# Model Validation using Automatically Generated Requirements-Based Tests *

Ajitha Rajan
Dept. of Comp. Sci. and Eng.
University of Minnesota
arajan@cs.umn.edu

Michael W. Whalen
Advanced Technology Center
Rockwell Collins Inc.
mwwhalen@rockwellcollins.com

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

## Abstract

*In current model-based development practice, validation that we are building a correct model is achieved by manually deriving requirements-based test cases for model testing. Model validation performed this way is time consuming and expensive, particularly in the safety critical systems domain where high confidence in the model correctness is required.*

*In an effort to reduce the validation effort, we propose an approach that* ***automates the generation of requirements-based tests*** *for model validation purposes. Our approach uses requirements formalized as LTL properties as a basis for test generation. Test cases are generated to provide rigorous coverage over these formal properties. We use an abstract model in this paper–called the* ***Requirements Model****—generated from requirements and environmental constraints for automated test case generation. We illustrate and evaluate our approach using three realistic or production examples from the avionics domain. The proposed approach was effective on two of the three examples used, owing to their extensive and well defined set of requirements.*

## 1 Introduction

In model-based development, the development effort is centered around a formal model of the proposed software system. It is thus critical to validate the model to ensure it satisfies the high-level requirements. Traditionally, model validation has been largely a manual endeavor wherein developers *manually* create requirements-based tests and inspect the model to ensure it satisfies the requirements. Figure 1 illustrates the traditional model validation approach. In the critical systems domain, the validation and verification (V&V) phase can be very costly and consumes a majority of the development resources. In this paper, we attempt to reduce the model validation effort by proposing an approach that *automates* the generation of requirements-based tests. In the rest of this paper, we refer to the model being validated as the Model Under Test (MUT).

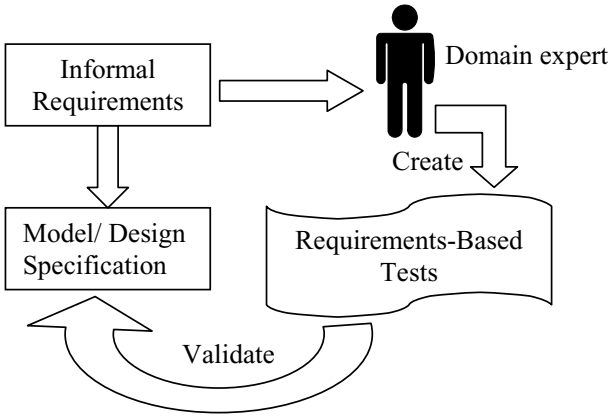


**Figure 1. Traditional Model Validation Approach**

Our approach uses a formalized set of requirements, as illustrated in Figure 2, as the basis for automated requirements-based test case generation. Generally, requirements are defined informally as, for example, "shall" statements or use-cases. Recent efforts (e.g., [11]) have shown that formalizing software requirements using notations such as temporal logics [4] and synchronous observers [6] is both possible and practical.

Given formal requirements and a system model to test (MUT), we showed in [17] that it is possible to automatically generate requirements-based tests to provide coverage over the requirements (as opposed to over the model). We defined a collection of coverage metrics over the structure of formal requirements for this purpose. Nevertheless, our previous approach cannot be used in MUT validation since it *uses the MUT itself* when generating requirements-based tests. In general, when testing to check whether an artifact

*This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

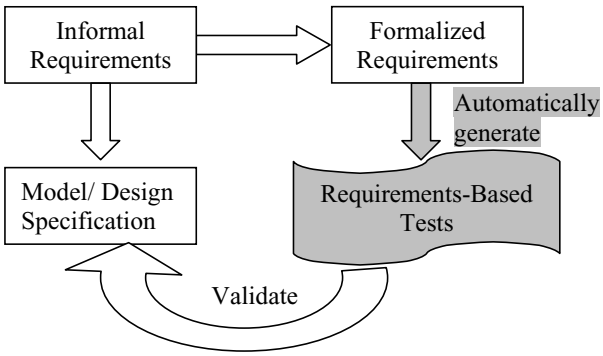


**Figure 2. Proposed Model Validation Approach**

satisfies a set of requirements, it is highly undesirable to derive the tests with guidance from the artifact under test.

To overcome this problem, we propose to alter our previous approach so that it uses an abstract model—we call it the *Requirements Model*—different from the MUT for requirements-based test case generation. This *requirements model* must contain only the behaviors *required* from the MUT, and not be constrained to the behaviors actually *provided* by the MUT. Given formalized requirements, constraints on the environment of the system, and information about the types of the inputs and outputs of the MUT, we can create an abstract model—independent from the MUT—that captures only the required behavior. In our work we create the requirements model by encoding the requirements and environmental assumptions as invariants. By using this requirements model as a basis for test case generation we can generate truly black-box requirements-based tests for MUT validation.

We assess the effectiveness of the proposed approach for model validation on three realistic models from the avionics domain: the Mode Logic for a Flight Guidance System (FGS), and two models related to the display window manager for a new airliner (DWM_1, and DWM_2). For each of these models we automatically generate requirements-based tests using the requirements model to provide Unique First Cause (UFC) coverage over the formal requirements [17]. UFC is a requirements coverage metric based on the Modified Condition Decision Coverage (MC/DC) criterion widely used in critical avionics systems [3]. We then run the generated test suite on the MUT and measure the *model coverage* achieved. In particular, we measure MC/DC achieved over the MUT.

From our experiment we found that the requirements-based tests did extremely well on the DWM_1 and DWM_2 systems (both production models) achieving more than 95% and 92% MC/DC coverage over the models. On the other hand, the requirements based tests for the FGS (a large case example developed for research) performed poorly providing only 41% coverage of the FGS model. We hypothesize that the poor results on the FGS were due to the inadequacy in the FGS requirements set; the requirements sets for the DWM_1 and DWM_2 systems, on the other hand, were developed for production and were extensive, well validated, and well defined. These experiences indicate that our approach can be effective in the validation testing of models in model-based development. In addition, the capability of measuring coverage of the requirements as well as the model enables us to assess the adequacy of a set of requirements; if we cover the requirements but not the model—as in the case of the FGS mentioned above—it is an indication that the we have an incomplete set of requirements.

The remainder of the paper is organized as follows: Section 2 describes our proposed approach in further detail. In Section 3, we describe the experiment conducted to evaluate our approach. Results obtained and their analysis is presented in Section 4. Section 5 discusses the implications of our experimental results, and Section 6 concludes.

## 2 Approach

In model-based development it is crucial to ensure that the model is correct with respect to the user requirements; we want to perform model testing. Needless to say, using the model itself as a basis for this testing is not suitable. Instead, we would like to somehow use the high-level requirements on the system to derive tests for the purpose of model testing. The main idea in our approach is to automatically generate tests for model validation directly from formalized requirements (currently formalized as Linear Temporal Logic–LTL properties).

Previously we developed an approach and tool support to generate test-cases to provide coverage of the requirements (as opposed to the model) [17]. This approach, however, uses the MUT as a basis for the test case generation. In short, our tool uses the model to find execution traces that demonstrate that the requirements are met (up to some predefined coverage of the requirements, for example, Unique First Cause coverage). Although this approach will help us find test cases, the tests are derived from the model itself and we are explicitly searching for execution traces of the model that satisfy the requirements; if there is at least one execution trace that would satisfy the requirement we will find that as a test case. Therefore, our earlier approach is useful to *illustrate how* a model can satisfy its requirements, but it is not suitable to investigate *whether or not* the model satisfies its requirements. To address the latter issue it is desirable to somehow generate test cases directly from the requirements without referring to the behavior of the MUT.

To achieve this goal, we alter our previous approach so

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

$$G((\neg Onside\_FD\_On \wedge \neg Is\_AP\_Engaged) \rightarrow X(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)) \quad \text{(b)}$$

**Table 1. (a) Sample high-level requirement on the FGS (b) LTL property for the requirement**

that it uses an abstract model derived only from the requirements without any information about the behavior of the MUT—we call this abstract model the *Requirements Model*. In the next section we provide additional relevant background. We discuss the methods and tool support to help create the requirements model in Section 2.2.

## 2.1 Requirements-Based Testing

There is a close relationship between high-level requirements and the properties (or formalized requirements) captured for verification purposes. As an example, consider the requirement from a sample Flight Guidance System (FGS) shown in Table 1 defining how the Flight Director (FD) is turned on by the Autopilot (AP). The property states that it is globally true (G) that if the Onside FD is not on and the AP is not engaged, in the next instance in time (X) if the AP is engaged, then the Onside FD will also be on. As can be seen, the Linear Temporal Logic (LTL) property is very similar in structure to the natural language requirement and the manual translation of a large set of informal requirements was straightforward [11].

The requirements coverage metric used in this paper is the *Unique First Cause* (UFC) coverage defined in [17]. The UFC metric is adapted from the Modified Condition/Decision Coverage (MC/DC) criterion [2, 7] defined over source code. MC/DC is a structural coverage metric that is designed to demonstrate the independent effect of basic Boolean conditions (i.e., subexpressions with no logical operators) on the Boolean decision (expression) in which they occur. Since requirements captured as LTL properties define paths rather than states, we broaden our view of structural coverage to accommodate satisfying paths rather than satisfying states. We defined these satisfying test paths by extending the constraints for state-based MC/DC to include temporal operators. A test suite is said to satisfy UFC coverage over a set of LTL formulae if executing the test cases in the test suite will guarantee that:

- every basic condition in a formula has taken on all possible outcomes at least once
- each basic condition has been shown to independently affect the formula's outcome.

We defined independence in terms of the shortest satisfying path for the formula. Thus, if we have a formula $A$ and a path $\pi$, an atom $\alpha$ in $A$ is the unique first cause if, in the first state along $\pi$ in which $A$ is satisfied, it is satisfied because of atom $\alpha$. The formal definition for UFC and the obligations for LTL temporal operators is presented in [17].

Several research efforts have developed techniques for automatic generation of tests from formal models using model checkers as test case generation tools [13, 14, 5]. Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [4]. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. In short, a counter-example is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

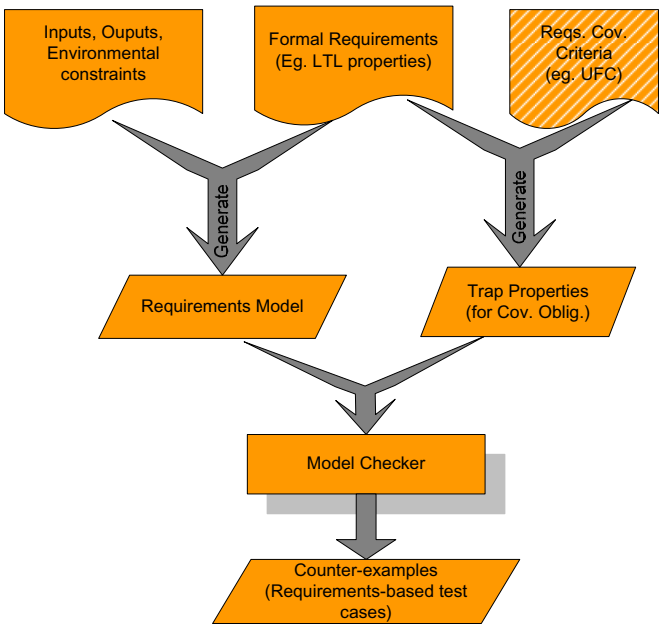


**Figure 3. Automated Requirements-Based Test Case Generation**

One way to use a model checker to find test cases is by formulating a test criterion as a verification condition for the model checker. Earlier we briefly described UFC over paths. Using this definition we can derive UFC obligations that show that a particular atomic condition affects the outcome of the property. Given these obligations and a formal model of the software system, we can now challenge the model checker to find an execution path that would satisfy one of these obligations by asserting that there is no such path (i.e., negating the obligation). We call such a formula a trap formula or trap property [5]. 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 show the UFC obligation of interest over the model. By repeating this process for all UFC obligations within the set derived from a property, we can derive UFC coverage of the property over the model. By performing this process on all requirements properties, we can derive a test suite that provides UFC coverage of the set of requirements. This process is illustrated in Figure 3.

When the model checker does not return a counterexample (or test case) for a trap property (in our case for an UFC obligation) it means that a test case for that particular test obligation does not exist. In the case of UFC obligations it implies that the atomic condition that the obligation was designed to test *does not* uniquely affect the outcome of the property. In each of these cases the original requirements property is *vacuous* [1], that is, the atomic condition is not required to prove the original property.

For reasons mentioned earlier in the section, we need to create a Requirements Model different from the MUT for requirements-based test case generation.

## 2.2 Requirements Model for Test Case Generation

The requirements model is created using the following information:

- requirements specified as invariants
- inputs, and the outputs of the MUT
- input constraints or environmental assumptions (if any)

The formalized requirements and environmental assumptions are specified as invariants in the requirements model. These invariants restrict the state space of the requirements model so that we only allow behaviors defined by the requirements. We built the requirements model in this fashion since tests derived for model validation should be based solely on the high-level requirements and the environmental assumptions, and should not be influenced by the structure and behavior of the MUT. In addition, the names of the inputs and outputs of the MUT are needed to construct concrete test cases that can be executed on the MUT.

We hypothesize that with a well defined set of requirements and environmental constraints, requirements-based tests generated from the requirements model to provide UFC coverage of the requirements will provide high MC/DC coverage of the model under test and, thus, be highly beneficial in the validation testing process. We empirically evaluate this hypothesis in Section 4 using three realistic examples from the avionics domain.

We developed a tool that allows the requirements model to be built in an automated fashion. The tool takes as inputs a formal set of requirements, environmental constraints, and the MUT. Requirements need to be formalized in the LTL notation and the environmental constraints specified as invariants. The tool starts with the MUT and strips out everything but the declaration for inputs and outputs of the MUT. To this stripped model (only containing the declarations for the inputs and outputs of the MUT), the tool automatically adds the formal LTL requirements and environmental constraints as invariants. The resulting model is the requirements model. Figure 4 illustrates the approach used in the tool to create the Requirements Model.

The tool currently supports only requirements expressed as safety properties; requirements expressed as liveness properties are not yet supported. In our work we have not found this to be a limitation since *all* requirements expressed over our case-examples can be expressed as safety properties. Converting safety properties into invariants in the requirements model is not a trivial task. The tool supports the SMV notation for the models [12]. Note that in the SMV notation, the declaration that allows us to specify a set of invariant states ("INVAR") only allows boolean expressions in its syntax. Thus for requirements defined using temporal operators, SMV will not allow them to be used directly in the "INVAR" declaration. For requirements containing the next state temporal operator, this issue can be resolved easily by defining additional variables in the requirements model. To see this, consider the following example of a naïve informal requirement:

```
"If Switch is pressed, then in the next
step the Light will turn on."
```

formalized in LTL as:

```
LTLSPEC G(Switch -> X Light)
```

The formalized requirement has the temporal next state operator $X$ in the expression within the global operator $G$. To express the above requirement as an invariant in the requirements model, we will declare a new variable, $req\_1$ to help us in the invariant definition.

```
ASSIGN
    init(req_1) := TRUE;
    next(req_1) := Switch -> next(Light);

INVAR req_1
```

As seen in the SMV assignment above, we specify the requirement as an invariant using this new variable. If the requirement does not contain any temporal operators, we do not need this additional variable. We can simply put the requirements expression in the invariant declaration (INVAR). This method of specifying invariants will not work for requirements defined using "Future" and "Until" LTL operators. Our tool does not currently support such requirements. We are investigating this issue and plan to resolve

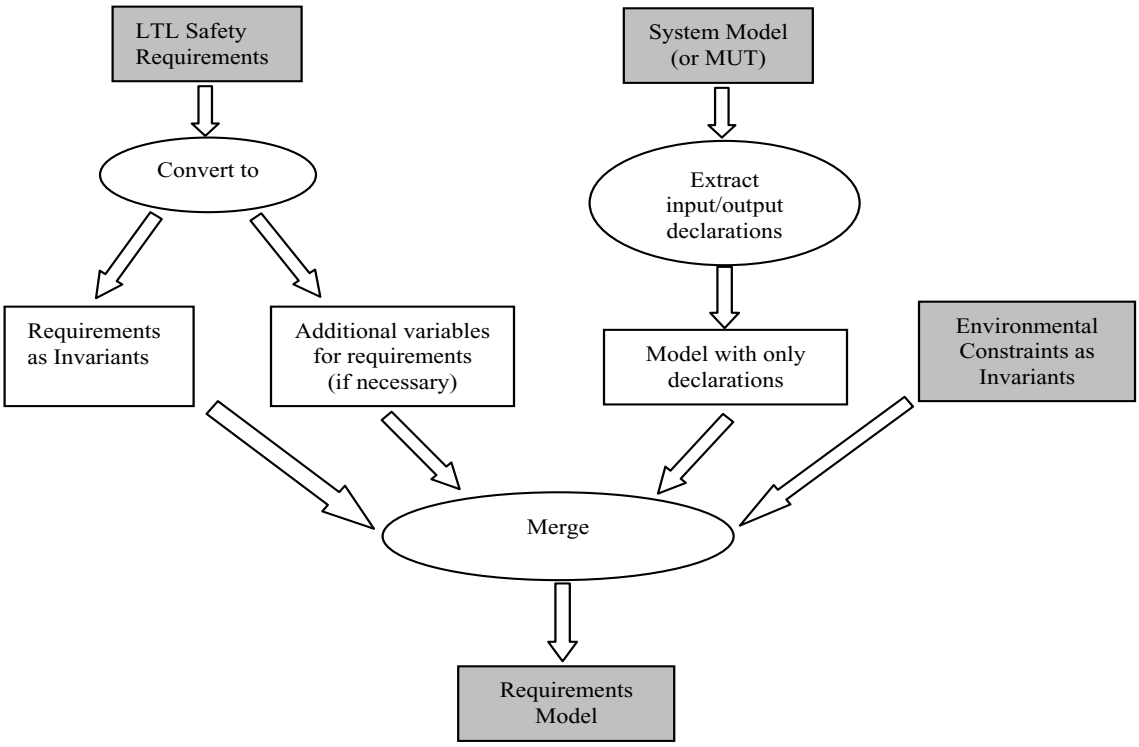


**Figure 4. Approach used in Tool that Automatically Creates the Requirements Model**

it in our future work. (As mentioned above, this restriction has not been a problem in practice since all requirements we have encountered in our case examples could be formalized as safety properties.)

## 3 Experiment

In this initial experiment we were interested in determining (1) the feasibility of generating requirements-based tests from a requirements model, and (2) the effectiveness of these test sets in validating the system model or MUT. We evaluated the effectiveness of the generated test cases using three realistic examples from the avionics domain - the FGS, and two models related to the Display Window Manager system (DWM_1, and DWM_2).

**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 this paper we focus on the mode logic of the FGS. The requirements and implementation model used in this paper are described in [10] and are similar to production systems created by Rockwell Collins Inc.

This system consists of 293 informal requirements formalized as LTL properties as well as a formal model captured in our research notation RSML$^{-e}$ [18].

**Display Window Manager Models (DWM_1 and DWM_2):** The Display Window Manager models, DWM_1 and DWM_2, represent two of the five major subsystems of the ADGS-2100 Display Window Manager (DWM), an air transport-level commercial displays system. The DWM acts as a "switchboard" for the system and has several responsibilities related to routing information to the displays and managing the location of two cursors that can be used to control applications by the pilot and copilot. The DWM must update which applications are being displayed in response to user selections of display applications, and must handle reversion in case of hardware or application failures, deciding which information is most critical and moving this information to the remaining display(s). It also must manage the cursor, ensuring that the cursor does not

appear on a display that contains an application that does not support the cursor. In the event of reversion, the DWM must ensure that the cursor is not tasked to a dead display.

The DWM_1 system consists of 43 informal requirements formalized as LTL properties. The formal model of the system was built in the Simulink notation from Mathworks, Inc [9]. The DWM_2 system consists of 85 informal requirements formalized as LTL properties. The formal model of the system was built in the Simulink notation.

### 3.1 Setup

The experiment constituted the following steps:

**Create the Requirements Model:** The requirements model as mentioned before was built using the formalized set of requirements, names of inputs and outputs of the MUT, and environmental assumptions for the system. We described the tool to build the requirements model in an automated fashion in Section 2.2. The modeling notation that we use in our tool is the SMV language.

**Generate Requirements Obligations:** We started with the set of requirements formalized as LTL properties. We generated obligations (as LTL specifications) to provide *requirements UFC coverage* over the syntax (or structure) of the LTL properties. The rules and tool to auto-generate UFC obligations is from our previous work [17].

**Generate Requirements-Based Test Cases:** We used the bounded model checker in NuSMV for automated test case generation. We generated test cases from the requirements model to provide UFC coverage over the properties. We discussed this approach in Section 2.1 previously. We ran the generated test suite on the MUT to measure the resulting model coverage achieved.

**Measure Model Coverage Achieved:** In this experiment we assessed the effectiveness of the test sets in terms of coverage achieved over the MUT since this is a major concern in our application domain. (In the future, we plan on evaluating the quality of these test sets in terms of their fault finding capability on the MUT.) To measure coverage achieved by the requirements-based test suites over the RSML$^{-e}$ model of the FGS, we leveraged tools built in a previous project [8] that allowed us to measure different kinds of coverage of the FGS model expressed in RSML$^{-e}$. To measure coverage over the DWM_1 and DWM_2 models in Simulink, we used the capabilities in our translation infrastructure [16]. The infrastructure allows us to translate the Simulink model into a synchronous language, Lustre [6]. We then measure coverage over the translated Lustre model. Using these measurement tools, we ran the test suite providing requirements UFC coverage and recorded coverage achieved over the MUT. In particular, we measured MC/DC achieved over the MUT. We chose to measure this coverage since current practices and standards [15] in the avionics domain require test suites to achieve MC/DC over the software.

## 4 Results and Analysis

Table 2 shows the coverage achieved by the requirements-based tests over the MUT for the three systems. The requirements based tests did poorly on the FGS covering a mere 41% of the model. On the other hand, the tests did well on the DWM_1 and DWM_2 systems covering more than 95% and 92% of the MUT respectively. Our findings and analysis for the systems are summarized in the remainder of this section.

### 4.1 FGS

As seen from Table 2, we generated 887 requirements UFC obligations for the FGS requirements, of which 835 resulted in test cases. The time expended in test generation was 47 mins using the NuSMV bounded model checker. For the remaining 52 obligations that did not result in test cases, the UFC trap property is *valid*, which means that the condition that it is designed to test *does not* uniquely affect the outcome of the property and, thus, no test case demonstrating this independent effect exists. The inability to find a test case may be an indication of a poorly written requirement or an inconsistent set of requirements. In this paper, we did not attempt to correct these properties, we decided to use the property set "as-is" as a representative set of requirements that might be provided before a model is constructed.

Table 2 shows that the test suite generated to provide UFC over the FGS requirements provides only 41% MC/DC coverage over the MUT. To explain the poor coverage, we took a closer look at the requirements set for the FGS and found that the poor performance of the generated test suite was due in part to the structure of the requirements defined for the FGS. Consider the requirement

*"When the FGS is in independent mode, it shall be active"*

This was formalized as a property as follows:

$$\mathrm{G}(m\_Independent\_Mode\_Condition.result \rightarrow \mathrm{X}(Is\_This\_Side\_Active = 1))$$

Note here that the condition determining if the FGS is to be in independent mode is abstracted to a macro (Boolean function) returning a result (the .result on the left hand side

| | # Obligations | # Tests Generated | Time Expended | MC/DC Achieved |
|---|---|---|---|---|
| **FGS** | 887 | 835 | 47 mins | 41.7% |
| **DWM_1** | 129 | 128 | $<$ 1 min | 95.3% |
| **DWM_2** | 335 | 325 | $<$ 2 mins | 92.6% |

**Table 2. Summary of Requirements-Based Tests Generated and Coverage Achieved**

of the implication). Many requirements for the FGS were of that general structure.

$$\mathrm{G}(Macro\_name.result \rightarrow \mathrm{X}\ b)$$

The definition of the macro resides in the MUT and is missing from the property set. Therefore, when we perform UFC over this property structure, we do not perform UFC over the—potentially very complex—condition making up the definition of the macro.

The macro $Independent\_Mode\_Condition$ is defined in RSML$^{-e}$ as:

```
MACRO Independent_Mode_Condition():
 TABLE
   Is_LAPPR_Active           : T *;
   Is_VAPPR_Active           : T *;
   Is_Offside_LAPPR_Active   : T *;
   Is_Offside_VAPPR_Active   : T *;
   Is_VGA_Active             : * T;
   Is_Offside_VGA_Active     : * T;
 END TABLE
END MACRO
```

(The table in the macro definition is interpreted as a Boolean expression in disjunctive normal form; each column in the table represents one disjunction; a * indicates that in this disjunction the condition on that row is a don't care.) Since the structure of $Independent\_Mode\_Condition$ is not captured in the required property, the test cases generated to cover the property will not be required to exercise the conditions making up the definition of the macro. We will thus most likely only cover one of the UFC cases needed to adequately cover the macro.

Note that this problem is not related to the method we have presented in this paper; rather, the problem lies with the original formalization of the FGS requirements as LTL properties. Properties should not be stated using internal variables, functions, or macros of the MUT; doing so leads to a level of circular reasoning (using concepts defined in the model to state properties of the model). If a property must be stated using an internal variable (or macro) then additional requirements (properties) are needed to define the behavior of the internal variable in terms of inputs to the system. For the FGS, a collection of additional requirements defining the proper values of all macro definitions should be captured. These additional requirements would necessitate the generation of more test cases to achieve requirements UFC coverage and we would presumably get significantly better coverage of the MUT.

Thus, for the FGS, our approach helped identify inadequacies in the requirements set by measuring coverage achieved on the MUT with the generated requirements-based tests. Rockwell Collins Inc. was aware of the problems related to defining requirements using internal variables from the model, learned from the FGS research model, and rectified this problem in later modeling efforts. The DWM models presented in the next section are two such models with well defined requirements.

## 4.2 DWM_1 and DWM_2

In contrast to the FGS, in the DWM models all the requirements were defined completely in terms of inputs and outputs of the system. Internal variables—if any—used when describing requirements were defined with additional requirements. The problems seen with the requirements of the FGS were not present in the DWM examples. As seen in Table 2, our test case generation approach was feasible on both the DWM_1 and DWM_2 systems. For the DWM_1 system, we generated 128 requirements-based tests from 43 formalized requirements in less than one minute. The generated requirements-based tests covered more than 95% of the MUT. On the DWM_2 system, we generated 325 test cases from 85 requirements in less than two minutes. The generated tests covered more than 92% of the MUT. Note that 10 of the 335 UFC obligations on the DWM_2 system did not generate test cases. This implies that the atomic conditions that these obligations were designed to test were vacuous in the requirement. It is evident from these results that in both the systems, the requirements-based tests cover most of the behavior in the MUT and therefore have the potential to be effective in model validation.

In our experiments we observed that for some of the generated requirements-based tests the outputs predicted by the requirements model differed from the outputs generated when the tests were executed on the MUT. This occurs because the MUT may define constraints not in the requirements model, constraints that cause the test cases to lead to different outputs. To illustrate, consider the naïve example of a requirement and one of its UFC obligation in Table 3. Let us suppose $a$ is an input and $b$ an output of the example system. The UFC obligation states that $(a \rightarrow Xb)$ is

*"If a is true then in the next step b will be true"*
(1)

$$G(a \rightarrow Xb)$$
(2)

$$(a \rightarrow Xb)\ U\ ((a\ \&\ Xb)\ \&\ G(a \rightarrow Xb))$$
(3)

**Table 3. (1) Example high-level requirement (2) LTL property for the requirement (3) UFC obligation for atomic condition b**

true until you reach a state where $a$ is true and in the next state $b$ is true, and the requirement continues to hold thereafter. This obligation would ensure that $b$ is necessary for the satisfaction of the requirement.

For illustration purposes, let us suppose the MUT is built in such a way that it imposes an additional constraint: `"Output b is always true"`; this model would still satisfy the requirement. Table 4 shows a requirements-based test case predicting a certain behavior through the requirements model but when executed through the MUT we get a different (but still correct) behavior. The test case results in different values for output $b$ between the requirements model and the MUT because of the the additional constraint imposed by the MUT.

**Requirements Model**

| Step | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| a | 0 | 0 | 1 | 0 |
| b | 0 | 0 | 0 | 1 |

**MUT**

| Step | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| a | 0 | 0 | 1 | 0 |
| b | 1 | 1 | 1 | 1 |

**Table 4. Sample Test Execution through Requirements Model and MUT**

Observing such differences may help developers in validating the additional constraint imposed by the MUT. Note again that differences in test results predicted by the requirements model and the actual results from the MUT do not imply that the MUT is incorrect. The MUT may be correct but simply more restrictive. Our approach does not address this oracle problem at this time and we currently rely on the developers to make a decision of whether or not the results are acceptable. (A more extensive discussion on this issue is provided in Section 5). In our experience, additional constraints in the MUT are usually correct and needed, and they are missing from the requirements set. Thus in addition to model validation this exercise may help developers in identifying these missing requirements.

The requirements-based tests generated using our approach provides very high coverage over the MUT for the DWM_1 and DWM_2 systems owing to their well defined set of requirements. Nevertheless we did not get 100% coverage over these systems. There may be several factors contributing to this result: (1) there may be missing requirements, (2) the model is violating some of the requirements, and (3) there may be a mismatch between our definition of UFC coverage in the requirements domain and the MC/DC coverage as measured in the model domain. Presently, we have been unable to determine which of these was the contributing factor on the two systems, but we plan to investigate it in the future.

## 5 Discussion

In our experiment, we found that our proposed approach for automatically generating model validation tests is feasible but the effectiveness of the generated tests is (not unexpectedly) subject to the quality of the requirements. For an incomplete set of requirements—such as the one provided for the FGS—the requirements-based tests provide low coverage over the MUT. Note that this is a problem with the requirements and not the proposed approach. Nevertheless, for a mature and extensive set of requirements—such as those provided for the DWM models—the generated requirements-based tests provide high coverage over the MUT. Additionally, our approach has the potential to help identify missing requirements, like in the case of the FGS. When requirements-based tests providing coverage over the requirements provide poor coverage over the model it is an indication that we have an incomplete set of requirements.

When validating and analyzing models using the generated requirements-based tests, we encourage developers to carefully consider the following two issues.

First, does every requirements coverage obligation result in a test case from the requirements model? If it does not it is an indication that the requirement (property) from which the obligation is derived is poorly written. For instance, if requirements UFC coverage is used it means that the condition that the obligation is designed to test *does not* uniquely affect the outcome of the property and is thus not required to demonstrate that the property holds. Investigating this issue further would help in getting better quality requirements.

Second, do the predicted outputs match the actual outputs? Compare the predicted outputs of the requirements-based tests from the requirements model with the outputs as they are executed on the MUT. As seen in our experiment, for all three systems, the predicted outputs frequently differ from the output actually produced by the MUT. We found

that this occurred because of the additional constraints that were in the MUT but not in the requirements model. Investigating this discrepancy in outputs will help validate these additional constraints.

The oracle problem, that is, deciding whether the generated requirements-based tests pass/fail on the MUT is not handled in this paper. As discussed, this decision cannot be made by simply comparing the outputs from the MUT with the ones predicted by the requirements model. We are currently investigating techniques that will automate the oracle problem in the future. One simple solution would be to discard the expected outputs generated in the requirements-based test case generation, run the tests through the MUT and collect the execution traces, and then use the requirements and the requirements coverage obligations as monitors over the traces to determine if a violation has occurred. This would be a simple solution drawing on readily available techniques from the run-time verification community.

Finally, it is worth noting that the requirements coverage metric used plays a key role in the effectiveness of the generated requirements-based tests. For instance, if we use decision coverage of the requirements, it would require a single test case that demonstrates that the requirement is satisfied (a negative test case that demonstrates that the requirement is not met would presumably not exist). Typically, a single test case per requirement is too weak of a coverage since it is possible to derive many rather useless test cases. If we again consider our sample requirement and formalization from Table 1. We can, for example, satisfy the decision coverage metric by creating a test case that leaves the autopilot disengaged throughout the test and disregards the behavior of the flight director. Although this test case technically satisfies the requirements, it does not shed much light on the correctness of the MUT. It is therefore important to choose a rigorous requirements coverage metric, like the UFC coverage, for requirements-based test case generation.

## 6 Conclusion and Future Work

In this paper, we proposed and evaluated an approach that automates the generation of requirements-based tests for model validation. The approach uses requirements formalized as LTL properties. Test cases were generated through an abstract model, that we call the requirements model, to provide requirements UFC coverage over the properties. We evaluated our approach using three realistic examples from Rockwell Collins Inc.: the mode logic from a flight guidance system, and two models related to the display window manager system. We measured MC/DC achieved by the generated tests on the MUT. We found our approach was feasible with regard to time taken and number of test cases generated for all three systems. The generated tests achieved high coverage over the MUT for the DWM models that had a well defined set of requirements. On the other hand, the tests generated from the FGS requirements covered the MUT poorly since the FGS had an incomplete set of requirements. Based on this initial evaluation we believe our proposed approach to generating requirements-based tests provides three benefits:

1. Saves time and effort when generating test cases from requirements.
2. Effective method for generating model validation tests when the requirements are well defined.
3. Helps in identifying missing requirements and over-constrained models.

We will assess the fault finding capability of the generated requirements-based tests in our future work. We also plan to evaluate other requirements coverage criteria for requirements-based test case generation.

## 7 Acknowledgements

We would like to thank Dr. Steven Miller from Rockwell Collins Inc. for his helpful discussions and insightful comments.

## References

[1] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Formal Methods in System Design*, pages 141–162, 2001.

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

[3] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200, September 1994.

[4] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

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

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] K.J. Hayhurst, D.S. Veerhusen, and L.K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA, 2001.

[8] Mats P.E. Heimdahl and George 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.

[9] Mathworks Inc. Simulink product web site. Via the world-wide-web: http://www.mathworks.com.

[10] S. Miller, A. Tribble, T. Carlson, and E. J. Danielson. Flight guidance system requirements specification. Technical Report CR-2003-212426, NASA, June 2003.

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

[12] The NuSMV Toolset, 2005. Available at http://nusmv.irst.itc.it/.

[13] Sanjai Rayadurgam. *Automatic Test-case Generation from Formal Models of Software*. PhD thesis, University of Minnesota, November 2003.

[14] Sanjai Rayadurgam and Mats P.E. 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.

[15] RTCA. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[16] Michael Whalen. Autocoding tools interim report. In *NASA Contract NCC-01-001 Project Report*, February 2004.

[17] Michael Whalen, Ajitha Rajan, Mats Heimdahl, and Steven Miller. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2006.

[18] Michael W. Whalen. A formal semantics for $RSML^{-e}$. Master's thesis, University of Minnesota, May 2000.