Specification-based Testing of Reactive Software: Tools and Experiments

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

ABSTRACT

Testing commercial software is expensive and time consuming. Automated testing methods promise to save a great deal of time and money throughout the software industry. One approach that is well-suited for the reactive systems found in telephone switching systems is specification-based testing.

We have built a set of tools to automatically test software applications for violations of safety properties expressed in temporal logic. Our testing system automatically constructs finite state machine oracles corresponding to safety properties, builds test harnesses, and integrates them with an application. The test harness then generates inputs automatically to test the application.

We describe a study examining the feasibility of this approach for testing industrial applications. To conduct this study we formally modeled an Automatic Protection Switching system (APS), which is an application common to many telephony systems. We then asked a number of computer science graduate students to develop several versions of the APS and use our tools to test them. We found that the tools are very effective, save significant amounts of human effort (at the expense of machine resources), and are easy to use. We also discuss improvements that are needed before we can use the tools with professional developers building commercial products.

[*] 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

Keywords

Specification-based Testing, Reactive Systems, Temporal Logic, Empirical Studies

INTRODUCTION

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. Reactive systems are also ubiquitous in the software for Lucent Technologies' 5ESS® telephone switching system [13], which provides telecommunications services.

Reactive systems are often safety-critical and must be thoroughly tested to ensure that they meet high-assurance 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.

We have developed a toolset that supports highly automated testing of reactive systems. In our approach, requirements are specified as a restricted class of temporal logic safety properties. From these specifications we automatically generate finite state machines (FSM's) 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.

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.

In order to assess the suitability and advantages of our approach on industrial systems, we have applied our toolset to several implementations of a Automatic Protection Switching (APS) system [2]. The purpose of this system is to manage M redundant resources – such as phone lines – to ensure that $N < M$ of the highest-quality resources are always selected for use. In earlier work, Ardis et al. [1] used temporal logic safety properties to formally specify the APS requirements. Based on this specification, we developed thirty APS implementations and tested them on literally millions of test cases. We found that our toolset automatically reveals violations of the requirements.

This work was inspired by Dillon&Yu [5], who present a method for testing reactive software against specifications written in a version of temporal logic ([12]) called Graphical Interval Logic [4]. 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 [15] test management system.

We have used standard temporal logic – rather than Graphical Interval Logic – mainly to take advantage of some temporal logic tools that we had developed in the course of earlier work [10].

Richardson et al. [16] also presents 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 work extends these both these efforts through an industrial case study.

In the remainder of this article we present our method and toolset, discuss the Automatic Protection Switching system, and our various implementations of it. We then present a case study and empirical observations. Finally, we discuss the challenges and opportunities that would come from using our tools in industrial software development.

TESTING THEORY AND TOOLS

The Computation Model

Our tools test reactive applications [6]. Specifically, this means that applications must conform to the *synchrony hypothesis* [3], which implies that applications must appear to operate in discrete "steps." The application receives a set of inputs, reacts to the inputs by computing and producing a set of outputs, and then quiesces, waiting for new inputs.

It also implies that the computations performed by the application are atomic with respect to their environment. In particular, no new inputs arrive from the environment while the application is computing. Alternatively, if inputs do arrive, they are queued for processing in the next step.

The Specification Language

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 hap-

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

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

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

Figure 1: Input Syntax for Temporal Logic

pen" – are reduced to safety properties. Consequently, safety properties are sufficient to describe the intended behavior of most reactive systems.

Temporal logic is a popular formalism for specifying safety properties [12]. 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 seconds or less". Figure 1 shows a specification of this property.

Figure 1 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.

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 1 is an example of a bounded response property). See [12, 10] for formal definitions of the operators.

The Theory

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, our system uses the following important fact about safety properties [18]:

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

This finite-state machine is a finite-state acceptor of all the violating executions; using the standard view of finite-state acceptors as generators [9], we can hence use these finite-state machines to generate offending executions and then check whether the application under test has the corresponding behavior: if so, the application violates the safety property. We refer to these finite-state machines as *oracles*.

Formally, we represent executions of reactive systems as sequences of pairs $< I, O >$, where I is a set of inputs and O is the set of outputs. Intuitively, I is the set of simultaneous inputs provided to the application, and O is its resulting set of simultaneous outputs. Given a sequence $< I_1, O_1 >< I_2, O_2 > \cdots < I_n, O_n >$, it is an execution of a reactive application if the following holds:

1. From the initial state, the application produces the set of outputs O_1 in response to the set of inputs I_1.
2. From the resulting state, the application produces the set of outputs O_2 in response to the set of inputs I_2.
3. And so forth through I_n and O_n.

Similarly, the language of the oracle finite-state machine corresponding to a safety property consists of sequences of of input-output pairs that violate the property. Our test cases are projections of these sequences onto the input sets.

The Toolset

The testing system has three components (see Figure 2): the *application under test*, the *test harness* and the *oracle state machines*. The *build tools* assemble these components 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.

```
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 3: Interface functions generated for an elevator application.

Below we describe the test harness, the oracle state machines and methods by which the build tools produce the test-enabled application.

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.

During the testing process the test harness repeatedly exercises the application. For this to be possible, the application must conform to the *harness interface*. This interface enables the test harness to 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) appear in Figure 3.

Oracle State Machines

Oracle state machines are the mechanism by which an application's flaws are revealed. They are constructed through the following chain of events. First, safety properties are specified by the system engineer using the temporal logic syntax described earlier [10]. Next, as an engineering convenience, we translate the temporal logic formulae into programs written in ESTEREL [3]. These programs express deterministic finite-state machines, which we extract easily by invoking the ESTEREL compiler. The resulting information is then analyzed and eventually linked with the test harness and the application.

The state machine information includes a list of states, the distinguished start state, the accepting states, and a set of transitions labeled with both input and output signals. An accepting state indicates a violation — the machine is driven into a accepting state if and only if a safety property has been violated. Each state transition is labeled with input and output signals. Therefore, state transitions are based on a combination of the inputs given to the system and the outputs received from it.

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.

The test-enabled application

The build tools link the oracle, the test harness, and the application 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 prop-

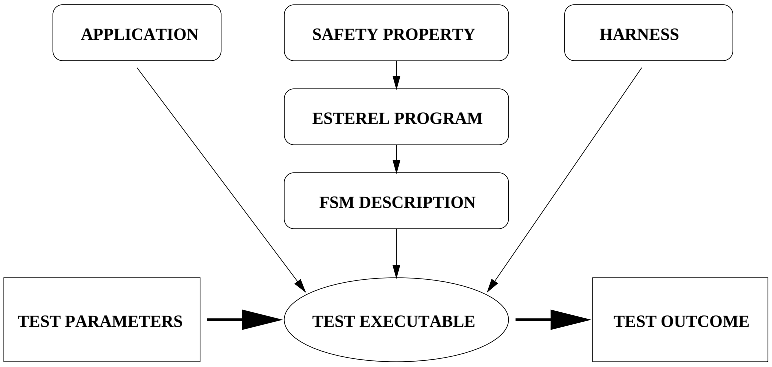

Figure 2: Architecture of the testing system.

erty has been violated. Otherwise, the cycle repeats until a violation is detected or the maximum test sequence length has been reached (this limit can be set at run-time). The user can also set 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.

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 is 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.

A TELECOMMUNICATIONS APPLICATION

The Automatic Protection Switching System

As described in [1], 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)" [2]. 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 4 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. [1]. We used this specification as the starting point for the following feasibility study.

FEASIBILITY STUDY

We conducted a feasibility study to understand the strengths and weaknesses of our testing tools. 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 above. Like the original APS, our model 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.

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.

We asked several developers to implement and test the initial specification. After each application passed acceptance test (we tested each safety property for 50 runs of 1,000 inputs per run), 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.

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.

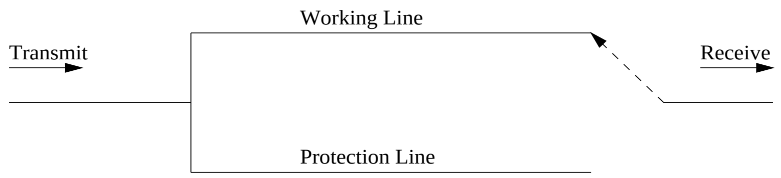

Figure 4: Architecture for 1+1 unidirectional APS

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 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 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.

EMPIRICAL OBSERVATIONS

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 of 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 forever. We then used relation and implication directives to assert valid safety properties throughout the remaining ones. For example, the implication relation `OPEN => STOPPED` tells the compiler 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.

INDUSTRIAL APPLICATIONS: CHALLENGES AND OPPORTUNITIES

The experiences we gathered from this study are overwhelmingly positive. These results compel us to believe that specification-based testing is indeed cost-effective in its intended setting. However, before we can apply this technology to industrial software, there are some issues that must be addressed. In this section we describe some of challenges and opportunities we see ahead.

Synchrony hypothesis

Early in this project we expected that the synchrony hypothesis would drastically limit the types of systems we could test. This fear has not materialized. There are certainly many systems for which our tools are inappropriate. However, in practice, most reactive systems can easily be designed to satisfy the 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 [11], 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.

Modular Design

Our testing technique involves a form of black-box testing. Therefore, it is appropriate for modules that have clear entry and exit points, 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.

One implication is that it may be difficult to test subsystems in the operational context. Some 5ESS subsystems, for example, are difficult to initialize or drive in the absence of the entire 5ESS environment. Nevertheless, our approach might 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.

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 [17] and O-Charts/O-MATE [8] – an object-oriented extension of Statecharts [7], 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 [17]. 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 [16] 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 a 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 [14]. Other times they may want to overload the system with a certain type or sequence of operations, for example, when performing stress-testing.

Costs and Benefits

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.

One the other hand, 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 [19] (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.

CONCLUSIONS

In this article we described a toolset for automatically testing whether software applications conform to temporal logic safety properties. This work was patterned on that of Dillon and Yu, and Richardson et al.

To better understand the practical potential of these tools we conducted a feasibility study. This study was highly cost-effective. It enabled us to thoroughly exercise our tools on a realistic application, while keeping costs to a minimum.

As part of the study, we formally modelled 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 than applied our testing tool 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 have found our tools to be highly effective at finding defects in the implementations. In particular, we were 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 needs work before an industrial study is attempted. The areas needing work are the management of the state space and the methods for generating test sequences. We are also interested in conducting further studies to understand the error coverage provided by this and competing testing methods. None of these issues appears to be intractable, and our near-term future work is focused on these areas.

We believe that this approach offers some exciting opportunities for black box testing of specific software properties – focusing testing effort where it is most important. We are also exploring the integration of these tools with object-oriented CASE tools used in the telecommunications industry.

REFERENCES

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

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

[9] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[10] 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.

[11] 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.

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

[13] 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.

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

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

[16] 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.

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

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

[19] Personal communication. Mary Zajac.