Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 05-019

Requirements-Based Testing in a Model-Based World

Mats P. Heimdahl, Michael W. Whalen, and Steven P. Miller

May 09, 2005

Requirements-Based Testing in a Model-Based World[*]

Mats P.E. Heimdahl
Dept. of Comp. Sci. and Eng.
University of Minnesota
heimdahl@cs.umn.edu

Michael W. Whalen
Rockwell Collins Inc.
400 Collins Road NE
Cedar Rapids, Iowa, 52498, USA
mwwhalen@rockwellcollins.com

Steven P. Miller
Rockwell Collins Inc.
400 Collins Road NE
Cedar Rapids, Iowa, 52498, USA
spmiller@rockwellcollins.com

ABSTRACT
Model-based software development offers new opportunities and challenges for validation and verification of safety-critical software. In this report, we describe an approach for validating the artifacts generated in a model-based development process. Our approach divides the traditional testing process into two parts: one that validates the formal model implements the high-level requirements and another that determines whether the code generated from the model is behaviorally equivalent. The focus in this report is on validation testing; in particular, we present a framework that enables objective measures of requirements coverage and provides the ability to achieve a high degree of automation.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms
Testing, Model-Based Development

1. INTRODUCTION
Traditionally, software validation and verification (V&V) has been largely a manual endeavor. Assuming that we have software requirements describing the behavior of the system, *validation* determines whether we are building the right system through reviews of the requirements and specification. *Verification* that the software satisfies its specification is archived through inspections of design artifacts and extensive testing of the implementations. In critical applications, the (V&V) phase is particularly costly and consumes a disproportionately large share of the development resources. Thus, if we could reduce the cost of V&V, dramatic cost savings could be achieved.

[*]This work has been partially supported by NASA contract NCC-01001.

In model-based development, the development effort is centered on a formal model of the proposed software system. This model is derived from some (formal or informal) high-level requirements describing the expected behavior of the software. For validation and verification purposes, this model can then be subjected to various types of analysis, for example, completeness and consistency analysis [22, 23], model checking [10, 16, 7, 9, 4], theorem proving [2, 3], and test-case generation [6, 1, 14, 12, 5, 29, 24]. Ideally, through manual inspections, formal verification, simulation, and testing we convince ourselves (and regulatory agencies) that the model behaves correctly. The implementation is then automatically and (hopefully) correctly generated from this model. There are several commercial and research tools that attempt to provide these capabilities. Commercial tools are, for example, Simulink® from the Mathworks [27], SCADE Suite^{TM} from Esterel Technologies [13] and Statemate from i-Logix [18]. Examples of research tool are SCR [23], RSML^{-e} [39], and Ptolemy [26].

The shift towards model-based development naturally leads to changes in the V&V process. There has been substantial research describing how, given a correct model, it is possible to automate much of the "down-stream" testing activities. Since the models are *executable*, they can be used as oracles for determining whether source code is functioning as intended. Also, it is possible to define notions of structural coverage for the models and to automate the generation of test cases to achieve structural coverage of models [31]. However, the *model validation* problem has been more difficult to solve. There are several questions that must be answered: What techniques can we use for model validation? Can these techniques be automated? How do we know when we have validated the model sufficiently? How can we determine whether there are missing or unstated high-level requirements?

In this report, we propose an approach that can answer some of these questions by providing objective measurement of the model-validation activities. The approach is based on defining high-level requirements as formal *properties* to be satisfied by the model. Given properties in a formal notation such as LTL [11], we can define several different structural coverage criteria for measuring coverage of properties, which in turn provides objective measures of how well the high-level requirements are covered during model validation. By extending existing techniques, it is possible to autogenerate test cases from properties to meet these structural coverage criteria. Finally, given a set of test-cases that achieve a certain level of structural coverage of the high-level requirements, it is possible to measure model coverage to objectively assess whether the high-level requirements have been sufficiently defined for the system. This approach yields several objective measurements that are not possible with traditional testing techniques, and integrates and cross-checks several of the validation and verification activities.

In the remainder of this report we will discuss the issues related to testing in a model-based world in Section 2 and the relationship between the various artifacts in model-based requirements in Section 4. To provide an example to illustrate our ideas we briefly discuss a flight guidance system in Section 3. Our contributions to requirements-based testing are introduced in Section 5 and Section 6 discusses our results and point to some future directions.

2. TESTING IN A MODEL-BASED WORLD

Figure 1 shows an overview of the different testing activities possible within a model-based development environment. We suggest dividing the testing task into two distinct activities: *model validation* testing and *conformance* testing. The former activity ensures that the model captures the behavior we really want from the software, i.e., we test the model to convince ourselves that it satisfies the "true" software requirements. The latter activity ensures that the code developed from the model (either manually or automatically) correctly implements the behavior of the model, i.e., that the implementation conforms to the specification.

Currently, to accomplish the model validation activity a set of requirements-based functional tests is developed from the informal high-level requirements to evaluate the required functionality of the model. The tests used in this step are developed from the informal requirements by domain experts, much like requirements-based tests in a traditional software development process. Key questions here are (1) when have we developed enough requirements-based tests and (2) can this process can be automated to some extent. Note that we want to use the high-level requirements as the oracle for these tests, not the model, as the model serves as the "implementation" of these requirements.

Given a formal modeling language, it is possible to define notions of structural coverage over a model, similar to structural coverage on source code. We can use these coverage metrics to measure how well we have tested the functionality of the model. It is likely that the initial set of requirements-based functional tests will not completely test all model structures; for example, there may be transitions and conditions not exercised by these tests. Therefore, the functional tests will in most cases have to be supplemented by a collection of white-box tests developed specifically to exercise a model up to a certain level of specification coverage.

When testing of the model has been completed and we are convinced that the model is correct, the testing process can switch from model validation testing to implementation conformance testing. In conformance testing, we are interested in determining whether the implementation correctly implements the formal model. In this stage, the formal model is now assumed to be correct and is used as an oracle. The testing activity confirms that for all tests, the implementation yields the same result as the formal model.

All tests used during the validation testing of the formal specification can naturally be reused when testing the implementation (step 2 in Figure 1). The test cases developed to test the model provide the foundation for testing the implementation, and the executable model serves as an oracle during the testing of the implementation. Again, this test set may not provide adequate coverage of the implementation and will most likely have to be augmented with additional test cases (step 3 in Figure 1).

The characteristics of the two testing activities outlined above are radically different. The differences between the activities manifest themselves in how we determine whether we have "good" test suites and how we verify that the system satisfies the tests. For model validation, we want to ensure that the system behaves correctly in realistic operational scenarios. Good model validation test suites should have the following characteristics:

Realistic Environments: Each test input vector should correspond to a reasonable scenario in the expected environment of the software.

Specificity / Traceability: Each test should be traceable back to one (or a handful) of requirements that it is specifically designed to check.

Similarity: Tests that check different aspects of a particular requirement or related requirements should be similar, to support easy manual validation of whether the software has passed the tests.

Completeness: There should be sufficient tests to check all of the high-level requirements.

Note that extensive tool support to help us generate tests from the model could be provided to aid in model validation (see Section 5.2), but the usefulness of the test and expected outcome (oracle) will always have to be provided manually. That is, we may be able to help automate the generation of input sequences but the determination if the test is realistic and yields the right outputs must be manually checked by the domain experts.

Conformance testing, on the other hand, has quite different requirements. In this activity, we assume that the model is correct, so we are only interested in ensuring that we have sufficient tests to find any discrepencies that exist between the behavior of the formal model and its implementation. Thus, the only important requirement for test suites is *completeness*: ensuring that we have covered the model in sufficient detail to obtain a high level of confidence that the model and the implementation behave equivalently. Also, in conformance testing, the model acts as an oracle. Therefore, conformance testing could potentially be completely automated: tests can be generated from the model or implementation, and the model-as-oracle decides whether the implementation passes the test.

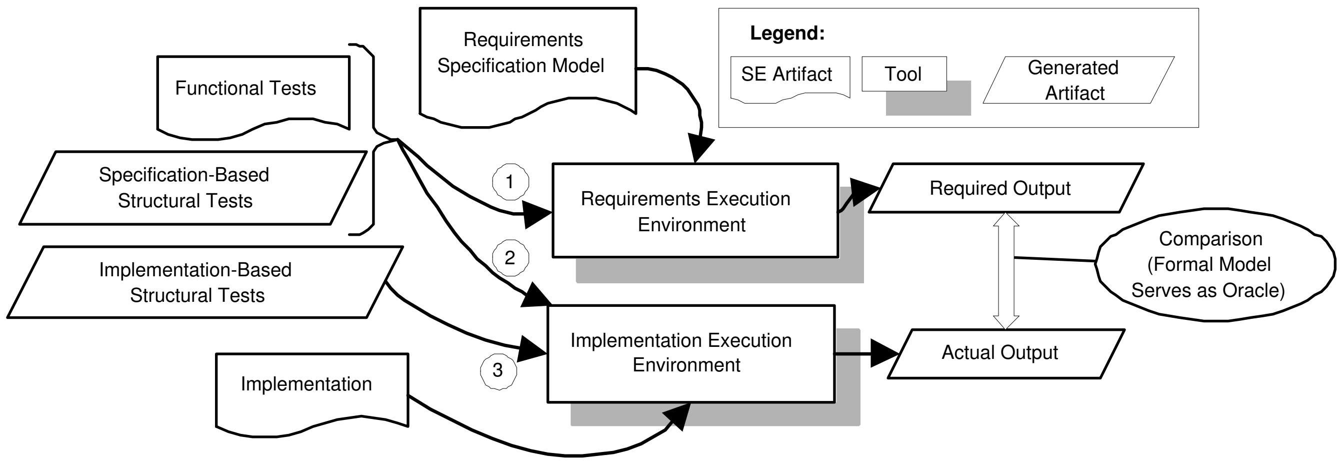
Legend:
SE Artifact
Tool
Generated Artifact
Requirements Specification Model
Functional Tests
Specification-Based Structural Tests
Implementation-Based Structural Tests
Implementation
1
2
3
Requirements Execution Environment
Implementation Execution Environment
Required Output
Actual Output
Comparison (Formal Model Serves as Oracle)

Figure 1: An overview of the specification centered testing approach.

3. CASE EXAMPLE: THE FLIGHT GUIDANCE SYSTEM

To provide a framework in which to discuss our approach to requirements-based testing we will us an example from commercial avionics—a Flight Guidance System (FGS). A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In our work we have focused on the mode logic.

The primary modes of interest in the FGS are the horizontal and vertical modes. The horizontal modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical modes control the behavior of the aircraft about the vertical, or pitch, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft's behavior.

4. REQUIREMENTS, MODELS, AND PROPERTIES

In our discussions in this report we will adopt the "World and the Machine" view of requirements put forth by Jackson and Zave [25, 17]. Machines are built to bring about some changes in the World; when the Machine is introduced to a World it interacts with that World through a shared interface. Requirements are a collection of statements about phenomena in the World that we want the Machine to help make true. A Specification is a set of statements that describes the behavior of the Machine as observed at the interface between the World and the Machine. The commonly used notions of "user requirements" or "high-level requirements" are often used to describe what Jackson and Zave call "Requirements", and "requirements specifications" or "low-level requirements" are typically used to describe their notion of "Specification".

The view briefly mentioned above is to some extend reflected in critical systems standards. ARP 4754 [37] provides guidelines for the system-level processes and DO-178B [35] provides guidelines for the software development processes in the avionics world. The notion of high-level software requirements lie at the border between the systems engineering activities covered by ARP 4754 [36] and the software engineering activities governed by DO-178B. Of particular interest here is the relationship between what these standards call the high-level requirements and the low-level requirements. The definitions used in DO-178B are:

High-level requirements: Software requirements developed from analysis of system requirements, safety-related requirements, and system architecture.

Low-level requirements: Software requirements derived from high-level requirements, derived requirements, and design constraints from which source code can be directly implemented without further information.

In general, we can roughly equate the notion of requirements from [25, 17] with high-level requirements and the notion of specification with low-level requirements. As we move to model-based development, the specification (or low-level requirements) will be largely replaced by the formal model. Nevertheless, when moving to a model-based world, the relationship between requirements, specifications, models, and properties has, in our opinion, not been adequately explored. Below we will present our view of how requirements, models, and formal properties fit together to form the basis of a complete requirements specification.

The relationship between the various artifacts is illustrated in Figure 2. In model-based development, the high-level requirements are used as the basis of the development of the model. We develop the model using a constructive modeling technique that yields an executable model that is typically easily accepted by practicing engineers. This model is developed to a level of detail where it could be used as a basis for manual coding or automated code generation without

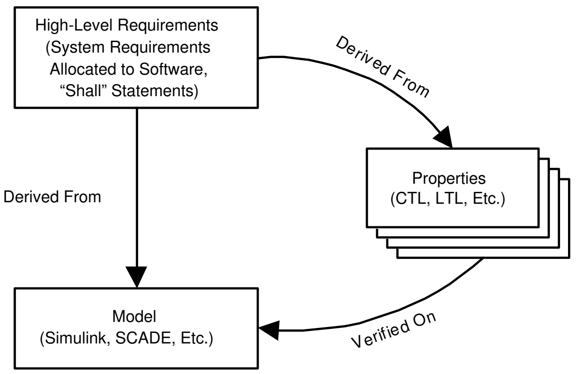
High-Level Requirements (System Requirements Allocated to Software, "Shall" Statements)
Derived From
Properties (CTL, LTL, Etc.)
Derived From
Model (Simulink, SCADE, Etc.)
Verified On

Figure 2: Relationships between requirements, properties and models.

any additional information. In this sense, the model serves as a specification of the software (or replaces the low-level requirements as defined in DO-178B).

The problem is now to determine if this model accurately captures the informal high-level requirements. The properties, captured using, for example, CTL or LTL, are very close in structure to the informal "shall" statements making up the requirements; in fact, in [28] we argue that the informal "shall" requirements are simply desirable properties of our model expressed in an informal and ambiguous notation. There is not generally a one-to-one relationship between many of the informal high-level requirement and the properties. One requirement may lead to a number of properties. For example, consider the requirement below taken from our flight guidance system.

> *"Only one lateral mode shall be active at any time."*

This requirement leads to a collection of properties we would like our model to possess. The number of properties depends on which lateral modes are included in this particular configuration of the flight guidance system; for example, an FGS may have five lateral modes leading to the following properties (stated informally).

1. *IF ROLL is active, HDG, NAV, LGA, and LAPPR shall not be active.*
2. *If HDG is active, ROLL, NAV, LGA, and LAPPR shall not be active.*
3. *If NAV is active, ROLL, HDG, LGA, and LAPPR shall not be active.*
4. *If LGA is active, ROLL, HDG, NAV, and LAPPR shall not be active.*
5. *If LAPPR is active, ROLL, HDG, NAV, and LGA shall not be active.*

On the other hand, some high-level requirements, for example, the requirement below (also from the FGS), lend themselves directly to a one-to-one formalization in the vocabulary of the model.

> *"If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged."*

For example, the requirement above can be formalized as, in this case, a CTL property for verification in the model checker NuSMV.

```
AG((!Onside_FD_On & !Is_AP_Engaged)->
    AX(Is_AP_Engaged -> Onside_FD_On))
```

The property states that it is always the case (AG) that if the Onside FD is not on and the AP is not engaged, in the next instance in time (AX) if the AP is engaged the Onside FD will also be on. Thus, capturing the informal high-level requirements as properties typically involves a certain level of refinement and extension to express the properties in the vocabulary of the formal model and to shore up any missing details to make formalization possible. For this reason, the formalization of the requirements is typically deferred until at least a partial model is available.

By formalizing these "shall" statements in a property specification language such as CTL or LTL we achieve several benefits. First, we can now use tools such as model checkers and theorem provers to verify that the properties (requirements) indeed hold in the model, an assurance that testing alone cannot provide. Second, through the act of formalization and analysis, we will most likely discover that many properties do not hold in the model. If that is the case, either the model or the property must be modified. In our experience, it is quite likely that the model is correct and the high-level requirement from which the property was developed was wrong or poorly written. Thus, the formalization and analysis provides a valuable crosscheck of the informal-high-level requirements, the formal properties, and the model. Finally, with the formal properties derived from the high-level requirements and the model replacing the low-level requirements, we are better equipped to understand and explore the notion of tests derived from the high-level and low-level requirements.

From the discussion above, there are several issues that merit further scrutiny. First, given the close relationship between the informal requirements and the property specifications, can the properties be used to help us objectively measure how well we have exercised the requirements on our system under development? Second, given that the properties represent requirements, can the properties be used to somehow help us generate requirements-based tests? These issues will be discussed further in the remainder of this paper.

5. REQUIREMENTS-BASED TESTING

The goal of the testing process remains largely the same in traditional and model-based software development: to ensure that the software implementation behaves in accordance with its stated requirements. To achieve this goal, we must demonstrate (among many other things) the following objectives:[1]

[1]Similar testing objectives can be found in common standards, for example, DO-178B tables A-6 and A-7

1. The executable code complies with the high-level requirements.
2. The executable code complies with the specification (low-level requirements).
3. Test coverage of high-level requirements is achieved
4. Test coverage of specification (low-level requirements) is achieved
5. Test coverage of the executable code is achieved

To satisfy these objectives, test cases are derived from the requirements as well as the specification.

Recent developments in test-case generation techniques have made it possible to generate vast numbers of test cases from a formal model [6, 1, 14, 12, 5, 29, 24]. Unfortunately, generating tests from the formal model and then using them to test the formal model tells us little about the correctness of the model. This approach would be an activity analogous to white-box testing of an implementation; an approach to testing that will not expose errors of omission and is designed to exercise the structure of the model (or implementation) rather than demonstrating compliance with the required operation of the system. In particular, the model cannot be used as an oracle to determine if a test has the desired outcome. This decision must be made with respect to the high-level software requirements. Unfortunately, to our knowledge, there is no objective measure of "requirements coverage". Instead, coverage is demonstrated through a best effort using traditional techniques such as partition analysis, the category partition method, and boundary value analysis. These tests must represent realistic operational scenarios and are extensively validated by domain-experts.

When we have the code in hand, we are in a position to objectively measure the code coverage achieved when executing the requirements-based tests and specification-based tests. Should this coverage be too low, we must either

- develop more requirements-based test cases to cover the uncovered code,
- extend the requirements in case the uncovered code reveals desirable behavior not stated in the requirements,
- determine that the code is valid but has been deactivated, or
- declare the code superfluous (dead) and remove it from the implementation.

With the completion of this step we have demonstrated test coverage of the implementation (objective 5). When the tests are executed on the object code running on the target platform, correct execution of the tests demonstrates the object executable code complies with the high-level and low-level requirements (objectives 1 and 2).

As the role of traditional requirements changes as described in Section 4 and the use of code generation from a model become prevalent, the role of requirements and specifications in the testing process changes in that "specification" can be replaced with "model".

5.1 Structural Coverage of Requirements

As mentioned in Section 4, there is a close relationship between the informal requirements and the properties captured for verification purposes. As an example, consider the requirement from the sample FGS in Figure 3.

"If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged."

```
AG((!Onside_FD_On & !Is_AP_Engaged) ->
    AX(Is_AP_Engaged -> Onside_FD_On))

G(((!Onside_FD_On & !Is_AP_Engaged) ->
     X(Is_AP_Engaged -> Onside_FD_On))
```

Figure 3: Our sample requirement from the Flight Guidance System presented informally, in CTL, and in LTL.

In our work we translated this requirement into equivalent Computational Tree Logic (CTL) and Linear Time Logic (LTL) formula and verified that our model of the FGS satisfied this requirement using a model checker (NuSMV). The properties are very similar in structure to the natural language requirements and this translation was quite easy to perform manually.

An analyst developing test cases from the informal requirements might derive the scenario below to demonstrate that the requirement is met.

1. Turn the Onside FD off
2. Disengage the AP
3. Engage the AP
4. Verify that the Onside FD comes on

Does this adequately cover this requirement? Should we develop a test-case to demonstrate that the Onside FD is not turned on inadvertently? If so, what would such a test-case look like? The specification of the requirement as a property allows us to define several objective criteria with which to determine whether we have adequately tested the requirement.

Below we discuss two different classes of structural coverage for properties. One is directly derived from the syntax of the property, and is similar to a structural coverage criteria defined for source code. The other is derived indirectly: an LTL property can be used to construct a Büchi automaton [11] and coverage criteria can be defined on the resulting graph.

5.1.1 *Structural Coverage Criteria over Property Syntax*

There are several kinds of structural coverage criteria that have been investigated for source code and for executable modeling languages [35, 14, 33]. It is possible to adapt some of these structural coverage criteria devised for modeling languages to fit subsets of CTL and LTL.

For example, decision coverage of the requirements would require a single test case that demonstrates that the requirement is satisfied by the model. Typically, this is too weak

	Is_AP_Engaged	**Onside_FD_On**	**X(Is_AP_Engaged)**	**X(Onside_FD_On)**	**Requirement**
1	F	T	T	F	T
2	T	F	T	F	T
3	F	F	T	T	T
4	F	F	F	F	T
5	F	F	T	F	F
6	F	F	T	F	F
7	F	F	T	F	F
8	F	F	T	F	F

Table 1: Truth assignments that will give us MC/DC-inspired coverage of our sample requirement.

of a coverage criterion since it is often possible to derive a vacuous test case. Considering our formalized requirement in Figure 3, we can satisfy the property by creating a test case that leaves the autopilot disengaged thoughout the test and verifying that the flight director does not come on. Although this test case satisfies the requirement, it does not shed much light on the correctness of our model.

A better alternative would be to adopt one of the more rigorous structural coverage criteria used in the avionics software domain, such as, the Modified Condition/Decision Coverage (MC/DC) criterion, for us in the requirements-based testing domain. This criterion was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications [8]. MC/DC was developed as a practical and reasonable compromise between decision coverage and multiple condition coverage. It has been in use for several years in the commercial avionics industry. The important aspect of this criterion is the requirement that testing should demonstrate the independent effect of atomic boolean conditions on the boolean expressions in which they occur.

A test suite is said to satisfy MC/DC if executing the test cases in the test suite will guarantee that:

- every point of entry and exit in the model has been invoked at least once,
- every basic condition in a decision in the model has taken on all possible outcomes at least once, and
- each basic condition has been shown to independently affect the decision's outcome

where a basic condition is an atomic Boolean valued expression that cannot be broken into Boolean sub-expressions. A basic condition is shown to independently affect a decision's outcome by varying only that condition while holding all other conditions at that decision point fixed. Thus, a pair of test cases must exist for each basic condition in the test-suite to satisfy MC/DC. However, test case pairs for different basic conditions need not necessarily be disjoint. In fact, the size of MC/DC adequate test-suite can be as small as $N + 1$ for a decision point with N conditions.

The MC/DC criterion requires several test-cases to test that the structure of a requirement has been adequately exercised. To achieve a coverage criteria similar to MC/DC on the requirements, it is necessary to systematically construct test cases that demonstrate that every basic condition in the property can have an impact on the truth value of the full property. Consider the truth assignments in Table 1. Here, a pair of truth assignments constitute an MC/DC pair. For example, the truth assignments in rows 1 and 5 are designed to demonstrate that Onside_FD_On can have an independent impact on the truth value of our sample requirement; with the truth assignment in row 1 the property is satisfied and with the assignment in row 5 (only differing from row 1 in the value of Onside_FD_On) the property is false. If we can find a test-case that exhibits the truth assignment in row 1 we have demonstrated one way of satisfying our requirement.

It is important to note that there is an important distinction between traditional MC/DC coverage and this coverage metric over properties. If our model is correct, we will not be able to find test cases that cause the property to be false. For the moment, we are interested in developing the test cases for the truth assignments that satisfy the requirement (cases 1–4). In the future, we may also investigate using automated tools to try to generate test cases for the negative assignments; experience has shown that properties rarely hold in initial versions of systems, so violating test cases would also be useful.

5.1.2 Structural Coverage Criteria over Büchi Automata

As an alternative to defining requirements test adequacy criteria over the property syntax, we can study the automata accepting sentences in the languages defined by the properties. Algorithms automating the synthesis of automata from temporal logic properties exist and we can construct automata, such as, a Büchi automaton, from LTL properties [11]; these algorithms can be easily incorporated in specification testing tools. We used the algorithm implemented in the SAL toolset from SRI International [38] to translate our property into the Büchi automaton of Figure 4.

The states with a double border are accepting states, i.e., any cyclic path through the automaton with an accepting state as part of the cycle is considered successful. With this description of the requirement we can start to objectively assess how well a set of test-cases covers the requirement.

If we replay the manually developed test-case above in the state machine in Figure 4, the test-case would start in state S0 with the Onside FD on and the AP engaged. When both

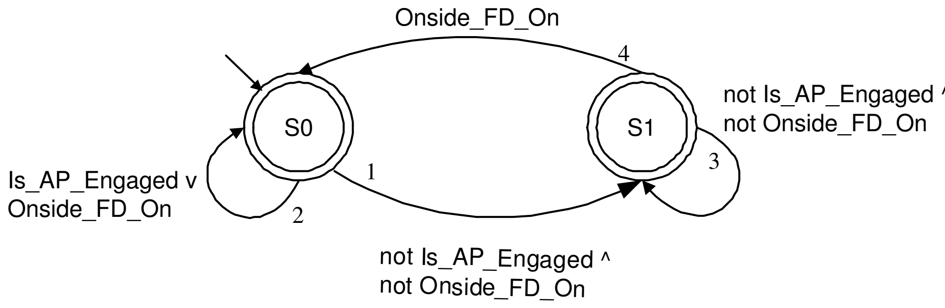

Figure 4: The requirement in above captured as a Büchi automaton.

the Onside FD have been turned off and the AP disengaged, the state machine would take transition 1 to state S1. As we engage the AP, if the model is correct, the Onside FD will come on and the state machine wold take transition 4 back to S0. Since this is an accepting state and part of a cycle, it is an acceptable requirements-based test. Note that this state machine defines both the inputs (AP engaged) as well as the expected output (Onside FD); exactly what we would expect from the high-level requirements.

Studying the state machine in Figure 4 we see that the test-case we developed manually really only covers part of the acceptable behavior of our system; there are many paths through this state machine that describe valid behaviors of the system. Coverage of these paths would be an objective measure of high-level requirements test coverage that we might want to consider. For example, we might want to make sure we cover all lassos (a lasso consists of a cycle free path from an initial state to an a accepting state followed by a cycle free path back to the same accepting state; the complete path will look like a lasso). This would force us to consider several additional test cases. For example:

1. Turn the Onside FD off and disengage the AP ($S0 \rightarrow S1$)
2. Leave the AP disengaged for one step ($S1 \rightarrow S1$)
3. Verify that the Onside FD does not come on

The test-case again turns the Onside FD off and disengages the AP. This will cause the state machine to take transition 2 from S0 to S1. Now we leave the AP disengaged for one execution cycle. Since the Onside FD shall not be turned on in this case, will take transition 3 from state S1 back to S1. Since this is an accepting state we have a lasso and a valid test sequence. This sequence exercises the scenario where the AP is never engaged and, consequently, the Onside FD does not need to be turned on. As should be clear from this discussion, we now have a systematic way of developing high-level requirements-based test cases as well as an objective way of measuring how well we have exercised the various high-level requirements.

Note here that the algorithm used to generate the Büchi automaton will influence the selection of test-cases. The automaton created in SAL is very compact (two states and four transitions) and has complex conditions on the transition (conditions involving *or* and *not*). A generalized Büchi automaton for our property created with the algorithm described by Gerth, Peled, Vardi, and Wolper [15] would yield nine states and 33 transitions. In this case, however, the conditions on the transitions would be simpler. Naturally, the test-cases required to yield lasso-coverage would be very different depending on which Büchi automaton we used as a basis for the test-case generation; a larger automaton with more lassos may give us a better suite of test-cases. Alternatively, one could require that every lasso in Figure 4 be exercised up to MC/DC coverage. This would require multiple test-cases per lasso and presumably provide a better test suite than simply requiring lasso-coverage. These tradeoffs in terms of test suite size and effectiveness of test-cases is completely unexplored and we hope to address this issue from both a theoretical as well as empirical perspective in the future.

Although there are many unknowns, we believe that test cases developed to some objective requirements coverage criterion, for example, property MC/DC coverage or lasso coverage, will likely exercise the requirement of interest in a more comprehensive manner than manually developed tests. At this time, however, it is unclear which coverage coverage criteria would be useful in practice. For example, our lasso criterion required us to develop a test-case that leaves disengaged. Does this add any value in detection of faults? This is just one of many questions that must be addressed in future work. To the best of our knowledge, there has been no research in this area and it is unclear what coverage measures would be suitable.

To summarize, viewing the high-level requirements as formal properties provides a mechanism where we can objectively measure how well our test suite covers the requirements; a capability that has not been previously available. Formalizing the requirements also provides us with the capability to automate the generation of requirements-based tests from the actual high-level requirements. This topic is discussed in the next section.

5.2 Automation of Test Generation

Given that the high-level requirements can be captured as formal properties, we can explore the possibility of automating the process of deriving high-level requirements-based tests. Several research efforts are underway to automate the generation of tests from the formal models. For example, the Critical Systems Research Group at the University of Minnesota have explored using model checkers to generate tests to provide structural coverage of formal models [21, 33, 31, 32]. Reactive Systems Inc. [34] has a commercial tool Reactis that uses heuristic search techniques to generate tests from Simulink [27] and Stateflow models. Another commercial tool, T-VEC [5] uses a constraint-based approach to automatically generate test cases from a number of different modeling languages, including Simulink.

In our work we have focused on the potential of model-checkers as test-case generation tools [31, 21, 20, 19]. Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [11]. Should a property violation be detected, the model checker will produce a counterexample illustrating how this violation can take place. In short, a counterexample is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test-case testing this transition—the sequence of inputs must take the model to state A; in state A, C must be true, and the next state must be B. This is a property expressible in the logics used in common model checkers, for example, LTL [30]. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such input sequence) and start verification. We call such a property a *trap property* [14]. The model checker will now search for a counterexample demonstrating that this trap property is, in fact, satisfiable; such a counterexample constitutes a test case that will exercise the transition of interest. By repeating this process for each transition in the formal model, we use the model checker to automatically derive test sequences that will give us transition coverage of the model. This general approach can be used to generate tests for a wide variety of structural coverage criteria, including coverage of requirements as described in Sections 5.1.2 and 5.1.1.

To automate the generation of high-level requirements based test, we would translate the informal high-level requirements to formal properties. These properties can then be translated to the suitable notation for an existing test-case generation tool, in our case a model-checker. We can then instruct the test-case generation tools to generate tests to cover our properties (our requirements) as opposed to the model of the system behavior. This approach would give us the capability to automatically generate tests to provide a predefined coverage of our requirements.

As an example, if we translate the model of our FGS example into the input language of a model checker, the following four trap properties would allow us to generate four test cases to exercise our sample requirement up to MC/DC coverage.

```
1. ~AG(~Onside_FD_On & Is_AP_Engaged &
       AX(Is_AP_Engaged) & ~AX(Onside_FD_On));

2. ~AG(Onside_FD_On & ~Is_AP_Engaged &
       AX(Is_AP_Engaged) & ~AX(Onside_FD_On));

3. ~AG(~Onside_FD_On & ~Is_AP_Engaged &
       AX(Is_AP_Engaged) & AX(Onside_FD_On));

4. ~AG(~Onside_FD_On & ~Is_AP_Engaged &
       ~AX(Is_AP_Engaged) & ~AX(Onside_FD_On));
```

Although generating tests to cover the requirements as opposed to the model seems like a simple task in theory, the formal notion of "requirements coverage" is, to the best of our knowledge, completely new and no existing tool supports this activity.

6. DISCUSSION

The move from traditional to model-based development of software provides opportunities as well as challenges for validating and verifying critical software systems. In this report, we point out that model-based development is a natural evolution of the traditional development process. In our view, the informal high-level software requirements are derived from the system requirements, safety requirements, and system architecture. The model is developed based on these requirements and captures the behavior described in the high-level requirements. Any additional information needed for implementation not captured in the high-level requirements is captured in the model; the model is a specification of the software behavior. The high-level requirements are formalized as properties that can then be verified to hold in the model. We view this collection of artifacts to be the requirements specification of the system under development.

The model-based development paradigm offers several new opportunities for testing discussed in his report, including the opportunity to provide objective measures of requirements coverage (Sections 5.1.1 and 5.1.2) in addition to the traditional code coverage measured today. To our knowledge, the notion of requirements coverage has not been previously addressed in any formal way. Furthermore, the formal models and properties enable us to automate large portions of the test-case development and test execution process (Section 5.2).

There are several areas related to requirements-based testing that merit further study. First, since the notion of formal requirements-based testing is new, there is little experience with how to best capture the informal requirements as formal properties as well as what coverage criteria to use after the requirements have been formalized. Both issues naturally will have an impact on the test cases developed and, consequently, on the effectiveness if the resultant test-suites. There is a great need for empirical studies to determine the effectiveness of various test-case generation techniques on realistic examples.

Second, there are several important issues that must be addressed to be able to automate the generation of requirements-based tests. Since "good" requirements test cases document realistic operational scenarios (as discussed in Section 2), we must constrain the search space used for generating test cases with automated tools. For example, if a model takes an altitude reading as an input, a realistic test scenario may require that the altitude changes be limited by a certain maximum altitude rate. Such constraints on the input variables can typically be added to the testing tool to limit its choices when the test cases are generated. The situation becomes more complex if the constraint on the environment involves several interrelated input variables. For instance, we may have two altitude reports provided from two redundant altimeters. These altimeters also provide information if they have passed an internal self-check. A constraint may now be that if both altimeters report that they are operating correctly, their altitude reports must be within some threshold of each other; a constraint that is less trivial to capture and enforce in the test-case generation tools of which we are aware. The ability to generate test cases that closely resemble what a skilled engineer would generate manually from the informal requirements will, in our opinion, be critical for the acceptance of automated techniques by practicing engineers and regulatory agencies.

Finally, we must explore the relationship between requirements-based structural coverage and model-based structural coverage. Given a "good" set of requirements properties and a test suite that provides a high level of structural coverage of the requirements, is it possible to achieve a high level of structural coverage of the formal model and of the generated code? That is, does structural coverage at the requirements level translate into structural coverage at the code level? If it is possible to achieve a high-level of code coverage from structurally complete requirements tests, then it might be possible to autogenerate tests that are immediately traceable to the requirements and that meet many of the regulatory requirements in standards such as DO-178B [35].

7. REFERENCES

[1] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, Nov. 1998.

[2] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, 1998.

[3] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In *Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, pages 89–107, San Jose, CA, January 1999. IEEE Computer Society.

[4] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. In *First ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.

[5] M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97*, June 1997.

[6] J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *Proceedings of the SPIN Workshop*, August 1996.

[7] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[8] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[9] Y. Choi and M. Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pages 109–118, Tokyo, Japan, October 2002.

[10] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transaction on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[12] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proceedings of TACAS'97, LNCS 1217*, pages 384–398. Springer, 1997.

[13] Esterel technologies. scade suite product description. http://www.esterel-technologies.com.

[14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[15] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18. Chapman & Hall, Ltd., 1996.

[16] O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[17] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May/June 2000.

[18] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[19] M. P. Heimdahl and G. Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[20] M. P. Heimdahl, G. Devaraj, and R. J. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.

[21] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.

[22] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[23] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[24] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, Grenoble, France, April 2002.

[25] M. Jackson. The world and the machine. In *Proceedings of the 1995 Internation Conference on Software Engineering*, pages 283–292, 1995.

[26] E. A. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[27] Mathworks inc. simulink product web site. via the world-wide-web: http://www.mathworks.com.

[28] S. P. Miller, M. P. Heimdahl, and A. Tribble. Proving the shalls. In *Proceedings of FM 2003: the 12th International FME Symposium*, September 2003.

[29] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.

[30] A. Pnueli. Applications of temporal logic to specification and verification of reactive systems: A survey of current trends. *Department of Applied Mathematics, The Weizman Institute of Science*.

[31] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.

[32] S. Rayadurgam and M. P. Heimdahl. Test-Sequence Generation from Formal Requirement Models. In *Proceedings of the 6th IEEE International Symposium on the High Assurance Systems Engineering (HASE 2001)*, Boca Raton, Florida, October 2001.

[33] S. Rayadurgam and M. P. Heimdahl. Generating mc/dc adequate test sequences through model checking. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW-03*, Greenbelt, Maryland, December 2003.

[34] Reactive systems inc. reactis product description. http://www.reactive-systems.com/index.msp.

[35] RTCA. *Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[36] *ARP 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems*. SAE International, November 1996.

[37] *ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[38] SAL' product web site. via the world-wide-web: http://www.csl.sri.com/projects/sal/.

[39] J. M. Thompson and M. P. Heimdahl. An integrated development environment prototyping safety critical systems. In *Tenth IEEE International Workshop on Rapid System Prototyping (RSP) 99*, pages 172–177, June 1999.