Generating Test Oracles via Model Checking

John R. Callahan
Dept. of Computer Science
West Virginia University
315 Knapp Hall Box 6330
Morgantown, WV 26506 USA
+1 304 293 3607 x3531
callahan@cs.wvu.edu

Stephen M. Easterbrook
NASA Software Research Lab
NASA IV&V Facility
100 University Drive
Fairmont, WV 26554 USA
+1 304 367 8352
steve@research.ivv.nasa.gov

Todd L. Montgomery
Dept. of Computer Science
West Virginia University
316 Knapp Hall Box 6330
Morgantown, WV 26506 USA
+1 304 293 3607 x3507
tmont@cs.wvu.edu

ABSTRACT
This paper describes a method for automatically generating (and re-generating) test oracles during software development using the counter-example generation mechanism found in most model checker tools. Given a state-based specification of a system, our method helps organize test cases into a complete cover of disjoint equivalence partitions on a test space. These partitions are comprised of paths in the test space that conform to specified requirements written in linear temporal logic (LTL) formulae or quantified regular expressions (QRE). The oracles can also be used to drive test executions in cases where the test environment must generate events and conditions in order to force particular behaviors in non-deterministic systems.

Keywords
specification-base testing, formal methods, model checking

INTRODUCTION
Software developers often use models to reason about the design of their systems, but keeping the models and source code in fidelity during development is a difficult task [1]. By fidelity, we mean that the results of analysis, simulation and testing are the same for either representation. Developers and testers need methods for maintaining fidelity as quickly and inexpensively as possible because updating models and code during development does not contribute to software construction and can be an expensive and slow process. Recent trends toward rapid software development emphasize the need for maintaining fidelity in an efficient manner [2]. For example, not only must the code implement behaviors as specified by a model during development, but a model itself may need to change based on discovered limitations of the implementation environment [3]. Maintaining fidelity between the code and models is important as the software evolves because any divergence sacrifices the benefits of formal analysis on the model and may lead to future problems including design errors, inconsistent documentation, and expensive rework.

Typically, a model provides an abstraction for specifying, communicating, and understanding aspects of the expected behavior of a software system. Examples of models include design patterns [4], finite state machines [5], object models [6, 7], functional descriptions [8], flow diagrams, process algebras [9], petri nets, and many other formal and informal notations. While it is possible in some cases to generate code directly from a model, most designers must develop software directly in a common programming language. To ensure that their code reflects their model, developers frequently test their software during development and refine their designs and code based on the results of these tests. Such a process is similar to software prototyping but is done primarily to confirm the viability of implementing a proposed design in a target environment. The use of testing in such circumstances establishes an informal relationship between the code and a design model. Few explicit software development methods, however, exist that support this process of refinement and co-evolution of designs and implementations. As a result, the behaviors expressed by models and code often diverge later in the development lifecycle because fidelity between them is difficult to maintain as changes are made to either representation.

White-box testing allows developers to establish some fidelity between models of their software requirements, designs, and code during development. The output traces of white-box tests, achieved via the use of assertions and monitoring statements (e.g., so-called debugging "print" statements), help to validate that the code behaves in accordance with a model of its design. If an inconsistency between a trace and the specified behavior is discovered, then the model or the code can be corrected as appropriate. For example, Bentley describes the use of monitoring statements in the implementation of a binary search program [10]. In one test case, the input and output of the program are correct, but inconsistencies between the intermediate values of the upper and lower indexes in the execution trace lead him to discover the error in the code. This form of debugging is common and informal, but relies on the existence of an external model of the program's design to compare against the actual behavior. The test oracle (in this case Bentley himself) relies on an intuitive model of binary search to verify that the program behaves

correctly during execution.

Manual examination of test traces and hand construction of test oracles are expensive methods. Because of the non-deterministic behavior of many systems, checking a test trace usually means checking for a pattern of events. A test oracle may accept many trace sequences that constitute valid behaviors for a particular test. In addition, an oracle may have to play an active role in the test environment to force the execution of specific behaviors under test. For example, two executions of the same test of a distributed system may result in different but valid execution traces due to network traffic and processor loads. Often, the trace scanners and test oracles become complex systems themselves in major projects.

Another major problem with testing is achieving adequate test coverage [11] of the specified behaviors of a system and the source code. Advocates of specification-based testing have long promoted the use of formal models as a source for generating test oracles [12, 13]. The use of specification-based testing has three benefits: leveraging the power of formal analysis on the specification itself, understanding of the dimensions of the test space, and automated generation of test cases, oracles, and test environments. Code coverage advocates focus on construction of test cases that will exercise paths within the source code. In general, a combination of both approaches is desirable to ensure that both specified behaviors and the implementation are adequately tested.

We have developed a method for generating test oracles based on model checking [14-16]. Our methods can be used to check traces during or after program execution. In our approach, a state-based model specification can be formally analyzed *and* used to monitor and drive test executions. Our process, called *formal testing*, is a specification-based testing process that uses model checking techniques to verify, organize, and generate white-box tests during evolutionary software development. While a model can be analyzed directly using model checking methods for safety, invariance, liveness, and other properties, it can also be used to generate test oracles and drivers. Our technique can be used to validate execution traces during white-box testing using a model checker as a semantic tableau [17, 18]. We also show how requirements can be specified as linear temporal logic (LTL) formulae [19] or quantified regular expressions (QRE) [20] and used to organize execution traces into disjoint equivalence partitions [21] that constitute a complete cover of a program's design space relative to a set of requirements. Finally, we demonstrate how our approach allows for smooth development and evolution of a system through the automatic regeneration of test oracles. This permits the co-evolution of the requirements, design, and implementation of a system. Our process helps identify inconsistencies that inevitably develop between models and implementations and helps to leverage more powerful forms of analysis faithfully throughout the development process. Combined with existing code coverage methods used during development, our approach helps analysts understand the test space of a problem in terms of both its specification and implementation.

Our approach is most effective on the development of control systems and communication protocols with state-based design specifications and requirements [22]. Although we assume that a state-based model exists a priori, we feel that this is not an unreasonable assumption given the proliferation of state-based modeling techniques in use [5, 23, 24]. We are exploring the use of our method in conjunction with functional approaches [8] by using similar methods for expansion of test cases along branches of proof trees produced by automated theorem provers [25].

MODEL CHECKING

A model checker takes a description of several concurrent, finite state machines as input and effectively analyzes the expanded computation tree for given properties. A *computation tree* is a conceptual structure that consists of a possibly infinite set of all execution paths. For example, consider the state machines shown in Figure 1 for sender (A) and receiver (B) processes using a loss-free version of the alternating bit protocol (ABP) [26]. Each edge denotes a two-character message. The first character represents the originator of the message and the second specifies a sequence number (called the alternation bit). Send operations are underlined. Figure 2 depicts the computation tree consisting of states of execution starting from the initial state of the system at its root. Each state is labeled with the ordered pair corresponding to the sub-state of the sender (A) and receiver (B) processes respectively. A legal path is the set of specific states starting from the root of the computation tree that constitute a valid interleaved execution of processes. In this case, the tree contains all legal execution paths that comprise the joint machine $A \cup B$ where each node is denoted by the composite state of each process.

Model checking is a method by which the computation tree can be searched effectively to ensure that certain behaviors (i.e., paths) with specific properties do or do not exist in the model. Normally, searching such a large structure is infeasible due to combinatorial explosion in the number of operations on states. But model checkers employ various methods to reduce the complexity of their search. For example, redundant states can be eliminated from searches due to the memory-less properties of finite state machines. Other related optimization techniques include partial order reduction [27] and the use of binary decision diagrams (BDDs) in symbolic model checkers [15]. With such methods, some model checkers claim the ability to analyze spaces comprised of 10^{10} to 10^{13} number of unique states [16].

Through the effective expansion of the computation tree by the model checker, the behavior of the model can be analyzed for specific properties. Such properties can be specified as linear temporal logic (LTL) formulae or quantified regular expressions (QRE) that describe sets of

paths (possibly empty) in the computation tree. For example, LTL formulae employ temporal operators [19] including the always (□), next (○), and eventually (◇) operators. The LTL formula

$$\square(\underline{A0} \rightarrow \Diamond A0)$$

describes all paths (if they exist) in which if A0 is sent, then eventually A0 is received. Given this formula, a model checker can determine whether or not any paths exist in the computation tree that satisfy this formula. Such formula can describe non-trivial patterns for sets of paths that have different behaviors but satisfy a common property.

DESIGN EVOLUTION

A finite state model can be used to specify a design solution for meeting a set of requirements. The design model is intended to be an abstraction of the eventual implementation of a system. Required behaviors of the system can be expressed as constraints on a design model. A model checker can be used to determine whether or not the model contains paths that satisfy a specific property. There are three categories of properties that correspond to set of paths in a model:

- no paths in the model should exhibit the property (i.e., a safety property)
- all paths in the model should exhibit the property (i.e., an invariance property)
- some paths in the model should exhibit the property (i.e., a liveness property)

Safety requirements are those properties which *none* of the paths in a model satisfy. To check a model for the absence on all paths of specific behavior means that effectively all paths in the model have to be explored. This corresponds to our intuitive notion that in order to check for the absence of a property, then exhaustive testing of all paths for a safety property is necessary but often infeasible. An invariant property is one that *all* paths in the model must satisfy. An invariant property is the logical complement of a safety property. Like safety properties, invariant properties require an effective search of all paths in the model to determine its presence on all paths.

Version 1: Loss-free ABP

In the loss-free ABP example in Figures 1 and 2, we can express some requirements as temporal formulae:

$$\square(\underline{A0} \rightarrow \Diamond A0) \qquad [R_1]$$

$$\square(\underline{B0} \rightarrow \Diamond B0) \qquad [R_2]$$

As stated before, the requirement R_1 specifies that the message A0 is eventually received if sent. Requirement R_2 specifies that an acknowledgement is always received if sent. Both properties identify a set of infinite paths in the computation tree. Indeed, in the loss-free model both properties hold for any path in the computation tree. Furthermore, each path in the computation tree in Figure 2 is an infinite path. For purposes of test generation, however, we need only to consider *finite paths* and *fixed cycles* in any model that exhibit the desired properties. Fortunately, the counter-example mechanism in most model checkers produce only finite paths and fixed cycles so that we can reason about properties of finite test traces.

Version 2: Adding Message Loss

Both properties R_1 and R_2 will be satisfied by exactly the same paths in the loss-free model because all messages and acknowledgements are delivered on every path. An implementation of the protocol, however, will probably encounter problems such as message loss and starvation in realistic environments. We can add these faults to the model to reflect more pragmatic conditions. This means that there will exist paths in which loss of a message deadlocks the protocol. Hence, properties R_1 and R_2 will not be satisfied by all paths but only by *some* paths in the computation tree. This situation, however, is desirable from the viewpoint of generating test cases because *all other paths* will satisfy the negation of the property. These two sets of paths can be used as different test cases with which to exercise an implementation.

Given properties that are satisfied by some paths in a model, we can use the counter-example generation mechanism to produce test cases for specific behaviors [16]. We assume that a partial, finite-state model of the system exists and that system requirements can be stated as LTL or QRE formulae. A SPIN model is specified in the Promela language as a finite set of asynchronous, concurrent processes that interact through shared variables and communication channels (a special case of shared variable). SPIN translates LTL and QRE formulae into Büchi automata [28] to determine if a behavior is contained in the model (i.e., a path exists in the computation tree). If the Büchi automata terminates or ends in an accepting state or cycle[1], then the search succeeds (the property was discovered). To search for the violations of properties such as R_1, the property is negated and then converted to the Büchi automata. If $\neg R_1$ terminates or occurs infinitely often in a cycle, this means that a path in the model exists such that A0 is never received. Furthermore, SPIN will produce this path as a counter-example and others paths if they exist that exhibit the same property. For example, SPIN will produce the sequence of events (where a double underline means that a message is lost):

$$\underline{B1} \rightarrow \underline{A1} \rightarrow \underline{B1} \rightarrow \underline{B1} \rightarrow \underline{B1} \rightarrow \ldots \underline{B1} \ldots$$

as a counterexample path that violates the property R_1 (i.e., satisfies the property $\neg R_1$). In this case, the system will deadlock because the acknowledgement message B1 from the receiver is always lost. Conversely, the cycle

$$\underline{B1} \rightarrow \underline{A1}^* \rightarrow \underline{B1} \rightarrow \underline{A0} \rightarrow \underline{B0} \rightarrow \underline{A1} \rightarrow \text{repeat from } *$$

satisfies R_1. All paths in the computation tree satisfy either

[1] An accepting cycle is a cycle in which an accepting state occurs infinitely often.

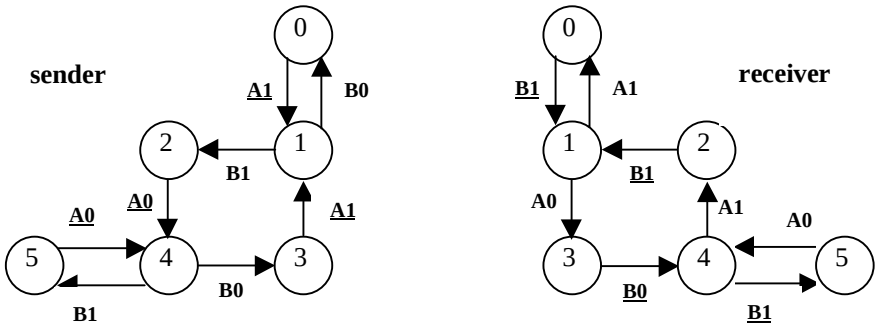

Figure 1: State machines for sender and receiver using alternating bit protocol (ABP)

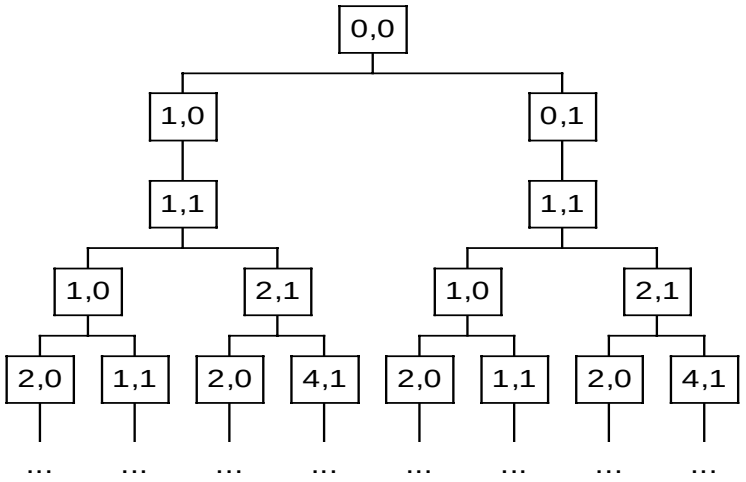

Figure 2: Infinite, joint computation tree for ABP example (without loss)

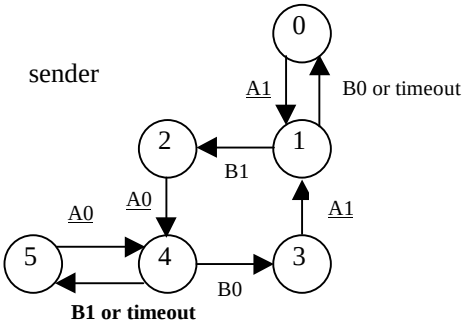

Figure 3: Modified sender with timeouts for retransmissions

R_1 or $\neg R_1$. This partitions the computation tree into two types of paths. If we modify the receiver to lose messages

as well, then we can also partition the computation tree in two partitions relative to properties R_2 and $\neg R_2$. By combining these properties and their negations pair-wise, we can construct four partitions on the computation tree:

$$R_1 \wedge R_2,\ R_1 \wedge \neg R_2,\ \neg R_1 \wedge R_2, \text{ and } \neg R_1 \wedge \neg R_2$$

These cases represent a complete cover of disjoint[2] partitions on infinite paths in the computation tree relative to the requirements R_1 and R_2. Some of these partitions will be empty because they are logically inconsistent. For instance, the partition $R_1 \wedge \neg R_2$ is empty because it is impossible for all sent messages to be received on paths on which acknowledgements are lost. Only the first and last partitions above are non-empty in this case. From a design perspective we can use model checking to confirm which partitions are empty and non-empty to ensure that a model expresses our intentions. If a partitioning analysis on a model does not satisfy our required behavior, we can refine the model until we achieve the desired coverage [29].

Version 3: Avoiding Loss using Retransmissions

In the previous section, the behavior expressed by paths in the non-empty partition $\neg R_1 \wedge \neg R_2$ represent those cases in which messages are lost. We can use paths in this partition to elaborate test cases that exercise the implementation under such fault conditions. We characterize paths that belong to partitions such as $R_1 \wedge R_2$ as *nominal paths* because in general they represent non-fault cases. We characterize other partitions and paths in them as *off-nominal* when they exhibit fault or error conditions. Non-empty partitions that exhibit off-nominal behaviors are useful for generating "stress" tests on the implementations.

We can address the issue of loss in the model by further modifying the sender machine to include retransmissions upon timeout. Figure 3 shows another version of the sender with timeouts that permit retransmissions. But this does not eliminate the problem of lost messages. The partition $\neg R_1 \wedge \neg R_2$ is still non-empty. This is because messages may *always* be lost. This is a realistic case since network links break and sites often fail in distributed systems. Most model checkers eliminate the problem of infinite loss of messages (usually caused by starvation of a process in the model) by allowing specific events to occur infinitely often within *all* cycles of the computation tree. This approach, called *fairness*, eliminates cases where a message is always lost by specifying that eventually a message is delivered after a finite number of retransmissions. As testers,

[2] The proof that CCC partitions are disjoint is by contradiction: assume that a path P satisfies at least two coverage properties C1 and C2. But C1 and C2 will differ on at least one requirement and its negation by definition of the CCC set. A path cannot satisfy a requirement and its negation. Therefore, the assumption is false and CCC partitions must describe disjoint sets of paths in the model.

Partition number	Conjunctive complementary closure (CCC)	
1	R_1	R_2
2	R_1	$\neg R_2$
3	$\neg R_1$	R_2
4	$\neg R_1$	$\neg R_2$

Table 1: Equivalence partitions on R_1,R_2

however, we are interested in the generation of test cases that will exercise *all* types of nominal and off-nominal execution behaviors including infinite loss. In practice, we will use fairness only during formal analysis, but we disable fairness mechanisms intentionally to generate test oracles for off-nominal partitions.

The above scenario involves the evolution of a design model based on the examination of partitions in the computation tree. If an implementation is being coded in parallel with this specification, we can use the model to reason about expected and unexpected behaviors that should or should not be handled by the implementation. In this case, the model helps us identify two important behaviors: paths in which messages are eventually delivered and those which deadlock due to infinite loss. In the next sections, we describe a method for automatically generating test oracles that force an implementation to exercise paths in specific partitions. For off-nominal cases such as infinite loss, a successful test might mean that an implementation of the ABP system fails on a timeout condition rather than deadlock. Model checking allows us to identify the complete test space and automatically construct and maintain a complete and disjoint test cover even as the model specification and implementation evolves.

AUTOMATED TESTING USING MODEL CHECKING

In this section, we demonstrate that we can co-evolve a model and its implementation so that they are in fidelity with one another during development. This is accomplished through iterative analysis and refinement of the model so that the partitions of a computation tree are empty and non-empty as expected by the designer. Next, we can automatically generate test oracles for each partition even after a model is changed. Each oracle can be used to force the implemented system into a specific behavior and check if the system behaves within the bounds of the partition expected by the oracle. If the code does not behave according to the model, this reveals an error in either the model or the implementation. Each failure must be examined individually to determine the source of the inconsistency. By resolving these inconsistencies, we can keep the model and code in fidelity during development, better leverage the benefits of formal analysis on the model, and reduce the cost of testing.

Most model checkers include mechanisms for producing

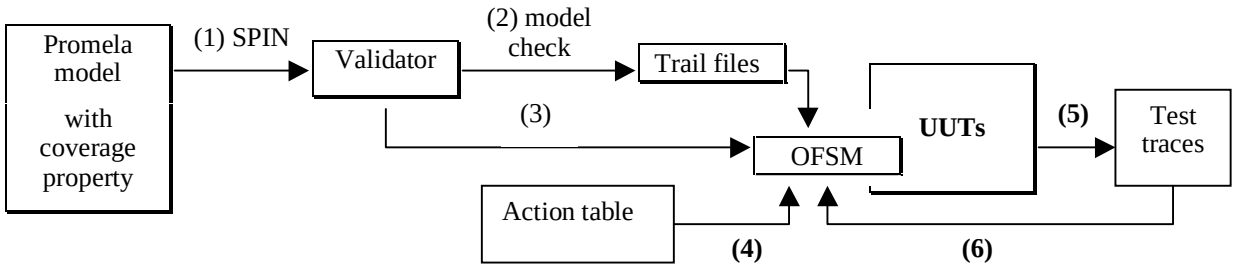

Figure 4: Formal testing process diagram

counter-examples if some paths in a computation tree exhibit a given property. The SPIN [16], Murphi [14], and SMV model checkers [30], for example, will produce counter-examples when paths exist in the computation tree that violate assertions or satisfy temporal formulae. Using this mechanism, we can purposely cause the SPIN model checker to generate counter-examples on demand for a given coverage property. The SPIN model checker will produce finite paths or fixed cycles that exhibit a given property if it exists in the computation tree. These counter-examples serve as *test templates* for test oracles that drive and verify an actual test sequence on an implementation. By composing these templates into a new finite-state machine, we can create a test oracle that monitors test executions for specific patterns of behavior and can also be used to drive the test process. Our test process is illustrated in Figure 4. The next sections describe this process in detail.

Equivalence Partitioning

Test analysts use equivalence partitioning on the input space of a program to minimize the number of tests needed to cover all expected behaviors of a software system under test [21]. Creating partitions, however, is usually an informal and difficult task. Traditionally, the input space is partitioned roughly into overlapping subsets based on some categorization of the expected output behavior of the program under test. In white-box testing, the expected behavior includes the intermediate values of internal variables as shown in the execution trace during the test.

Our approach involves partitioning paths in a computation tree based on combinations of requirements and their logical complements. The requirements are encoded as LTL formulae and quantified regular expressions. We construct a partitioning based the conjunction of all requirements and their complements as shown in Table 1. This partitioning is called the *conjunctive complementary closure* (CCC) of a set of requirements. Each combination is called a *coverage property* because it describes a unique set of paths. In general, the number of partitions is 2^n but some partitions will be empty depending on conflicts between requirements. For example, the two requirements R_1 and R_2 in the ABP example form a complete cover of the computation tree comprised of four disjoint partitions, two of which are empty. While the partitions created by a CCC are disjoint, this only applies to complete paths in the computation tree. Finite prefixes of paths may fall into one or more partitions.

Given a state-model written in Promela and a coverage property, we first use SPIN to produce a validator (step 1 in Figure 4). A validator is an executable model checker for a given coverage property (called a "pan" file in SPIN). When executed, a validator will produce a set of counter-example paths, called *trail files*, for a particular coverage property (step 2 in Figure 4). A trail file represents a single counter-example comprised of a path in the partition that exhibits the partition's behavior. SPIN can produce multiple trail files by supplying a command line argument to the validator to produce up-to a specified maximum number of counter-examples if found.

Test Oracle Generation

For each validator, we can produce an oracle (step 3 in Figure 4). This is accomplished by extracting the joint finite state machine from the validator using a command line option. This joint machine is a union of state machines for all processes in the model specification including the Büchi automata. From this extracted machine, we produce an oracle FSM (OFSM). The OFSM will serve as both a test driver and oracle for that partition.

Next, the OFSM reads the trail files produced by the counter-example mechanism for that partition and marks all paths in the state machine that are traversed by any trail file. The reason for the state marking is so that when the oracle acts as a driver, it can follow paths in its state machine that lead to behaviors contained in the oracle's partition. Without this information, the oracle does not know which path to take at non-deterministic points in the FSM to reach an accepting state.

For example, consider the branch point at which an oracle for the sender must decide whether to send a message or induce a message loss. In this case, the receiver is the unit-under-test and the oracle is acting as the sender. The oracle wishes to test for infinite loss conditions. At the branch point, the oracle can either send a message or cause the loss of a message. If it chooses the latter, it returns to the same state and can eventually deliver the message to the receiver.

But if we mark only paths of infinite loss, the oracle will always cause infinite loss. The implementation should eventually timeout and abort under such conditions. This would constitute a "successful" stress test of the implementation.

Test Execution

Before test execution, we must first identify what processes in the implementation represent the units-under-test (UUTs) and which processes will be represented by the OFSM. In an external file, we keep an association list of driver actions and checks for each transition event in the model (step 4 in Figure 4). When the oracle needs to execute an action to drive a test or check for the occurrence of an event, it looks up the action to execute based on the transition event in the model. For example, the SPIN event A0 might be associated with a procedure call in the action file that sends a packet or generates a bus event.

This approach implies the existence of a sophisticated test environment that drives UUTs by sending them events and can monitor responses from UUTs. It also assume assertions and monitoring must be embedded in the code to examine intermediate events and values during execution. Our case studies based on this approach have either employed a form of software bus [31], involved snooping messages in network-based distributed systems, or on advanced bus-based spacecraft implementations [32]. When a test execution fails to terminate in an acceptable state, it may be that an intermediate event or value was not captured. The oracle can detect this, but the fix may be to insert the necessary assertions or monitoring statements to complete the test run. This is a further aid in keeping fidelity between the model and code during development because it makes programmers consider where in their code they must insert such mechanisms in order to test their components relative to the system model. In addition, we are currently exploring generic techniques for scaffolding other software architectures and patterns that are not based on message handling systems.

Finally, when the action file is complete, each OFSM can be configured with information about the UUTs and tests for that partition can be executed (step 5 in Figure 4). Each OFSM will check for conditions, receive events, and generate events for all acceptance sequences marked in its state machine. If the behavior of the UUTs deviates from a specific sequence but remains within the acceptance path of any other path in the partition space for that oracle, the oracle continues to execute the test. However, if the behavior of the UUTs leaves diverts from the acceptance paths in the partition space or terminates in a non-accepting state, the oracle flags a test failure and produces a counter-example of its own for the errant test run.

Since there may be multiple valid paths in a partition, an oracle can be executed several times to force the behavior of different marked paths. Each test execution results in a *test trace* that records the actual path of the OFSM oracle/driver and UUTs during each path execution. These test traces can be analyzed off-line to verify their membership in a particular partition (or not). Traces are also used to minimize re-testing efforts as described in the next section.

Re-Testing

One of the major benefits of this approach is that the entire test suite can be regenerated when the model changes. Additional actions may need to be specified in the action file, but all trail files and OFSMs can be regenerated automatically. No other steps are needed to restructure test cases or reprogram oracles based on changed specifications. Additional requirements will cause repartitioning to occur based on new coverage properties. Relative to design model changes, however, changes and additions to the requirements are less frequent.

In addition, we can economize testing by marking paths in the new OFSMs that have been executed in previous OFSMs (step 6 in Figure 4). Assume that the design space of a model is partitioned into OFSMs P_1, P_2, … Multiple executions of oracle/driver P_i will produce test traces $T_{1,1}$, $T_{1,2}$, …, $T_{2,1}$, $T_{2,2}$, … If the model is changed to yield new partition oracles P_1', P_2', … then we can mark paths in P_i' that have been tested by oracle P_i. This is done by marking paths in P_i' that correspond to sequences recorded in test traces $T_{i,1}$, $T_{i,2}$, … If a previous test trace does not correspond to any sequence in the new OFSM, then it is discarded since similar behaviors will have to be re-tested as a result of model changes. This re-test minimization process identifies new paths created by model changes and assumes that the implementation remains unaltered. We are still exploring the use of code coverage analysis methods to minimize re-testing partition paths based on changes to the source code.

Consider the change to the ABP model when we added retransmissions based on timeouts. We can mark paths in the new FSM with traces created by the oracle for the coverage property R_1^R_2 in the loss-free model. In a subsequent test run of the R_1^R_2 oracle, loss-free paths would *not* be tested because they still exist in the partition on the new model. Instead, paths in the R_1^R_2 partition that contain finite lost messages would be tested first.

AN EXAMPLE

We have employed this approach on some simple graphical user interface (GUI) programs via a GUI testing tool called JavaStar [33]. JavaStar is able to act as a test environment for user interface applications written in Java. The generated OFSM takes the form of a script that calls on other JavaStar procedures to exercise events on the graphical components of the application. The OFSM exercises sequences of events that represent coverage properties of the test space.

The canonical JavaStar application is a tool that displays entries in a database of addresses. The GUI is comprised of a record display along with buttons for opening a database (**OPEN**), saving a database (**SAVE**,**SAVEAS**), closing a database (**CLOSE**), adding a new record (**ADD**), removing

a record (**REMOVE**), clearing the record display (**CLEAR**), and finding an entry by name (**SEARCH**). A state model of this system consists of two processes: a data store and the user interface. Some requirements for the system include the following formulae:

$$\Box(\text{AddRecord}(x) \rightarrow \Diamond\text{FindRecord}(x))$$

$$\Box(\text{RemoveRecord}(x) \rightarrow \neg\Diamond\text{FindRecord}(x))$$

These properties represents a *nominal* behaviors, because it is not always true that an added record can be found or a removed record cannot be found. This is because the user could REMOVE the added record or ADD a removed record before a subsequent SEARCH operation. It is unimportant that all paths in the model do not satisfy these properties. It is important, however, that we identify and understand the sources of off-nominal cases. Such cases are valuable test cases for stress testing an implementation. For the two formulae shown, our model quickly identifies the non-empty partitions for all cases. We then construct oracles for these partitions to see if

1. A record can be added to the database and then search for;
2. A record can be removed from the database, searched for an not found (i.e., an error message must appear in a window);
3. A record can be added to the database, removed, and not found in a subsequent search;
4. A record can be removed from the database, added again, and successfully found;
5. A record can be added to the database and then the database can be closed.

The technique is also useful for checking that a modified database is saved before closed (i.e., an error message appears in a window asking the user if they really want to quit and lose new records). We have found that the use of model checking allow us to organize the many confusing interactions of features into orderly test cases. While we are not able to exhaustively test all cases, the partition oracles provide an understanding of the complete test space.

The JavaStar tool is an environment that is highly amenable to our approach since it relies on test scripts (.jst files) that can contain sub-test procedures that drive events (e.g., simulate button clicks) and check for conditions of display values (e.g., the result displayed in a text field). We are able to generate non-trivial OFSMs for non-empty partitions that completely exercise an application relative to the stated high-level requirements.

PREVIOUS WORK

We previously used a manual version of this technique on the specification and implementation of the Reliable Multicast Protocol (RMP) [34]. We manually constructed test oracles using counter-example paths produced by the Murphi model checker. In this case, we hand-crafted test scenarios for classes of paths that we were interested in testing. If the model changed, we had to compare differences in the new counter-examples with the old counter-examples and manually update the scripts. The addition of the action file for mapping SPIN events (i.e., assignments, message sends and receives, checks for conditions) to test driver actions was a tremendous help.

One of the early conceptual barriers in our work was the initial and incorrect assumption that the all paths on the model had to satisfy all safety, invariant and liveness properties. When we relaxed this assumption, it made sense that model checking should help us explore the design space and generate unique test cases, rather than ensure complete correctness of a design. It also helped to remove "artificial" assumptions and simplifying assumptions in the model that corrupt fidelity with the implementation. The models we build are easier because they are naïve: they exhibit race conditions, starvation, and deadlock problems on some paths. We eventually realized that identification and classification of these paths in oracle partitions was important to the test and analysis process.

CONCLUSIONS

Our method has already been proposed for organizing test suites on several new projects that employ a message-based architecture and whose specifications contain complex state machines in several subsystems. Like the GUI example, the use of LTL and QRE partitioning allows the surreptitious discovery of interest cases that designers did not consider a priori. Formal analysis. This reinforces some of our previous work that shows that formal methods are not so valuable for the analysis they render, but rather the incremental understanding of the problem space that they let the designer explore prior to and during implementation [22].

Our approach is most effective on the development of control systems and communication protocols with state-based design specifications and requirements. Although we assume that a state-based model exists a priori, we feel that this is not an unreasonable assumption given the proliferation of state-based modeling techniques in use. We are exploring the use of our method in conjunction with functional approaches by using similar methods for expansion of test cases along branches of proof trees produced by automated theorem provers.

Testing remains a powerful and intuitive approach to ensuring the quality and reliability of software, but testing also has serious limitations. We believe that by managing testing via a formal methods we can reap the benefits of formal analysis on a model that is kept in fidelity with the code. The model can be analyzed for invariant and safety properties that are difficult to test for completely in an implementation. The composition of both testing and formal methods in this manner has great potential since both approaches complement the strengths and weaknesses of the other method.

ACKNOWLEDGMENTS

This work is funded by NASA Cooperative Agreement NCC 2-979 at the NASA/Ames IV&V Facility in Fairmont, WV and in part by DoD grant DAAH04-96-1-0419 monitored by the Army Research Office.

REFERENCES

1. Murphy, G.C., D. Notkin, and K. Sullivan, *Software Reflexion Models: Bridging the Gap between Source and High-Level Models*, in *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium*. 1995. p. 18-28.
2. McConnell, S., *Rapid Development: Taming Wild Software Schedules*. 1996, Redmond, WA: Microsoft Press.
3. Swartout, W. and R. Balzer, *On the Inevitable Intertwining of Specification and Implementation.* j-CACM, 1982. **25**(??): p. 438-440.
4. Gamma, E., *et al.*, *Design patterns: Elements of Reusable Object-Oriented Software*. 1995, Reading, MA: Addison-Wesley.
5. Heitmeyer, C., B. Labaw, and D. Kiskis. *Consistency checking of SCR-style Requirements Specifications*. in *Second IEEE International Symposium on Requirements Engineering*. 1995. York, UK.
6. Rumbaugh, J., *et al.*, *Object-Oriented Modeling and Design*. 1991: Prentice-Hall.
7. *Unified Modeling Language (UML) Summary*, . 1997, Rational Software Corporation: http://www.rational.com/uml.
8. Stocks, P. and D. Carrington, *A Framework for Specification-based Testing.* IEEE Transactions on Software Engineering, 1996. **22**(11): p. 777-793.
9. Jackson, D. and E.J. Rollins, *A New Model of Program Dependencies for Reverse Engineering*, in *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations*. 1994. p. 2-10.
10. Bentley, J., *More Programming Pearls*. 1988, New York: Addison-Wesley.
11. Piwowarski, P., M. Ohba, and J. Caruso, *Coverage measurement experience during function test*, in *ICSE15*. 1993, IEEECSP. p. 287-302.
12. Howden, W., *The Theory and Practice of Functional Testing.* IEEE Software, 1985. **2**(5): p. 6-17.
13. Poston, R.M., *Automated testing from object models.* j-CACM, 1994. **37**(9): p. 48-58.
14. Dill, D., *et al. Protocol Verification as a Hardware Design Aid*. in *IEEE Conference on Computer Design: VLSI in Computers and Processors*. 1992: IEEE Computer Society Press.
15. Clarke, E. and E. Emerson. *Characterizing Properties of Parallel Programs as Fixed Points*. in *7th International Colloquia on Automata, Languages and Programming*. 1981: LNCS 81.
16. Holzmann, G., *Design and Verification of Computer Protocols*. 1991: Prentice-Hall.
17. Dillon, L.K. and Q. Yu, *Oracles for Checking Temporal Properties of Concurrent Systems*, in *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations*. 1994. p. 140-153.
18. Dillon, L. *Generating Oracles from your Favorite Temporal Logic Specifications*. in *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 1996. San Francisco, CA.
19. Manna, Z. and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. 1992: Springer-Verlag.
20. Naumovich, G., L. Clarke, and L. Osterweil. *Verification of Communication Protocols Using data Flow Analysis*. in *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 1996. San Francisco, CA.
21. Myers, G., *The Art of Software Testing*. 1979, New York: John-Wiley.
22. Easterbrook, S. and J. Callahan. *Formal Methods for V&V of partial specifications: An experience report*. in *Proceedings, Third IEEE International Symposium on Requirements Engineering (RE'97)*. 1997. Annapolis, Md.
23. Heimdall, M. *Experiences and Lessons from Analysis of TCAS II*. in *International Symposium on Software Testing and Analysis (ISSTA '96)*. 1996. San Diego, CA.
24. Harel, D. and E. Gery, *Executable object modeling with statecharts*, in *ICSE18*. 1996, IEEECSP. p. 246-257.
25. Kancherla, M.P., *Test Generation via Automated Theorem Provers*, in *Computer Science*. 1997, West Virginia University: Morgantown, WV.
26. Bartlett, K., R. Scantlebury, and P. Wilkinson, *A note on reliable full-duplex transmission over half-duplex lines.* Communications of the ACM, 1969. **12**(5): p. 260-265.
27. Peled, D. *Combining partial order reductions with on-the-fly model checking.* in *6th International Conference on Computer Aided Verification*. 1994. Stanford, CA.
28. Alur, R., C. Courcoubetis, and D. Dill. *Model-checking for real-time systems.* in *Fifth*

Symposium on Logic in Computer Science. 1990. Philadelphia, PA.

29. Jackson, D. *Automatic Analysis of Object Models.* in *Software Quality Week*. 1997. San Fransisco, CA.

30. Burch, J., *et al.*, *Symbolic Model Checking for Sequential Circuit Verification.* IEEE Transactions on Computer-Aided Design, 1994. **13**(4).

31. Purtilo, J. *Polylith: An Environment to Support management of Tool Interfaces*. in *ACM SIGPLAN Symposium on Language Issues in Programming Environments*. 1985. Seattle, WA.

32. Williams, B. and P.P. Nayak, *Immobile Robots: AI in the New Millennium.* AI Magazine, 1996. **17**(3): p. 17-35.

33. *JavaStar Fact Sheet*, . 1997, Sun Microsystems: http://www.suntest.com/JavaStar/JavaStar.html.

34. Callahan, J. and T. Montgomery. *Approaches to Verification and Validation of a Reliable Multicast Protocol*. in *Proceedings of the International Symposium on Software Testing and Analysis.* 1996. San Diego, Ca: Association for Computing Machinery.