepting state; hence our toolset automatically reports the violation.

4 The Case Study

In this section we describe a technology transfer case study whose goal was to understand how our specification-based testing system could be moved into industrial practice. The case study was conducted in two parts. The

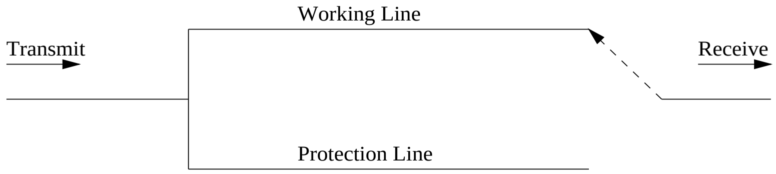

Figure 5: Architecture for 1+1 unidirectional APS

first part involved a small laboratory study to determine the feasibility of the approach. The second part involved an industrial study to determine the system's usefullness in practice.

4.1 Feasibility Study

We conducted the feasibility study to assess the strengths and weaknesses of our testing system. Our specific goals were to evaluate the costs and benefits of the tools and to determine what steps are needed to use them in practice. To conduct the study we developed a testbed of model systems to which we could apply our testing tools. Since we wanted the testbed to be as realistic as possible, we modeled it after the APS system described below.

4.1.1 The Automatic Protection Switching System

As described in [2], communication channels bridging switching systems need to interface to components manufactured by different vendors. In order to facilitate cooperation between components, standards have been established. One of the standards for maintaining connectivity is called "Automatic Protection Switching (APS)" [3]. The idea is to provide more than one line for each communication channel (in switching systems, reliability is often provided by duplicating critical elements). If a line degrades or fails, a backup line, called the "protection line" is used instead. The original version of APS is termed *1+1 unidirectional non-revertive*. In this strategy, a protection line is allotted for each working line (1+1), the decision to switch lines is only made by the receiving side (unidirectional), and a switch to the protection line remains in effect even after the working line clears to an equivalent condition (non-revertive).

Figure 5 shows the architecture for this style of APS. The transmitting side sends the same messages along both the working and protection lines. The receiving side monitors the status of the two lines, and selects one of them to accept messages. Each component may be assumed to fail independently of all others.

A standard redundancy method is used to check the accuracy of transmission of messages. We can assume that the number of erroneous bits received on the working line is continuously recorded, and that correction of messages is not an issue. (Some other protocol will take care of repair or retransmission of faulty messages.)

A line signal is considered *degraded* when it has a bit error rate (erroneous bits vs. total bits) within a dangerous range, typically between 10^{-5} and 10^{-9}. A line signal is considered to have *failed* when the bit error rate exceeds the degraded range, or whenever other hard failures have occurred, such as a complete loss of signal. Either a degraded or failed line may *clear* itself spontaneously. That is, the error rate may decrease to the normal, accepted range without any intervention by operators.

The expected response to a degraded or failed signal on the working line is to (automatically) switch to the protection line. However, that might not be appropriate if the protection line has already degraded or failed. Once a line has degraded or failed it will probably need to be replaced or repaired by a craft technician. Accordingly, operators are provided with a set of commands to change the configuration of the channel:

Remove line: The line is taken out of service.

Restore line: The line is placed in service.

Forced switch: The specified line is selected for communication, regardless of its current state.

Conditional switch: The specified line is selected for communication, as long as it is available and not in the failed state.

The application in this case is a protocol that will maintain the highest quality communication available while responding to operator requests and signal degradation and failure. The standards do not define a protocol, but they include example scenarios for one of the APS paradigms.

The inputs to the system are thus the states of the two lines and the operator commands. The output of the system consists of the state of the switch that selects the current communication line.

As we described earlier, the requirements of the APS were formally specified as part of a formal methods case study by Ardis et al. [2]. We used this specification as the starting point for the following feasibility study.

Like the original APS, the model used in the feasibility study is unidirectional and non-revertive, but has one protection line for every two working lines (i.e., 2+1 rather than the original 1+1). We chose this slightly more complex version in order to exercise a richer set of temporal logic safety properties.

4.1.2 Building the testbed

Along with the initial specification we developed five modified specifications. The modifications incrementally added new input signals, new line quality indications, and new operation semantics. The modifications were introduced to increase the variety and complexity of the safety properties under test. The specifications were between 12 and 17 pages in length and contained between 20 and 35 safety properties. The acceptance test for each application consisted of testing each safety property for 50 runs, each being a test sequence of length 1000.

We asked several developers to implement and test the initial specification. After each application passed acceptance test, we assigned them to different developers and asked them to implement and test the next modification request. We continued this process until all five modification requests were completed.

4.1.3 The Study

To create this testbed we designed and conducted the following study. Our goal was to create a set of code artifacts to be tested by our tools.

Study Setting. We ran this study during Spring 1996 at the University of Maryland. Sixteen graduate students in computer science acted as developers and the entire project took 6 weeks to complete.

Variables. For each modification we captured several dependent variables.

1. Self-reported development effort.
2. The number of test runs needed to pass acceptance test.
3. The results of each test run (i.e., which specific safety properties were violated).
4. The effect of each code modification (i.e., some previously accepted safety properties now violated, no effect, some previously violated properties now accepted – no new violations).

Threats to validity. There are several threats to validity of this study. Since this is a feasibility study we are most interested in threats to external validity.

Threats to external validity compromise our ability to generalize our results. We are aware of the following threats.

- System size. Our applications are very small compared to industrial systems. However, much of this difference is due to the absence of code to support fault tolerance, auditing and logging, and interfaces to the 5ESS system. This shouldn't compromise our test results, but may hide difficulties that appear when testing complex systems.
- Subject representativeness. Our subjects are competent programmers, but may not be representative of professional programmers. That is, they may make different types of errors than professional developers do.
- Development context. Professional developers may have workloads, responsibilities, organizational constraints, etc., that may make this tools difficult to use in practice.

Conducting the Experiment. We conducted the experiment in two phases: training and operation. In the training phase, we gave 6 hours of in-class instruction on temporal logic. We also provided 3 hours of instruction on the algorithms for converting temporal logic safety properties into test oracles.

We gave each student the initial requirements specification, which they implemented and tested within one week. Once a week for the following 5 weeks each student received an implementation generated in the previous week, all previous requirements specifications, and a new specification detailing the intended modification. They again implemented and tested the modification within one week. All the students successfully completed all the modifications.

Each time the students ran the testing tools we captured and timestamped the source code, and gathered testing statistics. By the end of the study the testbed consisted of 30 implementations of 300–500 lines of C code each.

During the study the developers ran the test tools over 200 times. Every time the test tools were run, each safety property underwent 50 test runs with 1,000 inputs per run. Since there were between 20 and 40 safety properties in each specification, each complete test involved 1 to 2 million test cases.

4.2 Observations from the Feasibility Study

After generating the testbed we examined the implementations and the test results, and surveyed the developers to assess the tool's performances and characteristics. We grouped these observations into five categories: testing completeness and efficiency, the nature of errors found, usability from the developer's perspective, usability from the specifier's perspective, and heuristics for generating the testing engines.

Testing Performance. While building the testbed the developers ran the testing tools over 200 times finding many violations. As we will describe shortly, many of the violations occurred only when the applications got into specific states. For example, some violations occurred only when a specific sequence of inputs was received, when a large amount of memory went unreclaimed, or when counters overflowed. These errors would have been extremely difficult to identify through *ad hoc* testing or code reading and finding them would have required vastly more human effort. From this perspective the tools are highly cost-effective.

On the other hand, the tools are clearly not useful for detecting performance inadequacies, system behavior under load (stress testing), or fault-tolerance, all of which are critical for an industrial APS system.

Also, the tools are not necessarily resource efficient. If each computation cycle is lengthy, running vast numbers of tests may be infeasible. In this case more traditional coverage-based testing methods may be more appropriate.

Error Detection. We drew several interesting observations about the kinds of errors made by developers and found by the testing tools. First, the most common errors were failures to handle rare cases, incorrect logic, and requirements misunderstandings. In our experience, relatively few failures resulted from faults appearing at single points in the program, although many testing techniques and studies appear to focus on such faults.

The test results showed two patterns: incorrect logic and requirements misunderstandings that caused failures on nearly every test run (40-50 violations on 50 test runs), and rare cases (violations on 1 or 2 runs out of 50).

Again, this brings up an efficiency trade-off. Given enough time, the tools will uncover problems in handling rare cases without human effort. Coverage-based approaches might find them more quickly, but require more human effort to construct appropriate test cases.

Usability from Developer's Perspective. We surveyed the developers to get their reactions on using the testing tools. Almost all of them were impressed with the speed and ease of generating test cases and running them. Essentially, this involves typing a single command. They also found that replaying the test traces while using a debugger helped them debug their errors quickly.

The biggest dissatisfaction came from having to wait on compiles. Each time an application is modified it needs to be linked with the test engine. This time was noticeable because the applications were small and their compilation time was negligible in comparison with that of the linking phase. This problem could be corrected, for instance, by adopting a client-server model rather than linking the application and test engine into a single executable.

Usability from Specifier's Perspective. Using formal methods requires programming activities at earlier stages of the life-cycle. However, there is little development support for programming at this stage. Our experience bears this out. We made many mistakes in specifying the APS. As with traditional programming, we made syntax errors, forgot to handle rare cases, and misunderstood our requirements. Unlike traditional programming, however, we had little development support.

For example, in one case we forgot to include a potential input in the input specification. The resulting test-engine never generated tests containing this input. Therefore, some safety properties violations went undetected without our knowledge. We found the problem by examining the traces.

When we implemented the small elevator example we accidentally left out an important bounded response formula (our only way to ensure progress). The developers assumed its presence anyway, but one made a logic error that caused the elevator to move to the third floor and stay there forever. Of course, no violations were detected, but the behavior was clearly inappropriate. Obviously, several things went wrong, but the end result was a flawed program that "appeared" to be correct.

Finally, because the tool has several translation steps, errors at one stage sometimes caused failures several stages later. Since building the testing engine is computationally expensive, this led to lots of frustrating debugging and rework of the specifications.

As the three previous examples show, technology such as simulators, syntax checkers, and debuggers will be crucial any time formal methods are used.

Heuristics for Generating the Testing Engine. Since the alphabet for APS has over 20 symbols, building the FSM requires large amounts of memory and computation time. As the specification got more complex we were unable to build the testing engine on a Sparc-4 with 32M of memory. Sometimes we ran out of virtual memory, sometimes we crashed the ESTEREL compiler. To work around this problem we used several heuristics to pare down the state space.

The first heuristic was to to divide large safety properties with conjunctions into their subformulas and test each subformula separately. The FSM's for the subformulas were smaller, but required us to run many more tests.

The second heuristic was to put "mutual exclusion" and "implication" directives in the specifications. The mutual exclusion directives inform the compiler that some signals will not appear at the same time (for instance, the elevator will not be on two floors at the same time and therefore will not emit signals `F-1` and `F-2` simultaneously). This allowed the ESTEREL compiler to omit many FSM state transitions. For example, an APS implementation is guaranteed to receive only one input signal at any step, so all inputs can be written into a mutual exclusion directive.

The last heuristic was to assert that once a safety property was tested it remained valid in the test of subsequent safety properties. Specifically, we used relation and implication directives to assert valid safety properties throughout the remaining tests of an application. For example, the implication relation `OPEN => STOPPED` asserts that when the `OPEN` signal appears, the `STOPPED` signal appears as well. Again, this allowed us to reduce the state space, but required us to re-test all properties when the application was modified.

4.3 Usability Study

In this section we describe an ongoing industrial case study to determine the effectiveness of this approach within a live software development.

4.3.1 The Industrial Applications

We considered the use of our testing technique and toolset on two applications in the Lucent Technologies' 5ESS telephone switching system. The first of these applications was some software in Lucent Technologies' 5ESS telephone switching system. In addition to traditional land-line networks, the 5ESS switch also supports wireless networks such as personal communications systems. The particular 5ESS application we considered is responsible for providing call processing features – such as originations, terminations, location registration, hand over, roaming, and call forwarding – for specific personal communication systems. The second application we considered was a hardware and software combination for interactive voice response systems under development at AT&T.

In both applications, part of the user input is derived from telephone events: establishing calls, disconnecting, and pressing telephone keys. In each case, the application responds to these user events with a variety of activity ranging from call connections to database transactions to speech recognition. Furthermore, both applications consist of a set of processes communicating largely through a messaging substrate.

4.3.2 The Study

Study Setting. Moving from the laboratory setting, we have attempted to apply our testing technique and toolset to the two reactive applications described above. In both applications, we worked with the developers, using the system to test their application. This step is necessary because the system is not a robust production tool.

Threats to Validity. There are several threats to validity in this study. Threats to internal validity compromise our ability to draw conclusions from our results. The major threat we considered was that important data would not be recorded. We limited this threat by being present whenever the system was being used.

Threats to external validity compromise our ability to generalize our results. We are aware of the following threats.

- Domain effects. Our applications are reactive systems that are embedded inside a large hardware infrastructure. These peculiarities may make our applications different from others.
- Development context. Professional developers may have workloads, responsibilities, organizational constraints, etc., that may make this system difficult to use in practice.

4.4 Observations from Usability Study

The experiences we gathered from this study are overwhelmingly positive. They compel us to believe that specification-based testing is indeed cost-effective in its intended setting. However, our system did not integrate easily with these pre-existing applications. Our attempts to use it revealed several issues that must be addressed before we can transfer this technology to industrial software development.

4.4.1 Benefits

Bridges the gap between formal verification and actual industrial applications. Formal verification techniques – such as those based on model-checking [7] – can be very useful as an aid to debugging applications. However, a significant limitation is that most formal verification systems analyze *abstract models* of the application, rather than the implementation itself. Thus, the abstract model must first be described – typically using a manual process. For large applications, this phase is costly and can be error-prone; in particular, the abstract model may not be faithful to the actual implementation. Thus, even if the model is proved to be correct, the actual implementation may still contain errors.

Our approach makes a step toward bridging the gap between formal verification and actual industrial applications, since the testing is based on formal methods but is performed on actual implementations. Furthermore, we believe that introducing our approach into the current process may aid in making further progress in the transfer of formal methods. For example, once software developers become familiar with the the concepts introduced by our toolset, they may well decide to implement applications directly using synchronous formalisms [11] instead of the usual general purpose programming language. These formalisms have the advantage that they have a formal semantics – which allows for automatic verification such as model-checking, and support code generation – which ensures that the automatic verification is closely related to the actual implementations.

Supports testing of individual properties of the application. Our experience shows that testing individual properties of an application individually can be considered complementary to system-wide integration testing. Our approach can bring more focus to the testing activities by highlighting the individual properties that are of interest to developers. Another benefit of our approach is that the scale of the testing activities depends on the properties being tested, not on the size of the application. Since properties can be separated into conjunctions of smaller properties, it is possible to balance computational memory resources against test execution times.

Applies to a large class of reactive applications. Early in this project we expected that the synchrony hypothesis would drastically limit the types of applications we could test. This fear has not materialized. There are certainly many applications for which our tools are inappropriate. However, in practice, most reactive applications can easily be designed to satisfy this hypothesis, since their computations are typically quite short, and inputs that arrive during a computation can be queued. In fact, we argue that large portions of the 5ESS switching system satisfy the synchrony hypothesis for these reasons.

Even when existing switch software is not compatible with our testing technique, it may still be possible to upgrade the software to satisfy the tool's requirements. For example, in a separate study [17], we re-wrote part of the 5ESS software. This new system satisfied the synchrony hypothesis and would have met our testing tool's interface requirements.

Provides a limited scope, but cost-effective testing strategy. We saw that this approach was excellent at finding problems involving rare scenarios. Since this is the most frequent root cause of problems detected in the field [31] (when the cost to repair defects is by far the greatest), the tool is very useful in these cases.

The value of this tool depends on a tradeoff between machine expense and human expense. Our approach is machine intensive but very inexpensive in human terms; therefore, with decreasing computing costs, the tradeoff appears to be worthwhile. Also, even though we sometimes found it necessary to re-engineer safety properties to yield smaller state machines (by indicating signal exclusion properties or simply splitting conjuncts), we found that the ordinary computing environments in a development setting was sufficient to handle the computational expense.

4.4.2 Challenges and Opportunities

We also found several new challenges and opportunities that must be addressed before our system can complete its transitions into practice.

Driving the Test Oracle. Our testing technique involves a form of black-box testing. Therefore, it is appropriate for modules that have clear entry and exit points, whose reactions can be completely controlled by an outside environment, and whose state and operations can be observed through outputs. In other words, the modules must fit into the test harness as stand-alone objects.

In our laboratory study, the applications were designed to satisfy these restrictions. However, in our industrial studies, the existing applications we considered did not satisfy these restrictions. In particular, we found two kinds of limitations.

First, since neither of these applications was a pure software application without mechanical side-effects, executing a generated test in an automated fashion sequence proved impractical. Real phone calls needed to be established in order to test the application; real timeouts would have to be simulated in order to trigger certain interesting behavior. Furthermore, it is the definition of these hardware systems that triggers certain internal signals (such as timeouts, speech recognition events, and recording-terminated-on-silence events) crucial to the behavior of the overall systems. Thus, effective testing using our technique would require working with an abstraction of the application, in which the actual hardware interfaces had been replaced by abstract models of the hardware systems.

The second kind of limitation concerned the environment of the applications. Typically, the environment satisfies many constraints; for example, certain input sequences are never generated by the environment. Thus, purely random testing of the form supported by our tools will typically generate many spurious errors that will never arise in practice. We are currently investigating whether temporal logic is a suitable paradigm in which to express environment constraints.

Despite these limitations, we believe that our approach could usefully and effectively be used in a monitoring mode to evaluate the run-time system behavior. Rather than generate inputs, the system could simply observe passing signals and consult the oracles to determine whether any safety properties have been violated. We believe that such a passive form of testing would still be very valuable; furthermore, it has the added benefit that no instrumentation of the application is needed.

An interesting observation is that object-oriented designs naturally conform to our interface. They have constructors, destructors, and driver methods. Also, since some of the system design in the telecommunications industry is done using object-oriented CASE tools such as Real-Time Object-Oriented Modeling (ROOM)/ObjecTime [28] and O-Charts/O-MATE [15] – an object-oriented extension of Statecharts [14], there is a clear opportunity to create test-enabled applications in conjunction with these executable design tools.

This looks promising because these objects satisfy the synchrony hypothesis by design [28]. The objects are essentially hierarchical finite-state machines, whose inputs are the external inputs to the system and the outputs from other objects. We are currently exploring this connection.

The Signal Mapping Problem. One difficult problem that arises in practice involves mapping specification names onto implementation names. This can be necessary when the specifications are written at a higher level of abstraction than the implementation. The work of Richardson et al [27] takes a significant step towards solving this problem, but more work is still needed in this direction.

Tool Enhancements. Although our current tools are written in C, the test system is inherently language-independent. We are exploring the construction of an oracle server to enable the seamless testing of reactive applications written in different programming languages, on different platforms, and at different geographic locations.

Another enhancement to our tool that might be useful is to allow inputs to be selected with non-uniform weights. Sometimes testers may wish to exercise the systems in conditions closely approximating its intended use, for example, using notions of operational profiling [22]. Other times they may want to overload the system with a certain type or sequence of operations, for example, when performing stress-testing.

Error Detection Effectiveness. One aspect of the tool's cost-benefits is that it is designed to ensure conformance to certain specifications. It finds only errors that can be specified in this paradigm. We need to develop a better understanding of error coverage provided by this approach.

Temporal Logic Specifications. We found in our case study that temporal logic can be a cumbersome notation for describing system properties. For example, our subjects were sometimes confused by the nesting of complicated formulas. We expect that these problems would be exacerbated in an industrial setting, where the safety properties are typically even more complex and contain many clauses corresponding to exceptions. Furthermore, small changes in the requirements can lead to significant changes in the specifications. This is a problem that we believe can be solved by more mature forms of specification where modularity, reuse and scalability are considered as a fundamental part of the design of the methodology. Maintainability and evolution are important parts of any technology to be transferred to industrial settings.

Graphical Interval Logic (GIL) [8, 9] offers a promising graphical alternative to standard text-based temporal specifications. Since our testing approach is based on that [9], it would be useful to explore whether the use of GIL – in place of our temporal logic language – mitigates some of these problems of text-based specifications.

5 Conclusions

To better understand the practical potential of these tools we conducted a two-part case study. The first study was designed to explore the approach's feasibility. This small-scale laboratory study was highly cost-effective and enabled us to thoroughly exercise our tools on a realistic application, while keeping costs to a minimum.

As part of the feasibility study, we formally modeled a common telephone switching application called the Automatic Protection System. We developed five extended specifications and asked a number of developers to implement them. We then applied our testing tools to all of the implementations. In total, we developed 30 implementations of six increasingly more complex APS specifications. (This was intended to simulate the application's evolution over time.) During the study we ran the test tools over 200 times. Each use may involve the testing of between one and two million inputs depending on the number of safety properties in the specification.

We found our tools to be highly effective at finding defects in the implementations. We were pleasantly surprised to find that there were no unexpected tool failures despite the wide variety of properties tested, implementation styles and compilers used to develop the code.

However, we did find that our current implementation still needed work before an industrial study could be attempted. Some of the areas that needed work were the management of the state space and the methods for generating test sequences.

In the second study we focused on the usefulness of the system for testing industrial software. As part of this study we worked with two development projects, helping them to test their software with our system. One of our major findings was the tool was very cost-effective. This was so for several reasons. This approach tests code, rather than abstract models. Therefore, the developers did not have to change their development practices greatly. Also since the approach tests individual properties rather than all properties, developers can manage their testing resources better. Finally, the approach was general enough to be used by more than one project.

From both these studies we drew several conclusions about the process of technology transfer.

The more the transfer process progressed, the more our focus shifted from developing basic technology to customizing it for use at the customer site. The feasibility study gave us a cost-effective way to identify general, context-independent issues, while the usability study was better at identifying specific, context-dependent issues. We want to stress that both studies were important and identified open research questions. Thus, we suggest that researchers should consider similar multi-phase strategies when transferring their technology.

Our system did not integrate easily into the projects in our industrial case study. We feel that customization effort is one of the biggest hurdles to transferring the system into practice. We're not suggesting that the system must run "out-of-the-box", but the more effort it takes to customize it, the less cost-effective the system becomes. We believe that there are many opportunities for research in this area. This also highlights the potential benefits of industry-academic collaborations.

The conclusion we drew from the discussion in the previous two paragraphs is that tools with narrow scope will be easier to transfer than those with broader ones. In other words, technology development may be easier when approached from a top-down perspective, however, technology transfer is much easier when approached from a bottom-up perspective.

Acknowledgements We would like to recognize the efforts of the experimental participants – an excellent job was done by all.

References

[1] T. J. Allen. *Managing the Flow of Technology.* MIT Press, Cambridge, MA., 1977.

[2] M. Ardis, J. Chaves, L. Jagadeesan, P. Mataga, C. Puchol, M. Staskauskas, and J. Von Olnhausen. A framework for evaluating specification methods for reactive systems. *IEEE Transactions on Software Engineering*, 22(6):378–389, June 1996.

[3] Bellcore. Synchronous optical network (SONET) transport systems: Common generic criteria. Technical Report TR-NWT-000253, Issue 2, Bellcore, 1991.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[5] E. Brinksma. A theory for the derivation of tests. In *Proceedings of the Symposium on Protocol Specification, Testing, and Verification*, 1988.

[6] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In *Proceedings of the Symposium on Protocol Specification, Testing, and Verification*, 1986.

[7] E. Clarke and R. Kurshan. Computer-Aided Verification. *IEEE Spectrum* **33***(6)*, pages 61–67, (1996).

[8] L. Dillon, G. Kutty, L. Moser, P. M. Melliar-Smith, and Y. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[9] L. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. *Software Engineering Notes*, 19(5):140–153, December 1994. Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering.

[10] M. Gaudel. Testing can be formal, too. In *Proceedings of International Joint Conference on Theory and Practice of Software Development, Volume 915 of the Lecture Notes In Computer Science*, 1995.

[11] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic, 1993.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[13] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[15] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Symposium on Software Engineering*, pages 246–257, 1996.

[16] L. Jagadeesan, C. Puchol, and J. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *Proceedings of the 7th International Conference on Computer Aided Verification, Volume 939 of the Lecture Notes in Computer Science*, pages 127–140, July 1995.

[17] L. Jagadeesan, C. Puchol, and J. Von Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. *Formal Methods in System Design*, 8(2), March 1996.

[18] L. Jangadeesan, C. Puchol, A. Porter, J. C. Ramming, , and L. G. Votta. Specification-based testing of reactive software: Tools and experiments. In *The Nineteenth International Conference on Software Engineering*, May 1997.

[19] A. S. Lee. A scientific methodology for mis case studies. *MIS Quarterly*, pages 33–50, March 1989.

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer-Verlag, 1992.

[21] K. Martersteck and A. Spencer. Introduction to the 5ESS(TM) switching system. *AT&T Technical Journal*, 64(6 part 2):1305–1314, July-August 1985.

[22] J. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill Book Company, 1987. See pages 227ff.

[23] O. Parissis and F. Ouabdesselam. Specification-based testing of synchronous software. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[24] D. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, December 1990.

[25] S. Redwine and W. Riddle. Software technology maturation. In *Proceedings of* 8^{th} *International Conference on Software Engineeering*, May 1985.

[26] D. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the International Symposium on Software Testing and Analysis*, August 1994.

[27] D. Richardson, S. Aha, and T. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, May 1992.

[28] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object Oriented Modeling.* John Wiley and Sons, Inc., 1994.

[29] L. Votta and A. Porter. Experimental software engineering: A report on the state of the art. In *The Seventeenth International Conference on Software Engineering*, May 1995.

[30] P. Wolper, M. Vardi, and A. Sistla. Reasoning about infinite computation paths. In *IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.

[31] Personal communication. Mary Zajac.