Specification-based Test Oracles for Reactive Systems

Debra J. Richardson
Stephanie Leif Aha
T. Owen O'Malley

Information and Computer Science
University of California, Irvine
Irvine, CA 92717 USA

Abstract

The testing process is typically systematic in test data selection and test execution. For the most part, however, the effective use of test oracles has been neglected, even though they are a critical component of the testing process. Test oracles prescribe acceptable behavior for test execution. In the absence of judging test results with oracles, testing does not achieve its goal of revealing failures or assuring correct behavior in a practical manner; manual result checking is neither reliable nor cost-effective.

We argue that test oracles should be derived from specifications and in conjunction with testing criteria, represented in a common form, and their use made integral to the testing process. For complex, reactive systems, oracles must reflect the multiparadigm nature of the required behavior. Such systems are often specified using multiple languages, each selected for its utility in specifying a particular computational paradigm. Thus, we are developing an approach for deriving and using oracles based on multiparadigm and multilingual specifications to enable the verification of test results for reactive systems as well as less complex systems.

1 Introduction

Testing is a costly, human-intensive process that is critical for assessing system behavior. Unfortunately, it often yields very little trustworthy information about the correctness of a software system. One reason for this is the lack of an effective means for determining whether the system has behaved correctly on test executions. Selecting test data to comprehensively "cover" the system is a definitive activity. The system is executed for this test data, but usually the outcome is left for someone to visually examine and guess whether the system has behaved correctly. The real knowledge about system behavior may be lost if this is done carelessly or incompletely. Unfortunately, result verification typically is inadequate due to its costly and complex nature; thus, the goal of testing — to reveal system failure or provide assurance in system correctness — is not achieved. Despite the expense, the testing effort has been for naught.

This problem can be tackled by incorporating oracles into the testing process. A *test oracle* determines whether a system behaves correctly during test execution. Typically, in testing sequential systems, determining that the system behaves correctly must only consider functionality. For *reactive systems*, which are characterized as largely event-driven and continuously reacting to external stimuli and internal events [Har87], many behavioral aspects must be considered, including not only functional but also timing and safety requirements. Multiple computational paradigms must often be used for different aspects to completely specify such systems [Zav89]. To determine behavioral correctness, oracles for such systems must judge all behavioral aspects in comparison with all system specifications and requirements.

We are developing an approach to deriving test oracles from specifications and incorporating their effective use into the testing process. Our approach is workable for a wide variety of specification formalisms and hence is usable for a variety of application domains, behavioral aspects, and computational paradigms. Specifications of complex and reactive systems often represent multiple computational paradigms and may be encoded in multiple languages. Thus we designed our approach to be applicable to multiparadigm and multilingual specifications, which we refer to here as *multi-specifications*. We derive test oracles from multi-specifications and compose these multiple oracles to check test results for behavioral correctness. In addition, we couple oracle derivation with testing and represent an oracle for each test class specified by the testing criteria in use. We are also examining the associated test monitoring needs. To enable comparison of test results to specification-based oracles, our approach hinges on a mapping between the specification's abstract values and the implementation's concrete values.

Test oracles should be an integral part of the testing process. Testers have systematic means of doing test data selection and test execution available; the testing process also includes oracle creation, test execution monitoring, and test

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant # MDA972-91-J-1012 and the University of California MICRO program and Hughes Aircraft Company under grant # 91-131. The content of the information does not necessarily reflect the position or the policy of the U.S. Government, the University of California or Hughes Aircraft Company, and no official endorsement should be inferred.

To appear in the 14th International Conference on Software Engineering, Melborne, Australia, May 1992.

result verification, although these are usually done informally and inaccurately. Our approach to deriving and using specification-based oracles provides a framework for formalizing these activities as well, thereby verifying test execution against the specification. Moreover, testing is an especially valuable means of validating the system against its requirements, which may or may not be reflected in the specification.

This paper describes our approach to deriving and using specification-based oracles in the testing process. We begin by defining our terminology related to testing and test oracles. In Section 3, we describe our approach to deriving and using oracles: we first describe how to derive oracles from specifications; we then discuss requirements for developing a monitor; and finally, we take up using oracles for recognizing [in]correct behavior. Section 4 presents a multi-specification and implementation of an example reactive system, which we use to illustrate our approach and the concepts presented. In Section 5, we discuss some related work, although little research has focused specifically on oracles. Finally, we discuss the relation of oracles to the testing process and conclude with our future plans.

2 Terminology

The basic testing process consists of several tasks, although not all testing activities perform each task. Test classes are generated to meet one or more testing criteria. The system [component] is executed for data satisfying the described test classes. System behavior may then undergo *verification* with respect to a behavioral specification, thereby checking the outputs and state changes for *behavioral correctness*. This process also facilitates *validation* of system behavior with respect to intended behavior[1]. Complex systems, such as reactive systems, have many aspects of behavior, including functionality, timing, safety, security and performance. The testing process should verify results against requirements for all such behaviors.

A *testing criterion* is a set of requirements on test data that reflects a notion of adequately testing a system [component] [Wey86]. A basic goal of most testing criteria is partitioning the system's test data domain into equivalence classes, such that a test of a representative of each class serves to test the entire class [GG75, WO80, RC85a]; in general, this goal is unattainable [GG75, How76]. A *test class* describes a subset of test data by an input condition. The input condition constrains input data and the initial system state. An input condition may be as specific as actual data or more general, like an expression describing the domain of data to test some required function. For a reactive system, the test data for a test execution consists of a sequence of stimuli and a test class describes a set of related sequences.

A *test oracle* is a mechanism for determining behavioral correctness of test executions. A test oracle has two components: the *oracle information* specifies what constitutes correct behavior, while the *oracle procedure* verifies the test execution results with respect to the corresponding oracle information. A typical, but inadequate, oracle procedure is "no run-time failures," which requires no oracle information. This oracle is certainly capable of detecting failures, but the absence of a run-time failure does little to assure that the implementation satisfies its specification. In general, oracle information provides a partial specification, because it may specify a property that must hold for specific test data or may specify only part of the execution results. The simplest, useful oracle information, input/output pairs, represents correct output values for particular input values. The oracle procedure for an input/output pair is simply "=", which compares the actual output to the oracle-specified output for a given input. Embedded assertions [Stu73, LvH85] are another form of oracle information; the oracle procedure compares intermediate (internal state values) to the oracle information and not just final results of program execution. Reactive systems require some form of intermediate oracle information, because they are not tested for complete executions but rather for stimulus/response pairs.

Execution information must be captured from the runtime environment by a *test monitor*. Testing simple systems usually does not require a sophisticated monitor as all execution information can be gleaned from system output. Test monitors for reactive systems, by the very nature of their behavior, must capture additional information from the runtime environment, such as internal events, timing information, stimuli, and responses . The *execution profile* is all the information captured by the monitor, including control and data information.

3 Deriving and Using Specification-Based Oracles

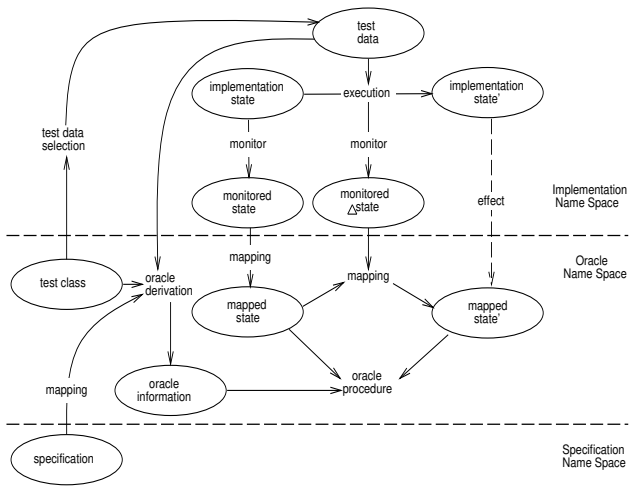


Figure 1: Approach for Specification-Based Oracles

Here, we describe our approach for verifying test results for behavioral correctness, which is based on multi-specifications for reactive systems. We derive oracles from specifications in conjunction with testing so that every test class has an oracle represented for it explicitly. Test execution is monitored and results are verified against the oracles corresponding to all test classes that are satisfied for the test

[1] Validation requires comparison of behavior with intended requirements, not necessarily the same as those specified.

data. To enable this comparison, the test results must be mapped to the oracle name space, which is derived from the multi-specification. was specifications Most of our approach is applied to the different specifications of a complex system independently. The monitor must be able to capture execution information needed for all specifications, and the oracles derived from all specifications are composed when test execution results are verified.

This section describes the three phases of our approach: deriving oracles from specifications for each test class, which is based on mapping from implementation to specification; monitoring test execution, which requires determining the monitor capabilities required for all oracles of each specification; and mapping and applying the oracle procedures to the execution profile. Figure 1 illustrates our approach. We will refer to and explain this figure further throughout this section.

Using our approach introduces inaccuracy when it is not used to its full power. We discuss the potentials for optimistic and pessimistic inaccuracy [YT89] below. In addition, we present the tradeoffs involved in each phase of our approach.

3.1 Oracle Derivation

To use oracles derived from specifications the name space of the test data (the implementation name space) must be mapped to the name space of the oracle information (the specification name space). The specification is then symbolically interpreted to represent the oracle information as assertions on the data that must hold at mapped control points. Additionally, this phase is integrated with testing criteria so that oracles are derived for generated test classes.

Developing the Mapping

To analyze a specification and an implementation, there must be a mapping between the concrete name space of the implementation and the abstract name space of the specification [GHM87]. Since this information is typically not found in the specification, nor can it be automatically derived, it must be developed by the tester, designer or implementor[2].

The oracle may actually be in a third name space, as illustrated in Figure 1, although it can be the same as either the specification or the implementation name space[3]. Our approach derives oracles whose name space is the same as the specification name space and maps the execution profile in the implementation space to the specification space. Thus, the mapping is applied to the test results. We chose this technique because this mapping usually folds state spaces. It is possible to map in the other direction, in which case the mapping would be applied in deriving the oracle information and the oracle name space would be that of the implementation. In the most general case, the mapping is cut by the oracle, requiring mapping from the specification space to the oracle space before the oracle is derived and mapping from the implementation space to the oracle space during or after test execution. Our future research will evaluate the tradeoffs between these techniques.

There are two basic categories of mappings: control and data. Control mappings are between control points in the implementation and locations in the specification, where the implementation and specification are considered at the same place and should be in the same state. Typically, the transformations are between invocations of procedures, functions or task entries in the implementation and transitions, events, or operations in the specification. Data mappings are transformations between data structures in the implementation and objects in the specification. These mappings are necessary because the implementation and specification often have different data types and operations, and hence data values are represented differently. As illustrated in Figure 1, our approach requires two related data mappings; one maps an entire monitored implementation state to the oracle name space and the other only maps the change in monitored implementation state. The former is applied to the initial state of the test execution, while the latter is applied at subsequent mapped control points. Note that the composite mapping from implementation state through monitored Δ state to mapped state$'$ has the effect of mapping from implementation state through implementation state$'$ to mapped state$'$. Additionally, that composite mapping must be equivalent to mapping from implementation state through monitored state and mapped state to mapped state$'$, which is determined by applying the oracle procedure[4].

The simplest maps are just transformations that convert implementation identifiers into specification identifiers, possibly altering the syntax slightly such as changing parameter order. Other mappings require more work to develop and use. For example, algebraic specifications require a conversion between abstract data values, which are sequences of operations applied to the constructors, and concrete values, which are program variables. One solution is to require the programmer to implement query functions to assist automation in converting the concrete state of the system into an abstract state [ROT89].

This phase of our approach introduces optimistic inaccuracy when insufficient information is mapped; the adequacy of the mapping determines the adequacy of the oracles and the adequacy of the monitor. For instance, there is a very important tradeoff between the data mapping complexity and failure detection. If the entire state is mapped in a detailed manner, then at any control point, the oracle procedure can catch potential failures in the state. If only external data are mapped, however, then failures can only be detected in external responses and must transfer appropriately [RT88]. Likewise, if the control mapping is inadequate and important control points are ignored, the state will not be mapped often enough to enable accurate failure detection. Thus, developing the mapping may require iterative refinement that adds or deletes control and/or data mappings as the oracle is determined [in]effective or the monitor is found too degrading.

Wing [Win90] classifies specification languages as property-oriented or model-oriented. Property-oriented languages define system behavior by stating properties that the

[2] Automatic mappings are possible if the structure of the implementation closely matches that of the specification or if the implementation was automatically derived from the specification.

[3] The advantage of a shared name space is that only one mapping is required.

[4] This equivalence corresponds to correctness implied by the commuting data-abstraction diagram in [GHM87].

system must satisfy. Model-oriented languages define system behavior by constructing models in terms of mathematical structures. Generally, property-oriented languages concentrate either on data or event ordering properties. Data property-oriented languages, such as assertion languages (e.g., Anna [LvH85]), axiomatic languages (e.g., Larch interface language [Win83, GHW85]), and algebraic languages (e.g., OBJ [GT79] and Larch shared language) only require data mappings, because the control mappings are trivial or included as part of the specification. For instance, a Larch specification includes a control mapping in the interface language between the shared language specification and the implementation. Event property-oriented languages, such as temporal logic [Pnu86] and interval logic [RG89], have control mappings that relate Ada invocations to events. Alternatively, model-oriented languages, such as Z [Spi89] and Statecharts [Har87], have both data and control mappings. Either data or control mappings, respectively, may be the identity function when the data or control structure of the specification happens to be identical to the implementation[5]. In Z specifications, the data mappings relate implementation variables to variables in the schema signatures, while control mappings relate Ada invocations to Z predicates. In state charts, data mappings relate implementation variables to states and state variables while control mappings relate Ada invocations to state transitions.

Deriving the Oracle Information and Procedure

Some specifications, such as those written in assertion languages, can serve as test oracles in their own right. For most, however, the specification cannot be evaluated directly for a specific test datum. It is necessary, therefore, to derive oracle information from the specification as well as an oracle procedure that evaluates the test results.

Determining behavioral correctness of complex, reactive systems requires oracles to capture multiple aspects of systems, including functionality, timing and safety. These multiparadigm aspects carry over to oracles; the different behavioral aspects of complex oracles are best described using different paradigms, and hence different formalisms, for oracle information. The choice of formalism used for oracle information naturally depends on the specification language. Following this multiparadigm approach, for instance, the timing characteristics for a reactive system might be specified in a temporal logic specification and the oracle information represented similarly, while the expected functional results are represented in another paradigm/formalism.

To enable a uniform approach for all aspects of a multi-specification, we standardize our treatment of specification languages and corresponding oracle formalisms by coercing all oracle information into assertions that must hold at specific control points. An "assertion" is a logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution [IEE83]. Deriving this "standard" representation for all oracle information derived from a multi-specification facilitates checking test results against all oracles.

[5] This may occur as a result of a design process that closely follows the specification.

Our basic technique for deriving oracle information from model-oriented specifications is symbolic interpretation of the specification [Kem85, RC85a, RC85b]. Symbolic names are assigned for the stimulus inputs and initial state variables, and the specification is symbolically interpreted. Effectively, this means that all "control paths" in the specification must be interpreted and represented as a compound assertion[6]; we will address partitioning this symbolic representation in conjunction with testing below. For property-oriented specifications, we use a simpler form of symbolic interpretation to derive oracle information. Symbollic names for stimulus and initial state are treated as free variables. Starting from each stimulus, referenced properties are expanded and resolution is done where possible. Likewise, the representation is partitioned in collaboration with testing criteria.

The results of symbolic interpretation are coerced into a standard oracle representation that consists of a partially ordered sequence of specification control points (parameterized events, transitions, operations, etc.) and corresponding data assertions that must hold when the control point is reached. The control points are determined by the control mapping described above (alternatively, the control mapping points may be dictated by where oracle information is needed). The simplest instantiation represents an input/output pair, where the control point is where output is produced and the assertion describes the relationship between the symbolic inputs and outputs. Reactive systems require oracle information at intermediate control points, because they must continuously respond to external and internal stimuli and are not tested for complete executions. Thus, a single oracle representation may specify stimulus / response pairs for one or more execution cycles, where control points may be parameterized and assertions may refer to the target system's state or state change.

Oracle procedures analyze test results against the oracle information to verify behavioral correctness. The oracle procedure binds the symbolic names(free variables) to the monitored/mapped test data and initial state values. The bound oracle representation and the monitored/mapped state are checked for consistency at each mapped control point.

Representing Oracles for Test Classes

As our purpose is to provide test oracles, we couple oracle derivation with test class generation. Test classes may be selected manually, but typically some testing criterion is used, which generates test classes that "cover" some portion of the system [component]. The oracle for a test class must specify acceptable behavior for all test data that satisfy the test class.

Testing criteria are either implementation-based [How75], typically called structural or white box testing, or specification-based [ROT89], often termed functional or black box testing. The choice of testing criteria impacts the most natural choice for the oracle name space. As discussed in the previous subsection, the mapping between implementation and specification may be cut by the oracle. Thus, the mapping may be done from the specification to the or-

[6] Depending on the formalism for the oracle information, this may require classifying iteration and/or recursion and developing a global symbolic representation (see [RC85b]).

acle or from the implementation to the oracle or in some combination. The oracle name space should be the same as the name space of the test classes so that oracle information can be represented for each test class. Thus, when deriving oracles in conjunction with specification-based testing, it is more straightforward to map the execution profile to the oracle(specification) name space, while in conjunction with implementation-based testing, it is more natural for the mapping to be done from the specification to the oracle(implementation) name space. Another consideration to be kept in mind is that derivation of specification-based oracles is more straightforward when collaborating with specification-based testing, because such test classes partition the test data domain in a way more clearly reflected by the structure of the specification. In coupling oracle derivation with implementation-based testing, the test classes may be orthogonal to the specification's structure. In this case, multiple oracle representations or multiple output assertions in a single oracle representation may be derived for one test class.[7]

Our "standard" oracle representation allows a single, universal oracle for all possible test data; it would consist of an assertion representing the complete symbolic representation of the specification. Partitioning the oracle information based on test classes, however, facilitates reuse and simplifies deriving oracle information and verifying test results with the oracle procedure.

Deriving the oracle representation for a single test datum is straight-forward; at the output events, assertions specify correct output data for the test datum by equations specifying input and output. A similar representation applies to an actual sequence of stimulus/response pairs for a reactive system. Deriving oracle information that holds for a test class is more difficult; the assertions in the oracle information must hold for all members of that class. The oracle representation for a test class is derived by restricting the symbolic interpretation or resolution by the test class description; this guides the symbolic interpretation to relevant portions in a model-oriented specification or guides the expansion of relevant properties in a property-oriented specification.

Deriving oracle information from specifications allows the information to be maintained in a reusable format. Thus, much of the cost of evaluating the specification is paid once for a test class and amortized over multiple test data in the class and over regression testing. Moreover, when coupled with testing criteria, much of the work required to develop the oracle information (such as symbolic interpretation) may have already been done in developing the test class description and can be reused as well. Another option to prefatory symbolic interpretation of the specification for a test class is to simply use a "subset" of the specification as the oracle information and to evaluation that "sub-specification" for the test data. The primary tradeoff between these two approaches is the cost of symbolic interpretation shared across multiple test data in a test class verses the cost of repeated evaluation of the specification for all test data. Moreover, there are tradeoffs between either approach as opposed to simply evaluating the full specification for all test data and verifying its consistency with the mapped execution profile[8]. Both approaches may introduce optimistic inaccuracy if a complete oracle representation is not derived. Some of the failure detection capability of the specification may be lost due to selecting parts of the specification. Future research will study these various tradeoffs.

[7] Our work in specification-based testing [ROT89] also, therefore, led us to the decision to represent the oracle information in the specification name space.

3.2 Monitor Development

Oracle procedures analyze test results against the oracle information to verify behavioral correctness. For all but the simplest of systems, more execution information than just the output is needed by the oracle procedure. For reactive systems, in particular, timing, internal state and events, stimuli and responses are typically required to determine behavioral correctness. Test monitors capture such information in an execution profile.

The use of multi-specifications and oracles derived from them may require a wide range of monitor requirements. Specifications that describe different aspects of a complex system may, in all likelihood, require differing monitor capabilities. The requirements for a single monitor for such a system are determined by composing the needs of the oracles derived from each specification. These needs are reflected in the mappings developed for each specification (which is tied to the derived oracle information) and thus consist of the control points to be mapped and the state data values to be mapped, both in the initial state and in the state change at control points.

Our approach has to balance the degradation caused by monitoring with the cost of applying the mapping and oracle procedure. The problems of monitoring reactive systems are shared between testing and debugging. McDowell and Helmbold [MH89] identify the fundamental problems as the probe effect, lack of a synchronized global clock, and nonrepeatability. The probe effect governs what the execution profile may effectively contain, since probes disturb the execution. The lack of a synchronized, global clock makes it difficult to determine the precise event ordering with multiple processors. Nonrepeatability is the phenomenon of multiple executions on the same test data that yield different results due to timing differences, some of which may be correct and some incorrect. Our efforts to capture execution profiles must address these problems. To address the probe effect, our approach monitors the implementation state change (after the initial state is captured) at specific control points as determined by the control mapping, as seen in Figure 1, rather than constantly probing the entire state. We approximate a global clock by time-stamping control points and using the partial order of these events in verifying behavioral correctness. We compensate for nonrepeatability by not requiring oracle representations to be completely ordered. There is a compromise between the level of abstraction of the execution profile and the solutions to these problems. Moreover, there is a compromise between the level of abstraction of the execution profile and the complexity of the mapping of the execution profile. Mappings that analyze more abstract information are easier to write but require some extra effort into designing and implementing monitors.

[8] This is another means of showing correctness corresponding to the commuting data-abstraction diagram in [GHM87].

This phase of our approach introduces optimistic inaccuracy if not enough information is monitored. But developing the monitor is determined by the needs of the oracles, so this inaccuracy actually falls onto the inaccuracy in the oracle information.

3.3 Oracle Use

Using specification-based oracles to assess system behavior of reactive systems incurs verifying consistency between the mapped execution profile and the multiple oracles derived from a multi-specification. Thus, the different mappings, oracle information, and oracle procedures must all be used together to evaluate the system's behavioral correctness.

Mapping Execution Profile

For each specification, the mappings developed (as described earlier) are applied to the execution profile, thereby transforming the monitored implementation state and state change to the corresponding states in the specification (oracle) name space. Since a single monitor meets the needs of all derived oracle information, there may be monitored control points and/or data that do not have a corresponding mapping for any particular specification. Any monitored results that are not in the mapping domain may be ignored.

Verifying Consistency

To verify consistency, each test execution is checked against all applicable oracles. As discussed, a test datum or sequence of stimuli for reactive systems may satisfy one or more test classes and these test classes may be associated with different specifications. To determine which test classes are satisfied, the symbolic names (or free variables) in the test class description are bound to the test data values. If this substitution is consistent, then the test class is satisfied and the associated oracle must be applied. Each test class has oracle information represented for it and, in general, each specification has a different oracle procedure.

The oracle procedure also applies a substitution to the symbolic names in the oracle representation. This is done at each control point; the symbolic names (and free variables) in both the mapped control event and the corresponding assertion are bound to test data values and state information from the execution profile. The assertions are then checked for consistency. Any unsatisfied assertion identifies an inconsistency between execution behavior and the specification-based oracle.

Thus, it is in verifying consistency that we compose multiple oracles for multiparadigm or multilingual specifications of complex, reactive systems. The conjunction of verifying test results against all applicable oracles serves as the combined oracle, and in this way the different paradigms are joined together to form a composite specification of the correct system behavior.

This phase introduces pessimistic inaccuracy when events in the mapped execution profile do not appear in the oracle representation. The obvious interpretation of this is that an inconsistency and possible failure has been revealed. In less complete specifications, however, such as a temporal logic specification that primarily considers ordering aspects of the system behavior, this appears to be an inappropriate explanation, although it does highlight a difference between the execution profile and the specification that should be analyzed further. The meaning of extraneous mapped control points is an active topic of our research.

4 Specifications, Implementation & Testing of an Elevator System

This section illustrates multi-specifications and our approach to deriving oracles from specifications and using them for verification of the behavioral correctness of test results for an elevator system. The elevator must react to external stimuli: button presses corresponding to calls for elevators and requests to service specific floors. Herein, we present specifications of the elevator in two different languages, corresponding to two computational paradigms. The Real-Time Interval Logic (RTIL) specification describes the temporal requirements of the elevator, in particular that all calls and requests are serviced. The Z specification describes the scheduling of services. Each specification, therefore, describes different properties that must be satisfied for the implementation to behave correctly.

For each specification, we first discuss the mapping necessary from the implementation name space, which corresponds to the execution profile, and the specification name space, which corresponds to the oracle. Next, we discuss the derivation of the oracle information and oracle procedure. We then provide a test class description and develop the oracle for that test class. The second subsection discusses the monitor capabilities required for the multi-specification and describes the execution profile for one test data sequence. Finally, for each specification we apply the mapping and oracle procedure to verify behavioral correctness for this test data.

In Appendix A, we provide the Ada package specification for an elevator system simulation. For brevity, we do not provide the package body, although the full implementation is provided in [RAO92]. We tested the elevator on a large test data set and found several inconsistencies between the implementation and the multi-specification. We have corrected most of these but have left some so as to illustrate the concepts underlying our approach. The one test data sequence presented here highlights inconsistencies between the implementation and interesting aspects of each specification (thus, please note that these are inconsistencies that we actually found through the process described herein). This single test data sequence satisfies the test class description used for each specification, although these vary by specification. The execution profile is described in Appendix B.

4.1 RTIL Specification & Oracle

Real Time Interval Logic (RTIL) specifies the event orders that are legal and those that are failures [RG89].

4.1.1 Specification

The RTIL specification of the elevator specifies the general liveness properties of the elevator, such as every call and request will be answered eventually. Note that the specification does not enforce a particular scheduling order.

Specification Events

The behavior of the elevators are specified in terms of the following events:

set_dir(elev,dir): The elevator's direction changes to (up,down,idle);

open(elev): The elevator's doors open;

close(elev): The elevator's doors close;

at(elev,flr): The elevator reached a new floor;

call(floor,direction): A call button was pressed on the given floor in the given direction;

request(elev,floor): The inside elevator button was pushed for the given floor.

Temporal Properties

The elevator shall not move with the doors open, thus the doors shall not open between starting and stopping the elevator.

$$\Box[\odot set_dir(elev, dir) \Rightarrow \odot set_dir(elev, idle)] \\ (dir = up \vee dir = down) \rightarrow\!\sim \odot open(elev)$$

The doors shall remain open at least 15 seconds, thus there shall be at least 15 seconds between opening and closing the doors.

$$\Box[\odot open(elev) \Rightarrow \odot close(elev)](duration \geq 15sec)$$

All requests shall be serviced eventually, thus after each request there shall be an interval when the elevator reaches the given floor and opens its doors without moving to another floor.

$$\Box[\odot request(elev, flr) \Rightarrow]\Diamond[\odot at(elev, flr) \Rightarrow \odot open(elev)] \\ flr2 \neq flr \rightarrow\!\sim \odot at(elev, flr2)$$

$$\Box[\odot call(flr, dir) \Rightarrow]\Diamond[\odot at(elev, flr) \Rightarrow \odot open(elev)] \\ flr2 \neq flr \rightarrow\!\sim \odot at(elev, flr2)$$

4.1.2 Oracle Derivation

Mapping

RTIL is event property-oriented, thereby requiring a control mapping. The control mapping, which is between implementation invocations and specification events for the elevator specification, is given in Table 1.

Table 1: Control Mapping for RTIL

Implementation	⇒	Specification
Elevator Task		
Set_Dir(Elev,Dir)	⇒	set_dir(Elev,Dir)
Open_Door(Elev)	⇒	open(Elev)
Close_Door(Elev)	⇒	close(Elev)
Controller Task		
At_Floor(Elev,Floor)	⇒	at(Elev,Floor)
Request(Elev,Floor)	⇒	request(Elev,Floor)
Call(Floor,Dir)	⇒	call(Floor,Dir)

Assertions in the temporal logic are over the implementation variables, so the data mapping is the identity function.

Oracle Information and Procedure

To generate the oracle information, the relevant properties of the specification are expanded and resolved to generate a set of expected time intervals and the appropriate assertions for those intervals. The relevant properties of the specification are determined by first considering each property referring to a stimulus event. Additional properties are considered relevant if their intervals are based on events from the bodies of a property being expanded. Note that the intervals in the oracle information correspond to the transitions for model-oriented oracle information.

The oracle procedure determines whether the current event completes any of the uncompleted intervals in the oracle representation and if so, checks the appropriate assertions. A failure is revealed whenever an assertion associated with an interval is not satisfied or an interval has not been completed by the end of the execution profile. If an event does form part of any of the intervals in the oracle information, then it may represent an inconsistency and a potential failure and must be checked further.

Test Class and Oracle

One test class for the RTIL specification is two calls and two requests; we verify that these are serviced. This test class is described:

Initial State
Don't care
Stimulus
call(Flr_1,Dir_1)
call(Flr_2,Dir_2)
request(Elev_1,Flr_3)
request(Elev_2,Flr_4)
Stimulus Constraints
None

To generate the oracle for this test class, symbolic interpretation derives the following set of intervals and assertions that must hold within them. For example, the $\mathsf{call}(\mathsf{Flr}_1,\mathsf{Dir}_1)$ generates the first two intervals, the first of which ensures that the call is answered and the second which ensures that the doors remain open at least 15 seconds. Note that the variables Elev_3 and Elev_4 were introduced to represent the elevators that respond to the calls.

Table 2: RTIL Oracle for Test Class

Intervals	Assertions
$\Box[\odot call(\mathrm{Flr}_1, \mathrm{Dir}_1) \Rightarrow]$ $\Diamond[\odot at(\mathrm{Elev}_3, \mathrm{Flr}_1) \Rightarrow \odot open(\mathrm{Elev}_3)]$	$\mathrm{Flr}_i \neq \mathrm{Flr}_1 \rightarrow$ $\sim \odot at(\mathrm{Elev}_3, \mathrm{Flr}_i)$
$\Box[\odot open(\mathrm{Elev}_3) \Rightarrow \odot close(\mathrm{Elev}_3)]$	$(duration \geq 15sec)$
$\Box[\odot call(\mathrm{Flr}_2, \mathrm{Dir}_2) \Rightarrow]$ $\Diamond[\odot at(\mathrm{Elev}_4, \mathrm{Flr}_2) \Rightarrow \odot open(\mathrm{Elev}_4)]$	$\mathrm{Flr}_i \neq \mathrm{Flr}_2 \rightarrow$ $\sim \odot at(\mathrm{Elev}_4, \mathrm{Flr}_i)$
$\Box[\odot open(\mathrm{Elev}_4) \Rightarrow \odot close(\mathrm{Elev}_4)]$	$(duration \geq 15sec)$
$\Box[\odot request(\mathrm{Elev}_1, \mathrm{Flr}_3) \Rightarrow]$ $\Diamond[\odot at(\mathrm{Elev}_1, \mathrm{Flr}_3) \Rightarrow \odot open(\mathrm{Elev}_1)]$	$\mathrm{Flr}_i \neq \mathrm{Flr}_3 \rightarrow$ $\sim \odot at(\mathrm{Elev}_1, flr2)$
$\Box[\odot open(\mathrm{Elev}_1) \Rightarrow \odot close(\mathrm{Elev}_1)]$	$(duration \geq 15sec)$
$\Box[\odot request(\mathrm{Elev}_2, \mathrm{Flr}_4) \Rightarrow]$ $\Diamond[\odot at(\mathrm{Elev}_2, \mathrm{Flr}_4) \Rightarrow \odot open(\mathrm{Elev}_2)]$	$\mathrm{Flr}_i \neq \mathrm{Flr}_4 \rightarrow$ $\sim \odot at(\mathrm{Elev}_2, flr4)$
$\Box[\odot open(\mathrm{Elev}_2) \Rightarrow \odot close(\mathrm{Elev}_2)]$	$(duration \geq 15sec)$

4.2 Z Specification & Oracle

A Z specification models the target system with a set of schemas, whose signatures specify state variables and whose predicates specify transitions between states [Spi89].

4.2.1 Specification

Our Z specification of the elevator focuses on the scheduling of the elevators and assigning calls to elevators. In particular, it does not specify the actions of the elevator doors or stopping at floors to let people enter or exit the elevator. Our specification is organized into four parts: general definitions, allocating calls to elevators, external events, and moving the elevator.

Note: Variables not mentioned in the schema are unchanged. We realize that this is not standard Z but have done so to shorten the presentation. In addition, we have adopted Ada's commenting style, which is also non-standard.

Initial Definitions

Direction == {up, idle, down}
Floor-Type == 1..Num-Of-Floors
Elevator-Type == 1..Num-Of-Elevators

Elevator[Elev]

Current-Floor: Floor-Type	--The current floor
Current-Dir: Direction	--The current direction
Requests: $\mathbb{P}$ Floor-Type	--The requests for this elevator
Calls: $\mathbb{P}$(Direction × Floor-Type)	--The calls for this elevator
Reverse: Direction → Direction	--Returns the reverse directions

Reverse = { (up ↦ down), (idle ↦ idle), (down ↦ up) }

The initial state for all of the elevators is on the first floor with no calls or requests and therefore idle.

Init-Elevator[Elev]

ΔElevator[Elev]

Current-Floor=1
Current-Dir=idle
Requests=∅
Calls= ∅

Choosing an elevator

As passengers call elevators to service a floor, the scheduler allocates the calls to particular elevators. To minimize waiting, the scheduler chooses the "best" elevator to service a call by computing a weighted sum for each elevator of it's projected path length and number of stops if it were to service this call. Once a call is assigned to an elevator, it is never reassigned to a different one.

To compute the length of the projected path, we find the floors where the elevator would turn. Each elevator services all of its goals ahead in the current direction then turns around. At any time, there are three segments of the path: the segment ahead of the current floor, the segment going the reverse direction and the segment behind the current floor. If the projected path does not have one of the segments, the value 0 represents the floor where the elevator turns. Knowing where the elevator will turn, we can find the length of each segment and add them together.

Ahead(Dir, Flr1, Flr2) is true if Flr2 is ahead of an elevator at Flr1 going the given direction. Furthest(Dir, Flr1, Flr2) returns whichever floor is furthest in the given direction. Therefore, Ahead is similar to less than for floors and Furthest is similar to max.

Floor-Or-Null == (Floor-Type ∪ {0})

Floor-Functions

Ahead: (Direction × Floor-Type × Floor-Type) → boolean
Furthest: (Direction × Floor-Or-Null × Floor-Type) → Floor-Type

∀ Dir: Direction ; From: Floor-Type ; To: Floor-Type •
  Ahead(Dir,From,To) = ((Dir = up ∧ From > To) ∨ (Dir = down ∧ From < To))
∀ Dir: Direction ; Flr1: Floor-Type ; Flr2: Floor-Type •
  (Flr1 = 0 ⇒ ((Dir,Flr1,Flr2) ↦ Flr2) ∈ Furthest) ∨
  (Flr1 ≠ 0 ⇒
    (Ahead(Dir,Flr1,Flr2) ⇒
      (Dir,Flr1,Flr2) ↦ Flr2) ∈ Furthest) ∨
    (¬ Ahead(Dir,Flr1,Flr2) ⇒
      ((Dir,Flr1,Flr2) ↦ Flr1) ∈ Furthest))

Path-Info

First-Turn : Floor-Type	--Turn after first segment
Second-Turn : Floor-Or-Null	--Turn after second segment
Third-Turn : Floor-Or-Null	--Turn after third segment
Stops: $\mathbb{N}$	--Number of stops in the path

Second-Turn = 0 ⇒ Third-Turn = 0

Update-Path-For-Call and Update-Path-For-Request update the turning points for calls and requests, respectively. Both schemas determine which section of the elevator's path the goal is in and update the appropriate turning points.

Update-Path-For-Call[Elev]

ΔPath-Info
ΞFloor-Functions
ΞElevator[Elev]

Floor? : Floor-Type	--The call's floor
Dir? : Direction	--The call's direction

Stops′ = Stops + 1
(Ahead(Current-Dir,Current-Floor,Floor?) ⇒
  First-Turn′ = Furthest(Current-Dir,First-Turn,Floor?) ∧
  (Dir? ≠ Current-Dir ⇒
    Second-Turn′ =
      Furthest(reverse(Current-Dir),Second-Turn,Floor?)))
(¬ Ahead(Current-Dir,Current-Floor,Floor?) ⇒
  (Second-Turn′ =
    Furthest(reverse(Current-Dir),Second-Turn,Floor?)) ∧
  (Dir? = Current-Dir ⇒
    Third-Turn′ = Furthest(Current-Dir,Final-Stop,Floor?)))

Update-Path-For-Request[Elev]

ΔPath-Info
ΞFloor-Functions
ΞElevator[Elev]

Floor? : Floor-Type	--The request's floor

Stops′ = Stops + 1
(Ahead(Current-Dir,Current-Floor,Floor?) ⇒
  First-Turn′ = Furthest(Current-Dir,First-Turn,Floor))
(¬ Ahead(Current-Dir,Current-Floor,Floor?) ⇒
  Second-Turn′ =
    Furthest(reverse(Current-Dir),Second-Turn,Floor))

Path-Distance computes the total distance function for an elevator over the projected path. The Path-Info state is updated with all of the calls and requests already assigned to this elevator as well as the potential call. Elevators are scheduled based on a "badness" value, which is a weighted sum of the number of stops and path lengths.

Distance-Badness == 1	--Weighting factor for distance
Stop-Badness == 3	--Weighting factor for stops

Path-Distance[Elev]

ΔPath-Info
ΞElevator[Elev]
ΞUpdate-Path-For-Call[Elev]
ΞUpdate-Path-For-Request[Elev]

Floor? : Floor-Type	--The new call's floor
Dir? : Direction	--The new call's direction
Badness!: $\mathbb{N}_1$	--Badness for the predicted path
Second-Segment-Len: $\mathbb{N}$	--Length of the second segment
Third-Segment-Len: $\mathbb{N}$	--Length of the third segment

First-Turn = Current-Floor; Second-Turn = 0; Third-Stop = 0;
Stops = 0;
∀ (Flr,Dir) ∈ Calls • Update-Path-For-Call(Flr,Dir);
Update-Path-For-Call(Floor?,Dir?);
∀ Flr ∈ Requests • Update-Path-For-Request(Flr,Dir);
(Second-Turn = 0 ⇒ Second-Segment-Len = 0)
(Second-Turn ≠ 0 ⇒
  Second-Segment-Len = abs(Second-Turn - First-Turn))
(Third-Turn = 0 ⇒ Third-Segment-Len = 0)
(Third-Turn ≠ 0 ⇒
  Third-Segment-Len = abs(Third-Turn - Second-Turn))
Badness! = Distance-Badness * (abs(First-Turn - Current-Floor)
  + Second-Segment-Len + Third-Segment-Len)
  + Stops-Badness * Stops

Best-Elev returns the elevator with the lowest badness value for a particular call.

Best-Elev

ΞPath-Distance[Elev-Type]

Floor? : Floor-Type	--The new call's floor
Dir? : Direction	--The new call's direction
Best! : Elev-Type	--The least bad elevator

Best! = Elev: Elev-Type | ∀ Other-Elev: Elev-Type |
  Path-Distance[Elev] ≤ Path-Distance[Other-Elev]

External-Events

The call and request goals for the elevators are created via two external events generated by button pushes.

Call-Event

Floor?: Floor-Type	--The caller's floor
Dir?: Direction	--The desired direction
ΔElevator[Best-Elev(Floor?,Dir?)]	--The answering elevator

(Dir? ≠ idle)
((Floor? = Num-Of-Floors) ⇒ Dir? = down)
((Floor? = 1) ⇒ Dir? = up)
Calls′=Calls ∪ {(Dir?,Floor?)}

Request-Event[Elev]

ΔElevator[Elev]

Floor?: Floor-Type	--The requester's floor

Floor?≤ Max-Floor
Requests′=Requests ∪ {Floor?}

External-Events$\widehat{=}$
  (∃ Elev: Elev-Type • Request-Event[Elev]) ∨ Call-Event

Moving the Elevator

The general approach is to move the elevator in the same direction until there are no more calls or requests ahead in that direction. Then, the elevator turns around and services all of the calls and requests in the other direction. If there are no more calls or requests, the elevator goes idle.

Move[Elev]

ΔElevator[Elev]

(Current-Dir = up) ⇒ (Current-Floor′ = Current-Floor + 1)
(Current-Dir = idle) ⇒ (Current-Floor′ = Current-Floor)
(Current-Dir = down) ⇒ (Current-Floor′ = Current-Floor - 1)

Service-Floor clears any appropriate requests or calls. It would also open and close the doors if they were specified in Z. Note that Service-Floor does nothing if there are no requests or calls on this floor.

Service-Floor[Elev]

ΔElevator[Elev]

Requests′= Requests\ {Current-Floor′}
Calls′ = Calls\ {(Current-Dir,Current-Floor′)}

Floors returns the set of floors that have called this elevator.

Floors == (λ Calls: $\mathbb{P}$(Direction × Floor-Type) •
  { Floor | (Floor,Dir) ∈ Calls })

Set-Direction sets the current direction based on the elevator state. The possible transitions, in order, are: being idle, starting to move, continuing in the same direction and reversing direction.

Set-Direction[Elev]

ΔElevator[Elev]
ΞFloor-Functions

(Requests ∪ Floors(Calls) = ∅ ⇒ Current-Dir′ = idle)
(Requests ∪ Floors(Calls) ≠ ∅ ∧ Current-Dir = idle ⇒
  ∃ Floor ∈ Request ∪ Floors(Calls) •
    Ahead(Current-Dir′,Current-Floor,Floor))
(Requests ∪ Floors(Calls) ≠ ∅ ∧ Current-Dir ≠ idle ⇒
  { Floor ∈ Request ∪ Floors(Calls) |
    Ahead(Current-Dir,Current-Floor,Floor) • Floor } ≠ ∅ ⇒
      Current-Dir′ = Current-Dir) ∨
  { Floor ∈ Request ∪ Floors(Calls) |
  Ahead(Current-Dir,Current-Floor,Floor) • Floor } = ∅ ⇒
      Current-Dir′ = reverse(Current-Dir))

Elevator moves involve moving to a floor, servicing the floor if necessary, setting the direction, and servicing the floor again. Floors are serviced twice so that if the elevator reverses direction, calls for the elevator going the other direction will be serviced. Note that if the elevator is idle, Move-Elev will not change the state.

Move-Elev[Elev] $\widehat{=}$Move[Elev]; Service-Floor[Elev];
  Set-Direction[Elev]; Service-Floor[Elev]

The schemas are composed to form a complete system by allowing an arbitrary order of external events or elevator moves. Naturally, these may happen concurrently.

Event-or-Move$\widehat{=}$ External-Events ∨
  (∃ Elev: Elev-Type • Move-Elev[Elev])
Functioning-Elevator$\widehat{=}$ Event-or-Move*

4.2.2 Oracle Derivation

Mapping

Z specifications are model-oriented specifications, so the oracle derived requires both control and data mappings. The control mapping, which appears in Table 3A, transforms procedure, function or entry invocations to schemas, which are transitions between states. For the Z elevator, the important points are calls, requests, reaching new floors and closing the doors[9]. The data mapping, which appears in Tables 3B & 3C, transforms implementation data values to specification

[9]Closing the doors is important because the implementation does not remove goals from the state until the doors close.

Table 3A: Z Control Mapping

Implementation	⇒	Specification
Controller Task		
Call(Floor:Floor_ID; Dir: Direction)	⇒	Call-Event(Floor,Dir)
Request(Elev: Elevator_ID; Floor: Floor_ID)	⇒	Request-Event[Elev](Floor)
At_Floor(Elev: Elevator_ID; Floor : Floor_ID)	⇒	Move[Elev]
Elevator Task		
Close_Door(Elev: Elevator_ID)	⇒	Service-Floor[Elev]

Table 3B: Z Data Mapping: Complete State

Implementation	⇒	Specification
for Elev in Elevator_ID		
State[Elev].Floor	⇒	Elevator[Elev].Cur-Floor
State[Elev].Dir	⇒	Elevator[Elev].Cur-Dir
for I in 1..State[Elev].Num_Of_Goals		
State[Elev].Goal(I)	⇒	Floor ∈ Elevator[Elev].Calls
	⇒	(Floor,Dir) ∈ Elevator[Elev].Requests

Table 3C: Z Data Mapping: State Change

Implementation Change	⇒	Specification Change
Δ State[Elev].Floor	⇒	Elevator[Elev].Current-Floor′ = Floor
Δ State[Elev].Dir	⇒	Elevator[Elev].Current-Dir′ = Dir
Insert Goal (Elev,Floor,Dir)	⇒	Elevator[Elev].Calls′ = Elevator[Elev].Calls ∪ { (Floor,Dir) }
	⇒	Elevator[Elev].Requests′ = Elevator[Elev].Requests ∪ { Floor }
Delete Goal (Elev,Floor,Dir)	⇒	Elevator[Elev].Call′ = Elevator[Elev].Call \ { Floor }
	⇒	Elevator[Elev].Requests′ = Elevator[Elev].Requests \ { (Floor,Dir) }

data values. Current floor, direction, and goals for each elevator are mapped[10].

Oracle Information and Procedure

To generate oracle information, the Z specification is symbolically interpreted to find the state where the stimuli have been responded to. This corresponds to restoring the previous state, which occurs when goals are removed from either the Calls or Requests sets and therefore they have been "handled". The oracle information is a list of schemas and assertions that must hold when those schemas apply.

The Z oracle procedure checks that each expected schema is called and checks both the assertions from the oracle information against the specification state.

Test Class and Oracle

One test class for the Z specification is two calls and two requests serviced by a single elevator; we verify that it services them in the correct order. This test class is described:

[10] Mapping the elevator's goals is relatively difficult because the implementation discards whether the goal was originally from a call or a request. Thus, the mapping is from one implementation state to two possible specification states.

Initial State
$Elevator[Elev_1].Current\text{-}Flr = Flr_0 \wedge$ $Elevator[Elev_1].Current\text{-}Dir = idle$
Stimulus
$call(Flr_0,up)$
$request(Elev_1,Flr_1)$
$call(Flr_2,up)$
$request(Elev_1,Flr_3)$
Stimulus Constraints
$Flr_0 < Flr_1 < Flr_2 < Flr_3$
$\forall i : Elev\text{-}Type \mid Path(Elev_1,Flr_2,up) \leq Path(Elev_i,Flr_2,up)$

Symbolic interpretation of the Z specification over the test class derives the oracle represented in Table 4.

4.3 Elevator Monitor and Test Execution

The mappings of each of the Elevator specifications determine the information that must be captured by the monitor. Thus, the multi-specification oracles are composed at this point to develop a single monitor that serves the purposes of all mappings. The monitor captures control points in the execution profile and records the state to be mapped at those points; this includes all data to be mapped of the initial system state and changes to any data to be mapped at later control points.

Control Map Monitoring

For oracles to be applied effectively, they must be able to monitor the program execution in sufficient detail. For

Table 4: Z Oracle for Test Class

Schema	Assertions
Call(Flr_0,up)	$Calls' = \{ Flr_0 \} \wedge Requests' = \emptyset$
Request($Elev_1$,Flr_1)	$Calls' = \{ Flr_0 \} \wedge Requests' = \{ Flr_1 \}$
Service-Floor[$Elev_1$]	$Calls' = \emptyset \wedge Requests' = \{ Flr_1 \} \wedge$ Current-Floor $= Flr_0$
Call(Flr_2,up)	$Calls' = \{ Flr_2 \} \wedge Requests' = \{ Flr_1 \}$
Move[$Elev_1$]	Current-Floor$' \leq Flr_1 \wedge$ Current-Dir $=$ up
Service-Floor[$Elev_1$]	$Calls' = \{ Flr_2 \} \wedge Requests' = \emptyset \wedge$ Current-Floor $= Flr_1$
Move[$Elev_1$]	Current-Floor$' \leq Flr_2 \wedge$ Current-Dir $=$ up
Request($Elev_1$,Flr_3)	$Calls' = \{ Flr_2 \} \wedge Requests' = \{ Flr_3 \}$
Service-Floor[$Elev_1$]	$Calls' = \emptyset \wedge Requests' = \{ Flr_3 \} \wedge$ Current-Floor $= Flr_2$
Move[$Elev_1$]	Current-Floor$' \leq Flr_3 \wedge$ Current-Dir $=$ up
Service-Floor[$Elev_1$]	$Calls' = \emptyset \wedge Requests' = \emptyset \wedge$ Current-Floor $= Flr_3$

our elevator the oracles require time-stamped traces of the following execution invocations:

Elevator Task:
- Set_Dir(Elev,Dir)
- Open_Door(Elev)
- Close_Door(Elev)
- Clear_Request_Button(Elev)
- Clear_Call_Button(Elev)

Controller Task:
- At_Floor(Elev,Floor)
- Request(Elev,Floor)
- Call(Floor,Dir)

Data Map Monitoring

For monitoring the implementation state, the monitor must also be able to capture the value of the relevant implementation variables and changes to those variables. For the elevator, these are the current floor, direction and goals. The values of these variables are captured for the initial state. To reduce the number of changes that must be monitored, we abstract the possible state changes into the following four "monitor Δ functions":

change_dir(elev,dir)	The elevator has turned around
change_floor(elev,floor)	The elevator has changed floor
insert_goal(elev,floor,dir)	A goal was added
delete_goal(elev,floor,dir)	A goal was deleted

Test Execution

A single test data sequence for the elevator was selected that satisfies the example test classes for both specifications; it is provided in Table 5.

During test execution, when a monitored invocation occurs, the invocation and any state changes since the last monitored invocation are captured in the exeuction profile. All of the oracles may not use all of the information in the execution profile. A graph describing the execution profile is given in Appendix B. The mapped execution profile will be shown for each specificaiton in the next subsections.

Table 5: Test Data for Elevator System

Input	Call(1, Up)	
	Delay(5.0)	
	Call(3, Down)	
	Delay(5.0)	– Let person get in
	Request(1, 6)	
	Delay(17.0)	– Let elev 2 get to floor 3
	Request(2, 2)	
	Delay(17.0)	– Let elev 1 get to floor 4
	Call(8, Up)	
	Delay(10.0)	– Let elev 1 get to floor 6
	Request(1, 10)	
	Delay(45.0)	– Let elev 1 get to floor 8
	Request(1, 12)	
	Delay(150.0)	– Let it finish the simulation

4.4 RTIL Testing

4.4.1 Oracle Use

Mapped Execution Profile

The test data sequence satisfies the RTIL test class with the following bindings: $Elev_1 = Elev_3 = 1$, $Elev_2 = Elev_4 = 2$, $Flr_1 = 1$, $Flr_2 = 3$, $Flr_3 = 6$, $Flr_4 = 2$, $Dir_1 =$ up and $Dir_2 =$ down.

The invocations and their parameters from the execution profile are mapped into specification events. Although each of the invocations in the control map would generate events, for brevity only the events that are relevant for this test class are included in Table 6. The at events for floors between goals are included because the assertion for servicing a goal is that there not be an at event between reaching the goal floor and opening the door.

Table 6: RTIL Mapped Execution Profile

T	Input Event	Elevator 1	Elevator 2
0	call(1,up)	set_dir(1,idle) open(1)	
5	call(3,down)		set_dir(2,up)
10	request(1,6)		
15		close(1) set_dir(1,up)	
20			at(2,2)
26			at(2,3) set_dir(2,idle) open(2)
28	request(2,2)		
31		at(1,2)	
36		at(1,3)	
41		at(1,4)	close(2) set_dir(2,down)
46		at(1,5)	
51		at(1,6) set_dir(1,idle) open(1)	
56			at(2,2) set_dir(2,idle) open(2)
66		close(1)	
71			close(2)

Verification

As the events are caputred and mapped, the oracle procedure checks if any of the intervals in the oracle representation

Table 7: Z Mapped Execution Profile

Time	Elevator 1	Data State Changes
0	Call-Event(1,up)	insert-goal(1,1,up)
10	Request-Event[1](6)	insert-goal(1,6,up)
15	Service-Floor[1](1)	delete-goal(1,1,up)
31	Move[1]	change-floor(1,2)
36	Move[1]	change-floor(1,3)
41	Move[1]	change-floor(1,4)
45	Call-Event(8,up)	insert-goal(1,8,up)
46	Move[1]	change-floor(1,5)
51	Move[1]	change-floor(1,6)
55	Request-Event[1](10)	insert-goal(1,10,up)
66	Service-Floor[1](6)	delete-goal(1,6,up)
81	Move[1]	change-floor(1,7)
86	Move[1]	change-floor(1,8)
92	Move[1]	change-floor(1,9)
97	Move[1]	change-floor(1,10)
112	Service-Floor[1](10)	delete-goal(1,10,up)

have been completed. One of the first completed intervals is between open(1) and close(1) and the assertion is that the interval is at least 15 seconds, which is true. A more complex example is the interval for checking the response to a call event. After each call event, the specification requires that there exist a sub-interval where an elevator reaches that floor and opens its doors. For instance, the interval for the call to floor 3 is between time 5 and the sub-interval when the floor is serviced at time 26. The assertion in that interval is that there are not any other at events for this elevator in the sub-interval, which is true.

At the end of the test execution, any uncompleted intervals are flagged as failures. In this test case, the first interval in the table is not completed and therefore marked as a failure.

4.4.2 Fault Localization

The problem is that the elevator was already at floor 1 and so did not cause an at event. Analysis of this failure reveals that the implementation was correct because the elevator was already at floor 1 and the specification was in error. At first, the fault appears to be an initial state problem that was caused by starting the test with a call to the floor 1 which is the initial location of the elevators. However, it is a more general problem than that and can occur after the elevator has been running. If the elevator services a request and then becomes idle because there are no goals left and then someone calls it to that floor, the same problem will occur.

4.5 Z Testing

4.5.1 Oracle Use

Mapped Execution Profile

The test data sequence satisfies the Z test class with the following bindings: $\mathsf{Elev}_1 = 1$, $\mathsf{Flr}_0 = 1$, $\mathsf{Flr}_1 = 6$, $\mathsf{Flr}_2 = 8$, $\mathsf{Flr}_3 = 10$.

The invocations and their parameters are mapped into schemas and the state changes are mapped from the implementation to specification name space. The mapped execution profile is in Table 7.

Verification

For each mapped invocation, when the next state in the oracle information is reached, the assertion is checked against the specification state. For example, when execution reaches Call-Event(8,up), the goals must be $\mathsf{Calls} = \{8\}$ and $\mathsf{Requests} = \{(6,\mathsf{up})\}$. Since they are, the test execution is correct up to that point. However, when execution reaches Move[1] for floor 9, the condition is false and this indicates a possible failure.

4.5.2 Fault Localization

This failure is caused by a missing branch fault and the following set of conditions: (1) The implementation keeps the current goals in a sorted list; (2) Goals are not deleted from the list until we close the door and start the elevator again, which prevents the controller from starting the elevator because it appears idle; (3) If the elevator is not idle, the search routines consider it already past the current floor, because buttons may be pressed at any time and so we assume that we have already left the "current" floor; (4) The failure occurs when there is more than one goal in the goal list, the elevator is servicing one of them, and a new goal is added. Since the first goal is considered at the end of the search path, any new goals inserted are inserted before it. Later, when the serviced goal is deleted, the other goals are out of order. Therefore, if the elevator was allowed to continue executing, it would service floor 8 and then return to service floor 6. The diagrams in Appendix B show the incorrect and correct path for the elevator.

5 Related Work

Weyuker recognizes the focus of testing research on test data selection and the underlying assumption, however ill-conceived, that the rest of the testing process – in particular, the verification of test results – is straightforward [Wey82]. The "oracle assumption" states that testers are routinely able to determine whether or not the test output is correct, and considers a program "non-testable" if either an oracle does not exist or it is impractical to determine the correct output. Weyuker discusses three approaches to testing non-testable programs and also notes some of their limitations. The first is the use of pseudo-oracles [WD81] through "dual coding", possibly in a very high-level language (VHLL) [Rau73]. This approach is not suitable for reactive systems due to the inefficiency of executable VHLL code, nor does this approach always provide independent versions [KL84, LM89]. Second, she suggests executing the program on "simplified" test data, for which correct output is known or can be easily determined, and extrapolating the resulting behavior to more complicated cases. This approach is not useful for reactive systems either, since it is often the complicated cases that are most error prone. A third alternative she discusses is to accept plausible results, but since most reactive systems are safety-critical, there is great risk in this approach as well.

Several additional approaches have skirted the oracles frontier: testing environments, test class descriptions, and executable specifications. Each of these areas of work has

relevance to some of the problems that our approach addresses. A more thorough exposition on work related to oracles is provided elsewhere [RAO92].

Testing environments support aspects of monitoring and driving the testing process. A "test driver" is a tool that exercises a system or system component providing test inputs and reporting test results [IEE83, Pan78]. A test driver sets up the target system's initial state, inputs test data, invokes the system (component), captures the output and final state, and verifies results for each test execution. A "test bed" is an environment containing the hardware, instrumentation tools, simulators, and other support software necessary for testing a system or system component [IEE83, Hua78, Stu73]. In addition to supporting the capabilities of test drivers, a test bed often collects additional test information, such as intermediate system states and timing information, which are necessary for testing reactive systems. Testing environments often support some means of describing classes of test data to exercise the system.

Test class descriptions represent oracle information by providing an abstract specification of desired tests and associate the expected behavior. Input/output pairs are a general, but degenerate, form of test class descriptions. Howden proposes a table of values to store input/output pairs [How78]. Goodenough and Gerhart [GG75] use decision tables to represent conditions on test classes and the associated actions relevant to the program's correct operation. "Test specification languages" provide a language for describing oracles as test classes, consisting of input data and expected output (and initial state and state changes). Ostrand *et.al.* describe a prime example of a test specification language, appropriately called TSL [BHO89], which describes test classes by specifying actual values for test data and expected results. "Test grammars" provide a mechanism for describing and generating test classes. Duncan and Hutchinson propose attributed grammars to describe the form of system inputs and outputs, where the attributes limit test data generation to meaningful test classes [DH81]. Bauer and Finger describe a method that derives a test grammar from an augmented finite state machine and then generates test classes, consisting of input and expected behavior [BF79]. Most of these test class description mechanisms are part of a supporting environment that drives test execution and result verification.

A relatively complete environment to testing and test verification for real-time systems modeled by an augmented finite state machine is described by Dasarathy *et.al.* [Das85, CDK85]. Sequences of stimulus/response pairs are generated to satisfy a selected test coverage criterion (to cover parts of the model, such as stimuli, responses, states, and/or transitions). Test data to satisfy these test class descriptions are run by a test driver in conjunction with a user-developed interface for the target system. Specified stimuli are caused, behavior is monitored, and responses are verified.

DIASTS is an environment that supports algebraic specification, implementation and testing of abstract data types. Each axiom of the specification is translated into a test procedure that invokes the implemented operations and includes an oracle procedure that verifies that the axiom is satisfied by relevant test executions. The user specifies test data, in the form of abstract operations and constant values that are passed to the relevant test procedures. A test driver exercises the data abstraction for the specified test data and reports any unsatisfied axioms (as well as test coverage). DIASTS requires that the representation mapping from concrete implementation to abstract specification be an explicit part of the abstract data type specification.

Executable assertions and executable specification languages provide automatic oracle information and a straightforward oracle procedure. Assertion languages allow assertions to be embedded within the implementation and thereby provide an explicit oracle that verifies system behavior for a particular test execution by constraining the system state at the assertion location. Two forms of executable assertions are interface oracles, which verify module interfaces through pre- and post-conditions on a module invocation, and trace oracles, which verify the state at arbitrary locations within the code [How78]. Anna [LvH85, San91] provides a formal language for annotating Ada implementations with assertions. Executable specification languages, such as Paisley and OBJ [ZS86, GW88, GK91], enable a form of dual coding, where the executable specification is providing a second version. Most executable specifications, however, are only partial specifications and do not specify all behavioral aspects of a system. Moreover, the mapping required between implementation results and specification execution is left for the tester and not made explicit.

Formal methods provide an analytical approach to verification, where an implementation is proved consistent with a specification. In this sense, the specification is an oracle that is verified for all possible inputs, rather than for specific test data. Of course, the specification may specify functionality or some other property, such as safety or security. Our approach combines analytical aspects of formal methods in test class generation, oracle derivation, and oracle procedures.

6 Conclusion

We have presented an approach to deriving and using specification-based oracles for the verification of test results. If the testing process does not determine whether the system behaves correctly, there is nothing to be gained by performing the tests. Typical means of test verification, such as manually examining results, are not effective. We gain unjustified confidence when oracles are not used. In general, whenever exhaustive testing is not done there remains error potential. But, we typically expect that the system has behaved correctly at least for the aspects of the system that have been tested (in particular, for the specific test data). Without an oracle, not even exhaustive testing can assure freedom from error. Furthermore, by deriving oracles from formal specifications, we are more confident that the oracles reflect the intended behavior. Of course, the specification could be faulty, so the test results should be validated against our notion of correct behavior as well. This ability to validate is an advantage of testing over strictly using analytical approaches.

We have developed a model of test oracles that supports an approach to deriving and using specification-based oracles. The model makes explicit the mapping required between the concrete name space of the implementation and the abstract name space of the specification. Our model is

general enough to allow the oracle name space to lie between these two or to coincide with either name space. The oracle name space coincides with that of the specification in the approach presented in this paper. We propose guiding oracle derivation by the testing criteria in use so that oracle information is represented for each test class generated. We discuss the techniques for deriving the oracle information and procedures for comparing it to the execution profile. We demonstrate how a monitor that satisfies the needs of all derived oracles must be developed. Finally, we discuss the application to multi-specifications and illustrate our approach for a reactive system that simulates an elevator specified in a two-way specification.

Deriving oracles from specifications for test classes and representing them in a common form encourages test class reuse and verification of behavior. During regression testing, oracles can be reused just as test data might be reused; the advantage to our approach is that test classes and the associated oracles are more general and may not need modification as often as a specific test datum/output pair would. Moreover, the synergism between testing criteria and derivation of oracles for test classes shares mutually required analysis objects and hence makes both tasks more cost-effective.

Another advantage of our oracles is that they may be represented by many formalisms and levels of abstraction. A test class may be as specific as actual test data or as general as a logical expression of some domain of test data. The oracle assertions, likewise, may specify actual output results, a computation in terms of the input, or a logical state that must be satisfied. Moreover, an oracle representation may represent a stimulus/response pair as opposed to initial/final state, which is often more appropriate for reactive systems that may not even have a final state except in erroneous circumstances. In addition, the representation also supports manually derived test classes and expected outcome, such as those a developer might create to desk check a specification. This encourages the actual testing of these data when the implementation is complete.

The derivation and use of oracles must be incorporated into software processes. Oracle information should be developed as a part of the software development process; it should be acquired as the system is specified as well as being added whenever test classes are considered. Oracles should be maintained as part of the software configuration process and remain consistent with specifications. Finally, they should be used in the software testing and maintenance (regression testing) processes. This is straightforward but only if oracles are explicitly represented, whereby they facilitate automation of the testing process. We are developing formal testing processes that proactively effect complete testing activities, integrate testing criteria [RAO89], and use oracle information to evaluate the results of the testing process.

Most phases of our approach can be automated. As discussed, it is difficult to automatically derive the mapping between the implementation space and the specification name space, so for now this must be specified or developed as a part of the design process. Once the mapping is developed, however, the oracle can be derived by symbolic interpretation (or the similar but simpler technique of resolution). In conjunction with test class generation, oracle representation is done by substitution and simplification into the oracle information. The monitor development could be automated by a monitor system parameterized by implementation procedure, function and entry call names as well as data to be monitored; these parameters are determined by the specific mappings. The remainder of the approach uses these derived components.

Future work is essential in this area. In particular, we must evaluate the tradeoffs and inaccuracies present in our approach, as pointed out in Section 3. In addition, we plan to develop a prototype oracle system that is integrated within a proactive testing process. We have designed a Larch oracle system, based on the Larch family of specification languages [GHW85]. Larch is a two-tiered language, where a control mapping between the shared language specification and the implementation is specified in the interface language specification, but deriving a Larch oracle requires a data mapping. The Larch oracle system will use the specification as oracle information to detect failures in the execution profile. The Larch language was chosen because of its rigorous semantics, public domain toolset and inclusion of the control mapping. However, Larch also has several disadvantages such as no control structures internal to the specification, which make it impossible to symbolically interpret the specification without using the implementation. Before developing a Larch oracle prototype, we will further evaluate the tradeoffs between Larch and other specification languages.

We have described one approach for integrating formal methods and testing. We are also pursuing other means of integrating the two disciplines. We are designing specification-based test criteria that select test classes based on formal specifications [ROT89]. We are developing ways of utilizing the reasoning power of formal methods to provide more effective fault detection capabilities [RT88]. And, we are investigating the formal specification and development of the testing process [RAO89]. We believe that with enhanced interaction between formal methods and testing in the assessment of software systems, software will be of genuinely higher quality.

A Ada Elevator Package Specification

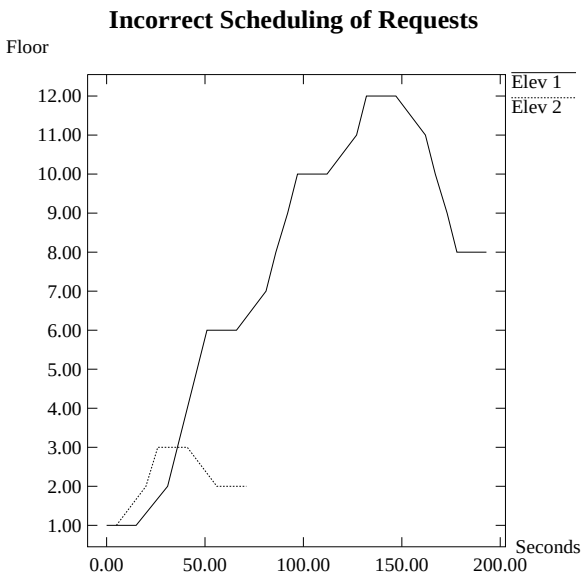
```
package Elevator_Pkg is
  -- This package is the interface to the elevators.
  Num_Of_Elevators: constant integer := 5;
  subtype Elevator_ID is integer range 1..Num_Of_Elevators;
  Num_Of_Floors: constant integer := 20;
  subtype Floor_ID is integer range 1..Num_Of_Floors;
  type Direction is (up,idle,down);
  task Elevator is
     -- Start the elevator moving the given direction
    entry Set_Dir(Elev:Elevator_ID;Dir:Direction);
     -- Open the elevator doors
    entry Open_Door(Elev:Elevator_ID);
     --Close the elevator doors
    entry Close_Door(Elev:Elevator_ID);
     -- Turn off the lights in the buttons
    entry Clear_Request_Button(Elev:Elevator_ID;
                               Floor:Floor_ID);
    entry Clear_Call_Button(Floor:Floor_ID; Dir: Direction);
     -- Turn off the elevator
    entry Shut_Down;
  end Elevator;
end Elevator_Pkg;
```

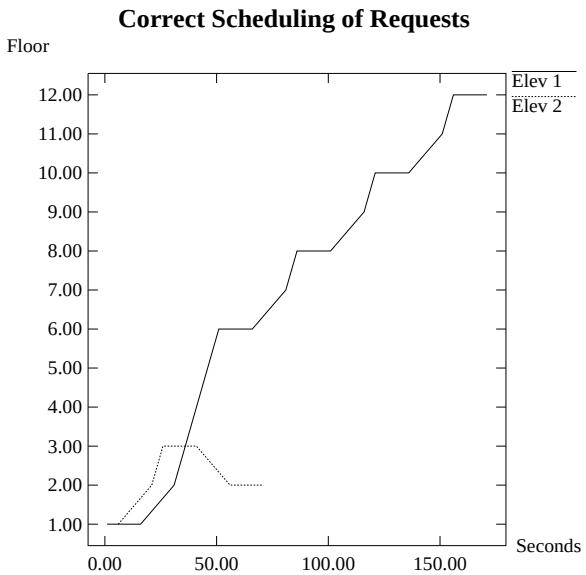
```
with Elevator_Pkg;
use Elevator_Pkg;
package Controller_Pkg is
  -- This package schedules the elevators to service
  -- the calls and requests
  task Controller is
     -- The given elevator has reached the current floor.
     -- The controller removes any relevant goals and opens
     -- the doors, if appropriate.
    entry At_Floor(Elev:Elevator_ID; Floor:Floor_ID);
     -- A user pushed a request button
    entry Request(Elev:Elevator_ID; Floor:Floor_ID);
     -- A user pushed a call button
    entry Call(Floor:Floor_ID; Dir:Direction);
     -- Turn off the elevator
    entry Shut_Down;
  end Controller;
end Controller_Pkg;
```

The implementation body appears in [RAO92].

B Test Execution Profile

The actual execution trace is too large to present here. Instead, we present a graphic view showing the location of the elevators during the course of the execution.





References

[BF79] J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *Proceedings of the Fourth International Conference on Software Engineering*, pages 425–432, Munich, Germany, September 1979.

[BHO89] Marc J. Balcer, William M. Hasling, and Thomas J. Ostrand. Automatic Generation of Test Scripts from Formal Test Specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 210–218, Key West, Florida, December 1989. ACM SIGSOFT.

[CDK85] M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto. Requirements-Based Testing of Real-Time Systems: Modeling for Testability. *IEEE Computer*, pages 71–80, April 1985.

[Das85] B. Dasarathy. Timing Constraints of Real-Time Systems: Constructs for Expressing Them and Methods of Validating Them. *IEEE Transactions on Software Engineering*, SE-11(1):80–86, January 1985.

[DH81] A. J. Duncan and J. S. Hutchinson. Using attribute grammars to test designs and implementations. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 170–177, San Diego, CA, April 1981.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[GHM87] John D. Gannon, Richard G. Hamlet, and Harlan D. Mills. Theory of modules. *IEEE Transactions on Software Engineering*, SE-13(7):820–829, 1987.

[GHW85] John Guttag, James Horning, and Jeanette Wing. "The Larch Family of Specification Languages". *IEEE Transactions on Software Engineering*, pages 24–36, September 1985.

[GK91] Carlo Ghezzi and Richard A. Kemmerer. Executing formal specifications: The ASTRAL to TRIO translation approach. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 112–122, Victoria, British Columbia, October 1991. ACM SIGSOFT, ACM Press.

[GT79] Joseph A. Goguen and Joseph J. Tardo. "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications". In *IEEE Conference on Specification of Reliable Software*, pages 170–188, 1979.

[GW88] Joseph A. Goguen and Timothy Winkler. *Introducing OBJ3*, 1988. SRI-CSL-88-99.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[How75] William E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, C-24(5), May 1975.

[How76] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2(3), September 1976.

[How78] William E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, 4(4):293–298, July 1978.

[Hua78] J. C. Huang. Instrumentation and software testing. *IEEE Computer*, 11(4):28–37, April 1978.

[IEE83] ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee. *IEEE Standard Glossary of Software Engineering Terminology, Standard 729-1983*, 1983.

[Kem85] Richard A. Kemmerer. "Testing Formal Specifications to Detect Design Errors". *IEEE Transactions on Software Engineering*, SE-11(1), January 1985.

[KL84] J. Knight and N. Leveson. A Large Scale Experiment in N-Version Programming. In *Proceedings of the Ninth Annual Software Engineering Workshop*, NASA/Goddard Space Flight Center, Greenbelt, MD, November 1984.

[LM89] Bev Littlewood and Douglas R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, December 1989.

[LvH85] David C. Luckham and Friedrich W. von Henke. An Overview of ANNA, a Specification Language for Ada. *IEEE Transactions on Software Engineering*, 2(2):9–22, March 1985.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[Pan78] D. J. Panzl. Automatic software test drivers. *IEEE Computer*, 11(4):44–80, April 1978.

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J. W. de Bakker, W.-P de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, Berlin, 1986.

[RAO89] Debra J. Richardson, Stephanie Leif Aha, and Leon J. Osterweil. Integrating testing techniques through process programming. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 219–228, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based Test Oracles for Reactive Systems. Technical Report TR-92-01, Department of Information and Computer Science, University of California, January 1992.

[Rau73] J. C. Rault. Extension of hardware fault detection models to the verification of software. In W. C. Hetzel, editor, *Program Test Methods*, pages 255–262. Prentice Hall, New Jersey, 1973.

[RC85a] Debra J. Richardson and Lori A. Clarke. "Partition Analysis: A Method Combining Testing and Verification". *IEEE Transactions on Software Engineering*, SE-11(12):1477–1490, December 1985.

[RC85b] Debra J. Richardson and Lori A. Clarke. Testing Techniques Based on Symbolic Evaluation. In T. Anderson, editor, *Software: Requirements, Specification and Testing*, pages 93–110. Blackwell Scientific, 1985.

[RG89] Rami R. Razouk and Michael M. Gorlick. A Real-Time Interval Logic for Reasoning About Executions of Real-Time Programs. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 10–19, Key West, Florida, December 1989. ACM SIGSOFT.

[ROT89] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to specification-based testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 86–96, Key West, Florida, December 1989. ACM SIGSOFT. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).

[RT88] Debra J. Richardson and Margaret C. Thompson. The Relay Model of Error Detection and Its Application. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[San91] Sriram Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 123–129, Victoria, British Columbia, October 1991. ACM SIGSOFT, ACM Press.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, New York, 1989.

[Stu73] Leon G. Stucki. Automatic Generation of Self-Metric Software. *Record of the 1973 IEEE Symposium on Software Reliability*, pages 94–100, April 1973.

[WD81] Elaine J. Weyuker and Martin Davis. Pseudo-oracles for Nontestable Programs. In *Proceedings of the ACM 81 Conference*, 1981.

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[Wey86] Elaine J. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, SE-12(11):367–375, November 1986.

[Win83] Jeannette Wing. *"A Two-Tiered Approach to Specifying Programs"*. PhD thesis, Massachussetts Institute of Technology, 1983.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

[WO80] Elaine J. Weyuker and Thomas J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

[YT89] Michal Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 53–62, Pittsburgh, May 1989.

[Zav89] Pamela Zave. A Compositional Approach to Mulitparadigm Programming. *IEEE Software*, 6(5):15–25, September 1989.

[ZS86] Pamela Zave and William Schell. Salient Features of an Executable Specification Language and Its Environment. *IEEE Transactions on Software Engineering*, SE-12(2):312–325, February 1986.