Test-Sequence Generation from Formal Requirement Models

Sanjai Rayadurgam and Mats P. E. Heimdahl

Department of Computer Science and Engineering, University of Minnesota
200 Union Street S.E., 4-192, Minneapolis, MN 55455, USA
E-mail: {rsanjai,heimdahl}@cs.umn.edu

Abstract

This paper discusses a method for generating test sequences from state-based specifications. We show how a model checker can be used to automatically generate complete test sequences that will provide arbitrary structural coverage of requirements specified in a high-level language like SCR or RSML^{-e}.

We have defined a language independent formal foundation for test sequence generation using model checkers that is suitable for representing software artifacts like requirements models, software specifications, and code. This paper shows a concrete application of our formal framework for test generation in the requirements modeling domain. The framework allows one to define structural coverage criteria in terms of the formal model of a software artifact and describes how test sequences can be generated to satisfy those coverage criteria using a model-checker. The approach is illustrated using examples. We define various criteria in terms of the specification language, translate those into criteria in the formal framework, and demonstrate how we generate the test sequences.

1. Introduction

Software development for high assurance systems, such as the software controlling aeronautics applications and medical devices, is a costly and time consuming process. In such projects, the validation and verification phase (V&V) consume approximately 50%–70% of the software development resources. Although there have been breakthroughs in static V&V techniques, such as model checking and theorem proving, *testing* is still an invaluable V&V technique that *cannot* be replaced. Currently, the majority of the V&V time is devoted to the development of test cases to adequately test the required functionality of the software (black-box, requirements-based testing) as well as adequately cover the implementation (white-box, code-based testing). Thus, if the process of deriving test cases for V&V could be automated and provide requirements-based and code-based test suites that satisfy the most stringent standards for critical systems (such as, for example, DO-178B– the standard governing the development of flight-critical software for civil aviation), dramatic time and cost savings would be realized.

In this paper, we outline a *specification-centered* approach to testing where we rely on a formal model of the required software behavior for test-case generation, as well as, an oracle to determine if the implementation produced the correct output during testing. Our work is based on the hypothesis that *model checkers* can be effectively used to automatically generate test sequences that provide a predefined *structural coverage* of a formal specification. In [19], we defined a formalism suitable for representing software engineering artifacts in which various structural test coverage criteria can be defined. Here, we show how this formal foundation can be used to generate structural tests from a formal specification of the required software behavior, using a small example from the avionics domain. To illustrate the approach, we define a set of structural coverage criteria that are applicable to requirements specified in RSML^{-e} [23, 22] or a similar formal language. While the specific criteria are indeed dependent on the specification language, the formal foundation is language independent and the underlying approach is equally applicable to any other language that can be model-checked. We show how the model can be translated into the input language of a model checker like SMV and how the coverage criteria can be captured as CTL or LTL properties. Test sequences are then generated by challenging a model checker to find counter examples to the coverage criteria—such a counter example comprises a test sequence. This strategy has been used by [8] and [2].

The rest of the paper is organized as follows. Section 2 provides a short overview of related efforts. Section 3 describes our overall approach and Section 4 provides an overview of our formalism. Sections 5 and 6 introduce our case example, briefly discuss RSML^{-e}, and illustrate our test sequence generation approach. Finally, we provide a short discussion of our work and the conclusions.

2. Background and related work

Much research has gone into both model checking and software testing. The coverage in this section is by necessity cursory. We attempt to present the research efforts that are most relevant to our current work.

2.1. Testing and model checking

Model checking is a form of formal verification where a finite state representation of a proposed system can be explored exhaustively to determine if it satisfies desirable properties. The properties to be verified are cast as assertions of formulas in an appropriate temporal logic and the system behavior is specified as some form of a transition system. Temporal logic formulas typically make claims about system behavior as it evolves over time. Both temporal logics and model-checking have been active research areas for more than a decade [7, 6]. Many popular temporal logics like Computation Tree Logic (CTL and CTL*) [7] and Linear-time Temporal Logic (LTL) [18] have associated model-checking systems such as SPIN [11], SMV [15], and extensions to PVS [17, 20].

Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation. 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. We use the model checker as a test data generator by posing various test coverage criteria as challenges to the model checker. For example, we can assert to the model checker that a certain transition in a specification cannot be taken. If this transition in fact can be taken, the model checker will generate a sequence of inputs (with associated outputs) that forces the model to take this transition—we have found a test case that provides transition coverage of this transition. With this approach we will be able to automatically generate test suites for user defined levels of specification test coverage.

2.2. Test data generation

Although much work has gone into definition and evaluation of various test selection strategies [24], the ability to automate the selection of tests is limited.

To our knowledge, two other research groups are pursuing the promising approach to use model checking to generate test cases. Gargantini and Heitmeyer [8] describe a method for generating test sequences from requirements specified in the SCR notation. To derive a test sequence, a *trap property* is defined which violates some known property of the specification. In their work, they define trap properties that exercise each case in the event and condition tables available in SCR—this provides a notion of branch coverage of an SCR specification. A model-checker is then used to produce a counter-example to the trap property. The counter-example generated assigns a sequence of values to the abstract inputs and outputs of the system, thus making it a test sequence. Our work differs in that we do not tie our test sequence generation to any particular formalism (such as SCR), instead we base our generation on an underlying formalism that makes our results language independent. Also, we demonstrate how a rich collection of structural coverage criteria (both path-based and condition-based) can be easily captured in our formalism.

Ammann and Black [2, 1] combine mutation analysis with model-checking based test case generation. They define a specification based coverage metric for test suites using the ratio of the number of mutants killed by the test suite to the total number of mutants. Their test generation approach uses a model-checker to generate mutation adequate test suites. The mutants are produced by systematically applying mutation operators to both the properties specifications and the operational specification, producing respectively, both positive test cases which a correct implementation should pass, and negative test cases which a correct implementation should fail. Their work focuses on the specification and on mutation analysis. Our proposed approach takes a broader view and includes more traditional test selection criteria that are based on the structure of the software artifact being tested.

Blackburn and Busser [3] have developed the T-VEC system which uses a theorem-proving approach to test case generation. A valid system property is specified as a logical formula. In the process of proving the formula, the tool generates a test vector with specific values for the input and output. A sequence of inputs that leads the system to to the state of interest could then be obtained by explicitly constructing a formula constraining all the states in the sequence. Simon Burton at the University of York [4], also discusses the use of theorem-proving approach for generating test cases.

3. Test generation framework

General Testing Framework: Figure 1 shows an overview of our proposed *specification-centered testing* approach. During the development of a formal requirements model, the model must be extensively inspected and analyzed, as well as extensively tested and simulated (step 1 in Figure 1)—an approach we advocate in *specification-based prototyping* [21].

A set of tests are developed from the informal require-

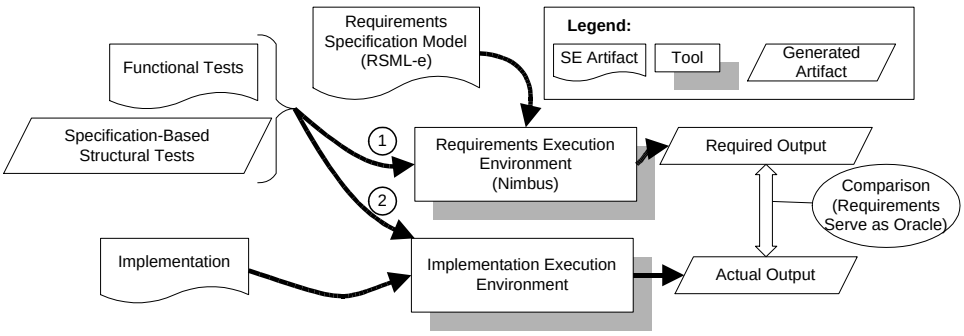

Figure 1. An overview of the specification driven testing approach.

ments to evaluate the required functionality of the model (functional tests). These functional tests may not, however, reach all the parts of the formal model—there may be transitions and conditions not exercised by these tests. The functional tests will in most cases have to be complemented by a collection of white-box tests developed specifically to exercise a specification up to a certain level of *specification coverage*. This test case generation approach is the focus of this paper. We use a model checker to automatically derive test suites providing user specified coverage of the specification (specification-based structural tests).

After we have generated test sequences from the specification, for instance, to provide basic condition coverage of the specification, all tests used during the testing of the formal specification can naturally be reused when testing the implementation in a later stage (step 2 in Figure 1). The test cases derived to test the formal specification provide the foundation for the testing of the implementation and the executable formal specification serves as an oracle during the testing of the implementation.

Finding Tests with a Model Checker: The general approach of finding test sequences using model checking techniques is outlined in Figure 2.

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, the logic CTL. 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. The model checker will now search for a counterexample demonstrating that this 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. The proposed test generation process is outlined in Figure 2.

4. Formal foundation

To provide a rigorous foundation for our work we have defined a formal framework that can be used to capture any software artifact that can be model checked, such as code or formal models. This framework serves as a formal basis for formulating various coverage criteria and for deriving trap properties. The formal model is covered in depth in [19]. Here we provide a short overview relevant to the rest of this paper. Note that we defined this framework to make our approach language independent and not tied to a specific formalism like RSML^{-e} or SCR.

We assume that the system state is uniquely determined by the value of *n* variables, $\{x_1, x_2, \ldots, x_n\}$, where each x_i takes its value from its domain D_i. Thus, the reachable state space of the system is a subset of $D = D_1 \times D_2 \times \ldots \times D_n$. The system may move from one state to another subject to the constraints imposed by its transition relation.

The transition relation is a subset of $D \times D$, specified as a Boolean predicate on the values of the variables defining the state-space. We assume that there is a transition relation for each variable, x_i, specifying how the variable may change its value as the system moves from one state to another. Informally speaking, the transition relation for x_i can be thought of as specifying three components: a set of *pre-state* values of x_i, a set of *post-state* values for x_i and the condition which *guards* when x_i may change from a pre-state value to a post-state value. We adopt the usual convention that primed versions of variables refer to their post-state values while the unprimed versions refer to their pre-state values.

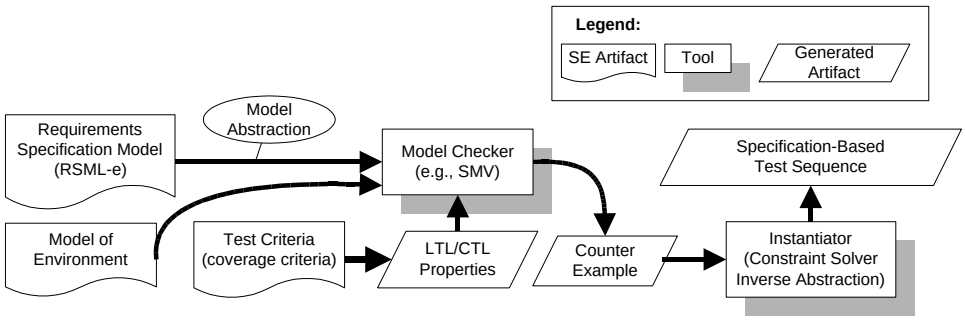

Figure 2. Test sequence generation overview and architecture.

We define $\delta_{i,j}$, the j^{th} **simple transition** of x_i, as a conjunction of a pre-state predicate $\alpha_{i,j}$ whose only parameter is the variable reference x_i, a post-state predicate $\beta_{i,j}$ whose parameters are from the set $\{x_1, ..., x_n, x'_1, ..., x'_i\}$, in which every clause includes the variable reference, x'_i, and a predicate $\gamma_{i,j}$ called the **guard** whose parameters are from the set $\{x_1, ..., x_n, x'_1, ..., x'_{i-1}\}$. Thus, $\delta_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} \wedge \gamma_{i,j}$.

The **complete transition** for a variable x_i, denoted δ_i, is the disjunction of all simple transitions for x_i. Thus, $\delta_i = \bigvee_{j=1}^{n_i} \delta_{i,j}$, where n_i is the number of simple transitions for the variable x_i.

Note that the definitions require an ordering among variables and thus eliminate circular dependencies among the post-state values of the variables.

The **transition relation** Δ, is the conjunction of the complete transitions of all the variables $x_1, ..., x_n$. Thus, $\Delta = \bigwedge_{i=1}^{n} \delta_i$.

We finally define a basic transition system M as a tuple, $M = (D, \Delta, \rho)$, where D represents the state-space of the system, Δ represents the transition relation, and ρ characterizes the initial system state. Programs expressed in common programming languages, as well as, specifications written in languages like SCR, RSML^{-e}, and Statecharts can be mapped into this basic transition system model which can then be analyzed using a model-checker like SMV.

The motivation for separating the pre- and post-state predicates and the guard is so that we can easily express both path based and condition based test coverage criteria. Path based coverage criteria will be defined over the pre- and post-state predicates; condition based criteria will be defined over the guard.

We are now ready to define the notion of a test sequence. In the following discussion, if p is a predicate and s_i and s_j are states, we use $p(s_i, s_j)$ to denote the value of the predicate p obtained by substituting the unprimed variable references with the values of the corresponding variables in state s_i and the primed variable references with the values of the corresponding variables in state s_j. If a predicate is parameterized only using unprimed variable references then we use $p(s_i)$ instead. A test case s is simply a sequence of states $\langle s_1, \ldots, s_m \rangle$, where each state is related to its successor by the transition relation Δ of the system and $\rho(s_1)$ is true, that is, $s_1$is an initial state. A test suite is a set of test cases.

We can use this formal model to define various test coverage criteria. For example, if we consider the guard on a transition as the equivalent of a decision point in a program, we can define *guard coverage* as guard taking both *true* and *false* values in the test suite and thus having an effect on whether a transition is taken. Formally, a test suite achieves simple guard coverage if for any simple transition (α, β, γ) there exist test cases s and t such that for some i and j:

1. $\alpha(s_i) \wedge \beta(s_i, s_{i+1}) \wedge \gamma(s_i, s_{i+1})$; i.e., the simple transition is taken in the transition from s_i to s_{i+1} in the test case s
2. $\alpha(t_j) \wedge \neg\beta(t_j, t_{j+1}) \wedge \neg\gamma(t_j, t_{j+1})$; i.e., the simple transition is not taken in the transition from t_j to t_{j+1} in the test case t

Incidentally, this coverage is equivalent to the notion of "branch coverage" for SCR specifications discussed by Gargantini and Heitmeyer [8]. When all transitions are explicitly specified, condition (1) alone would suffice, since in that case, not taking a given simple transition will be equivalent to taking a different simple transition.

As we will show later in this paper, this approach can be used to formalize arbitrary structural coverage criteria suitable for a software artifact. Since we will use RSML^{-e} to illustrate specification based test-case generation, the next section provides a short overview of RSML^{-e}.

5. The RSML^{-e} language

RSML^{-e} is based on the the RSML language developed by the Irvine Safety Research group [12]. RSML^{-e} was developed as a requirements modeling language specifically for embedded control systems. It is designed to be

easy to read and understand by engineers without sophisticated mathematical backgrounds, and is a hybrid graphical/tabular notation, similar to SpecTRM/RL [13] and SCR [10]. RSML^{-e} specifications consist of variables, interfaces, functions, and macros. *Variables* describe the internal state of the system. *Interfaces* describe how the specification interacts with the external environment. *Functions* and *Macros* are mechanisms for representing common expressions and predicates, respectively, in order to make specifications more concise.

There are two classes of variables in RSML^{-e}, *input* and *state* variables. *Input variables* represent inputs received from the external environment. These could correspond to monitored variables like sensor values, input from other software systems or the user interface of the system. *State variables* represent all variables whose values are computed by the RSML^{-e} specification. State variables can be grouped into states and outputs, which differ by purpose but not semantics. States describe the internal state of the model, similar to mode classes and terms in SCR [10], while outputs are computed in order to be presented to the external environment. Unlike SCR, however, it is possible to define: (1) arrays of state variables, (2) hierarchical relationships between state variables, and (3) numeric state variables.

To illustrate the different constructs, we use a small example of an altitude switch [21], illustrated in Figure 3. The altitude switch is designed to turn on a device-of-interest (DOI) in an airplane when the altitude of the airplane drops below a certain threshold.

For the altitude switch, the control software must communicate with the DOI, an altimeter, and a pilot's display that contains an inhibit and reset button as well as a "fault detected" indicator. The interfaces shield the rest of the specification from details about how the system is connected to the external environment. In this case, the reset and inhibit buttons are separately wired into the controller, along with an altimeter value and the DOI status.

The input variables: `InhibitSignal`, `Reset`, `DOIStatus`, and `Altitude` are assigned by the input interfaces as messages are received from the environment. The states are `SystemMode`, which describes the overall mode of the specification; `AltitudeStatus`, which describes where the plane is in relation to the altitude threshold; `DOI`, our model of the device of interest, and `ASWOpModes`; the operational modes for the altitude switch. There are two outputs for the model: `DOICommand` and `FaultDetected`, which describe the command to be sent to the DOI and whether or not the switch has detected any faults. This specification uses hierarchical state variables: `DOI`, `AltitudeStatus`, and `ASWOpModes` are children of `SystemMode`.

Assignment relations in RSML^{-e} determine the value of

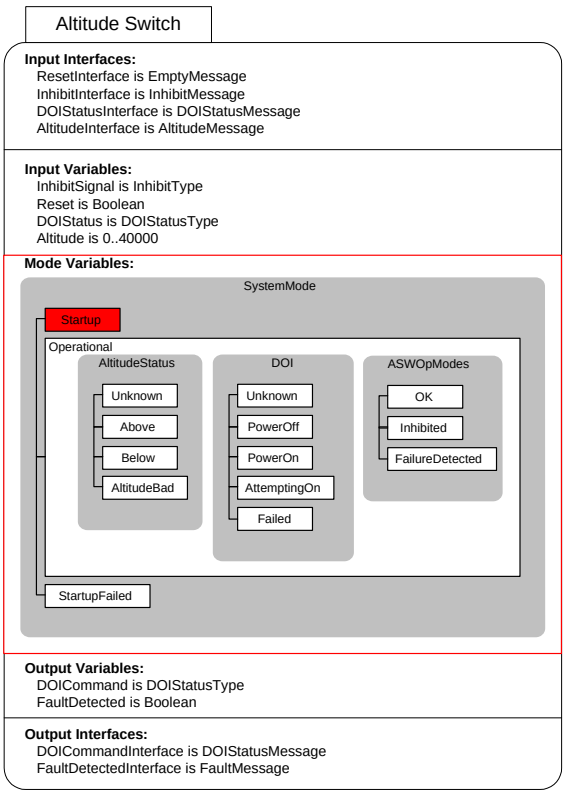

Figure 3. A graphical view of the components of the altitude switch.

state variables. They are all organized as case functions, with a *condition* describing when the case is relevant, and a *value-assignment* describing the value of the relation. For state variables, the value-assignment is an expression that describes the new value of the state variable. Mechanized procedures exist to ensure that the assignment relations are complete and consistent [9], which ensures that they are total functions.

The assignment relations are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. An object is data-dependent on any other object that it names in its assignment relation. If a state variable is a child variable of another state variable, then the child is also dependent on its parent variable. The value of each object can be computed after the items on which it is data-dependent have been computed.

:= PowerOn

Condition:

PREV_STEP(..DOI) = AttemptingOn	F	T
PREV_STEP(..DOI) EQ_ONE_OF {PowerOn, Unknown}	T	F
DOIStatus = On	T	T
AltitudeStatus = Below	T	*

Figure 4. An example condition

Conditions are simply predicate logic statement over the various states and variables in the specification. The conditions are expressed in disjunctive normal form using a notation called AND/OR tables [12]. (Figure 4 shows a portion of the transition definition for the state variable DOI—the condition under which DOI powers on.) The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated columns. An asterisk denotes "don't care."

6. Coverage criteria and test generation

In this section, we give examples of how various structural coverage criteria can be expressed as temporal logic properties and used in test sequence generation. Note that the criteria are defined to illustrate the approach only, the focus of this paper is not the definition of effective test coverage criteria; the criteria have not been evaluated for their effectiveness—this is a project currently underway. Others have done initial work in this area. For example, Offutt , *et al.* [16] discuss various criteria for deriving test cases from state-based specifications and report on their effectiveness in detecting errors and their actual coverage on an implementation of a cruise control system.

Our approach to defining test criteria is based on the structure of the specifications, analogous to structural coverage criteria defined for code. The goal is to ensure that test cases cover various constructs in the specification. Thus the criteria themselves are language dependent while the formal framework in which they are expressed could be applied to any similar specification language or code.

In the following discussion, a *test case* is to be understood as a sequence of values for the input variables in an RSML^{-e} specification. A *test suite* is simply a set of such test cases. A test case is said to assign a given value to a variable if the value appears somewhere in the sequence of values for that variable. As we briefly explained, trap properties are used to generate counter-examples using a model checker. These properties are derived from the structural coverage criteria. For the purposes of illustration, we use the altitude switch example discussed in Section 5. This example was manually translated from the RSML^{-e} language to the input language of SMV (Cadence Version)[14]. An automated translator with user definable abstraction mechanisms is under construction.

6.1. State coverage

Definition 1. *A test suite is said to achieve state coverage of a state variable in an RSML^{-e} specification, if for each possible value of the state variable there is at least one test case in the test suite that assigns that value to the given variable. The test suite achieves state coverage of the specification if it achieves state coverage for each state variable.*

Consider, for example, the state variable `DOI` in the altitude switch specification example:

```
STATE_VARIABLE DOI : { Unknown, PowerOff,
          PowerOn, AttemptingOn, Failed };
```

A test suite would achieve state coverage on `DOI`, if for each of its five different possible values, there is a test case in which `DOI` takes that value. In this example we define the criteria in terms of the post-state (β) in our formal framework. Note that a single test case might actually achieve this coverage by assigning different values to `DOI` at different points in the sequence. One could use the following LTL formulas to generate the test cases:

```
1. G ~(DOI = Unknown)
2. G ~(DOI = PowerOff)
3. G ~(DOI = PowerOn)
4. G ~(DOI = AttemptingOn)
5. G ~(DOI = Failed)
```

In each case, the property asserts that `DOI` can never have a specific value and the counter-example produced is a sequence of values for the system variables starting from an

initial state and ending in a state where DOI has the specific value. As an example, the following two-step input sequence below would exercise the AttemptingOn value of DOI. The last two lines are state variables in the specifications while the rest are input variables. The InputSource variable identifies the interface that received the input in each step. The example shows the relevant part of a trace generated by SMV as a counter-example to property (4):

```
    InputSource  -           AltitudeIface
       Altitude  -           0
AltitudeQuality  -           Good
 InihibitSignal  NoInhibit   NoInhibit
        ivReset  0           0
      DOIStatus  Off         Off
 AltitudeStatus  Unknown     Below
            DOI  Unknown     AttemptingOn
```

6.2. Condition coverage

Definition 2. *A test suite is said to achieve condition coverage of a given RSML^{-e} specification, if each guard condition (specified as either an* AND/OR *table or as a standard Boolean expression) evaluates to true at some point in some test case and evaluates to false at some point in some other test case in the test suite.*

This criterion is expressed over the guard (γ) in our formalism and is in tune with the notion that a given condition is considered covered only if both the true and false branches of the condition are taken. As an example consider the following guard condition for a transition from *any* state into Below for AltitudeStatus state variable:

```
EQUALS Below IF
 TABLE
   BelowThreshold()     : T;
   AltitudeQualityOK()  : T;
   ivReset              : F;
 END TABLE
```

Test cases to cover the condition can be generated using:

```
1. G((BelowThreshold() & AltitudeQualityOK()
     & ~ivReset) -> ~(AltitudeStatus = Below))

2. G(~(BelowThreshold() & AltitudeQualityOK()
      & ~ivReset) -> (AltitudeStatus = Below))
```

Note that the counter-examples, if any, for the two trap properties would satisfy, respectively, the two conditions for *simple guard coverage* defined in Section 4.

6.3. Column coverage

Instead of looking at the complete condition as a single unit, one could look at parts of the condition, e.g. each column in an AND/OR table and treat those as conditions to define the coverage criteria.

Consider the following that specifies a transition for the state variable ASWOpModes from *any* state to the FailureDetected state:

```
EQUALS FailureDetected IF
TABLE
  DOI IN_STATE Failed                  : T *;
  AltitudeStatus IN_STATE AltitudeBad  : * T;
  ivReset                              : F F;
END TABLE
```

The following trap properties could be used to obtain coverage of the first column:

```
1. G((DOI IN_STATE Failed & ~ivReset)
     -> ~(ASWOpModes = FailureDetected))
2. G(~(DOI IN_STATE Failed & ~ivReset)
     -> (ASWOpModes = FailureDetected))
```

Since each column of an AND/OR table typically covers one scenario when the state variable shall change value, this notion of coverage would exercise each such scenario. Whether this is a useful notion as a coverage criteria remains to be seen.

6.4. Clause-wise transition coverage

Finally, to illustrate the approach for a realistic test coverage criterion, we look at the code based coverage criterion called modified condition/decision coverage (MC/DC) and define an analogous criterion. MC/DC was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications [5]. Ideally, one should test every possible combination of values for the conditions thus achieving *multiple condition coverage*. However, the number of test cases required to achieve this grows exponentially with the number of conditions and hence becomes huge or impractical for systems with tens of conditions per decision point. 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. 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 program has been invoked at least once,
- every basic condition in a decision in the program 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.

If we think of the system as a realization of the specified transition relation, it evaluates each guard on each transition to determine which transitions are enabled and thus each guard becomes a decision point. The predicates in turn are constructed from clauses—the basic conditions.

Definition 3. *A test suite is said to achieve clause-wise transition coverage (CTC) for a given transition of a variable in an RSML^{-e} specification, if every basic Boolean condition in the transition guard is shown to independently affect the transition.*

Consider the following transition for `DOI`:

```
EQUALS PowerOn IF
 TABLE
  PREV_STEP(DOI) IN_STATE AttemptingOn        : F T;
  PREV_STEP(DOI) IN_ONE_OF {PowerOff, Unknown}: T F;
  DOIStatus = On                              : T T;
  AltitudeStatus IN_STATE Below               : T *;
  ivReset                                     : F F;
 END TABLE
```

To show that each of the basic conditions in the rows independently affects the transition, one should produce a set of test cases in which for any given row there are two test cases, such that one makes the row *true* and the other makes it *false*, the rest of the rows have the same truth values in both test cases, and in one test case the transition is taken while in the other it is not. Ideally, one would like to show that each row affects each column independently in which it is not a don't care term. For the purposes of this example, let us just consider the first column. We may generate the trap properties by examining the truth value for each row in the first column as follows:

```
0. G((~R1 & R2 & R3 & R4 &~R5) -> ~POST);
1. G(( R1 & R2 & R3 & R4 &~R5) ->  POST);
2. G((~R1 &~R2 & R3 & R4 &~R5) ->  POST);
3. G((~R1 & R2 &~R3 & R4 &~R5) ->  POST);
4. G((~R1 & R2 & R3 &~R4 &~R5) ->  POST);
5. G((~R1 & R2 & R3 & R4 & R5) ->  POST);
```

where `Ri` stands for the basic condition in the i^{th} row of the table and `POST` represents the post-state condition `DOI = PowerOn`. We systematically generated the trap-properties by first using the truth values in the column to prefix each row with the appropriate sign and then toggling one row at a time. In essence, property (0) asserts that whenever the transition condition is met the post-state condition will not be satisfied. A counter-example to this will be a test case in which the transition is taken. Each of the remaining trap-properties assert that whenever the transition condition is not satisfied because of a single row having the wrong truth value, the post-state condition will be met. Counter-examples to these, if any, will be test cases in which the transition is not taken. For example, see the test sequence below, produced by SMV, for the property (2):

```
    InputSource  -       AltitudeIface DOISI.. DOISIface
       Altitude  -       0             0       0
AltitudeQuality  -       Good          Good    Good
 InihibitSignal  NoInh.. NoInhibit     NoInh.. NoInhibit
        ivReset  0       0             0       0
      DOIStatus  Off     Off           Off     On
 AltitudeStatus  Unknown Below         Below   Below
            DOI  Unknown AttemptingOn  Failed  Failed
```

Nevertheless, on verifying these trap-properties using SMV, properties (1) and (5) did not produce any counter-examples, but were in fact shown to be true. On closer examination of property (1), it is clear that it fails to produce a counter-example because the first two rows cannot be simultaneously true. In fact, both refer to distinct partitions of the pre-state values of `DOI`. Property (5) is, however, more involved. Elsewhere in the specification, `AltitudeStatus` is forced to take the value `Unknown` when `ivReset` is true; therefore, `R4` and `R5` cannot both be true at the same time. As a side-effect of generating these test cases, dependencies among the rows of the table were discovered. This could potentially point out errors in the specifications, though in this case there were none. The remaining properties produced counter-examples providing coverage of the first column of the table.

For this example, the execution time of the model-checker was less than a second for each trap-property. In cases, where the trap-property did not produce a counter-example, the property had a contradiction in the antecedent of a conditional expression, which led the model-checker to immediately report that the property was true.

7. Discussion and conclusion

We have demonstrated an approach to automate test-case generation for software engineering artifacts such as formal specifications or code. We use structural coverage criteria to define what test cases are needed. The test cases are then generated using the power of a model checker to generate counter-examples. We briefly discussed our formal foundation, and illustrated the process with an example. The mapping from the specification language to the underlying formalism, though not discussed in detail here, can be formally defined and an effort to automate this translation is underway. Our approach is to cast the criteria in terms of the underlying model, thus providing a uniform set of coverage criteria for a variety of specifications written in high-level specification languages. Initial results indicate that the approach could scale to larger systems, provided some issues discussed below are addressed. Th approach has the potential to significantly reduce costs associated with generating test cases to high levels of structural coverage required for high assurance software.

There are, several challenges that need to be addressed. Although the approach has worked well for us this far, state space explosion is a major concern when model-checking

software artifacts and we expect to encounter this problem in the future. Often, specifications of software systems written in higher-level languages, such as SCR and RSML^{-e}, would include huge or infinite state-spaces that cannot be handled by model-checkers. The translation to a model checker must then abstract away enough detail to make model checking a feasible approach to test-case generation. Nevertheless, abstraction techniques that reduce the state-space could make instantiating the test case with concrete data a non-trivial task since the counter example would be presented in the abstract domain. These are active areas of research. Further, for test case generation, we are typically attempting to verify a property that does not hold in the model. The complete state-space need not be explored for generating a counter-example. This fact is at times exploited when model-checkers are primarily used for finding property violations rather than verification. In the cases where a counter-example cannot be found (a transition that cannot be taken, MC/DC conditions that cannot be satisfied, etc.), the verification run can be terminated and the property tagged for later manual analysis. We have not encountered this situation to date.

A goal in testing is to have a limited number of test cases without sacrificing test efficacy. In other words, one would prefer as small a test-suite as possible for a given test criteria. When generating test cases using our proposed method, a shorter sequence could be subsumed by a longer one. In general, two strategies can be adopted in reducing the test-suite size. One may pre-process by combining trap properties for different test cases. For example, with MC/DC coverage, a condition may generate several redundant trap properties. One could compare trap properties and eliminate duplicates. The other approach is to post-process the results. As soon as a test-case is generated one could execute the test sequence, measure the actual coverage on the artifact, and then remove all covered areas from further consideration. We could understand how this works in terms of program control flow. If a path to a decision point includes another decision point, a test case that exercises predicates in the former might also exercise certain predicates in the latter. Thus when generating test cases for the former, one could identify the tests that need not be generated for the latter. The task in specification-centered testing, however, is to find an appropriate order of generating test cases so that longer test sequences are constructed before shorter ones.

References

[1] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, Nov. 1999.

[2] 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.

[3] 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.

[4] S. Burton, J. Clark, and J. McDermid. Testing, proof and automation. An integrated approach. In *Proceedings of First International Workshop on Automated Program Analysis, Testing and Verification*, June 2000.

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

[6] 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.

[7] A. S. E. M. Clarke, E.A. Emerson. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, pages 244–263, April 1986.

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

[9] 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.

[10] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[11] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[12] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[13] N. G. Leveson, M. P. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.

[14] K. L. McMillan. Symbolic Model Verifer (SMV) - Cadence Berkeley Laboratories Version. Available at http://www-cad.eecs.berkeley.edu/~kenmcmil/smv.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[16] 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.

[17] S. Owre, N. Shankar, and J. Rushby. *The PVS Specification Language*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta relese edition, April 1993.

[18] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[19] 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.

[20] J. Rushby. Model checking and other ways of automating formal methods. position paper for panel on Model Checking for Concurrent Programs; Software Quality Week, May/June 1995.

[21] J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[22] J. M. Thompson, M. W. Whalen, and M. P. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.

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

[24] H. Zhu, P. Hall, and J. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.