Autom Software Eng (2007) 14:37–57
DOI 10.1007/s10515-006-0004-y

On the effect of test-suite reduction on automatically generated model-based tests

Mats P. E. Heimdahl · Devaraj George

Published online: 9 February 2007

Abstract Model checking techniques can be successfully employed as a test-case generation technique to generate tests from formal models. The number of tests-cases produced, however, is typically large for complex coverage criteria such as MC/DC. Test-suite reduction can provide us with a smaller set of test-cases that preserve the original coverage—often a dramatically smaller set. Nevertheless, one potential drawback with test-suite reduction is that this might affect the quality of the test-suite in terms of fault finding. Previous empirical studies provide conflicting evidence on this issue. To further investigate the problem and determine its effect when testing implementations derived from formal models of software we performed an experiment using a large case example of a Flight Guidance System, generated reduced test-suites for a variety of structural coverage criteria while preserving coverage, and recorded their fault finding effectiveness. Our results indicate that the size of the specification based test-suites can be dramatically reduced and that the fault detection of the reduced test-suites is adversely affected. In this report we describe our experiment, analyze the results, and discuss the implications for testing based on formal specifications.

Keywords Specification-based testing · Test reduction · Fault finding · Model checkers · Automated test generation

1 Introduction

In model-based development, the development effort is centered around a formal description of the proposed software system. The main idea behind model-based

This work has been partially supported by NASA grant NAG-1-224 and NASA contract NCC-01001. We also want to thank the McKnight Foundation for their generous support over the years.

M. P. E. Heimdahl (✉) · D. George
Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., 4-192, Minneapolis, MN 55455, USA

development is that through manual inspections, formal verification, and simulation and testing we demonstrate to ourselves (and to any regulatory agencies) that the software specification possesses desired properties. The implementation is then automatically generated from this specification and, in theory, little or no additional testing of the implementation is required.

With the use of formal models comes the ability to automatically generate specification based tests from the models. This capability may be used to generate large numbers of tests to use as *conformance tests* to provide assurance that the generated code is correct with respect to the specification from which it was generated. This type of conformance testing will most likely be required since it is unlikely that regulatory agencies will fully trust a complex code generation tool in the foreseeable future. For example, we may generate test-suites that provide MC/DC coverage (Chilenski and Miller, 1994a; RTCA, 1992) of the formal model, execute the tests on the generated code, and show that the specification and code behave equivalently for this test-suite.

The cost of generating, executing, storing, and maintaining these test-suites can be reduced through *test-suite reduction techniques*. Test-suite reduction aims to remove (or not generate at all) test-cases from a test-suite in such a way that "redundant" test-cases are eliminated. For example, a reduced test-suite T_R may provide the same structural coverage as a test-suite T with significantly fewer test-cases. Previous studies conducted on C code have shown that test-suite reduction techniques significantly reduce the number of test-cases in a test-suite while maintaining the structural coverage of the original suite (Wong et al., 1997, 1998; Rothermel et al., 1998; Jones and Harrold, 2003). The effect on the *fault finding* capability of the reduced test-suites is, however, unclear and the studies show conflicting evidence. Wong et al. (1997, 1998) found no significant effect in fault finding ability between the full suites and the reduced suites. On the other hand, Rothermel et al. (1998) and Jones and Harrold (2003) showed that the reduced test-suites can be dramatically worse with respect to fault finding.

To investigate the effect of test-suite reduction in the domain of automatically generated conformance test-suites, we conducted an experiment where we compared the test-suite size and fault finding capability of reduced test-suites generated to six different test-adequacy criteria suggested in the literature. As a case study we used a production sized model of a Flight Guidance System (FGS) provided by Rockwell Collins Inc. as a basis for test-case generation. As target implementation for our testing effort we used mutated models where we seeded "representative" faults; faults we deemed likely to be introduced when implementing a system from the formal model. The results presented in this reports constitute an expanded description of our findings presented in a paper at the Automated Software Engineering conference (Heimdahl and Devaraj, 2004).

Our results show that one can dramatically reduce our automatically generated conformance test-suites while maintaining desired coverage of the model. We also found that the fault finding of these reduced test-suites was adversely affected, and that the reduction is quite significant in the domain of specification based testing. Although further studies are needed, the results indicate that test-suite reduction may not be an effective means of reducing testing time—the reduction in the number of faults found may be unacceptable.

In the remainder of the paper we review relevant literature, describe our experimental set up, results obtained, and draw conclusions from the results.

2 Background

To put our current work in context it is necessary to provide information regarding related studies as well as the domain in which we performed our work. We will briefly discuss the approach to testing made possible when working with formal models and automatic test case generators. We will then cover the most closely related test-suite reduction experiments and contrast them with the study presented in this report.

2.1 Model-based development

As mentioned in the introduction, in the embedded systems community, there is a trend towards *model-based* (or specification based) development. In model-based development, the development effort is centered around a formal description of the proposed software system. For validation and verification purposes, this *formal specification* can then be subjected to various types of analysis, for example, completeness and consistency analysis (Heimdahl and Leveson, 1996; Heitmeyer et al., 1996) model checking (Grumberg and Long, 1994; Chan et al., 1998; Choi and Heimdahl, 2002; Heitmeyer et al., 1998; Clarke et al., 1999), theorem proving (Archer et al., 1998), Bensalem et al.(1999), and test case generation (Callahan et al., 1996; Gargantini and Heitmeyer, 1999; Engels et al., 1997; Blackburn et al., 1997; Offutt et al., 1999; Jasper et al., 1994; Rayadurgam and Heimdahl, 2001a). Through manual inspections, formal verification, and simulation and testing we demonstrate that the software specification possesses desired properties. The implementation is then—ideally—automatically generated from this specification. Naturally, manual design and coding of the implementation is also widely practiced. There are several commercial and research tools that provide these capabilities. Commercial tools are, for example, Simulink from the Mathworks (http://www.mathworks.com, SCADE Suite from Esterel Technologies (Technologies, 2004) and Statemate from i-Logix (Harel et al., 1990). Examples of research tools are SCR (Heitmeyer et al., 1995), RSML^{-e} (Thompson and Heimdahl, 1999), and Ptolemy (Lee, 2003).

The capabilities of model-based development allow us to follow a process outlined in Fig. 1. The testing effort has in this process been largely moved from unit testing of the code to functional testing of the formal model. In addition, there is a need to perform conformance testing to assure that the derived (either automatically generated or manually coded) implementation is behaviorally equivalent to the specification—a task that lends itself to automatic test-case generation from the formal specification. Test-suites generated for conformance testing are the focus of our study.

2.2 Test-cases and model checkers

Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation (Clarke et al., 1999). Should a property violation be detected, the model checker will

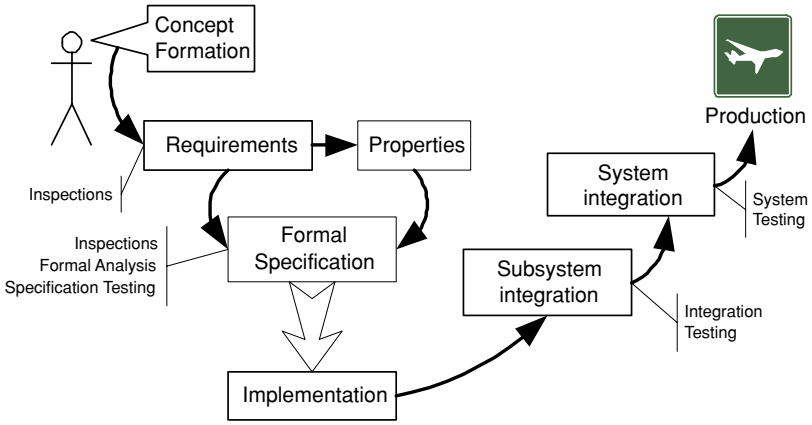

Fig. 1 Specification centered development process

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.

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 (Pnueli, 1986). 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* (Gargantini and Heitmeyer, 1999). 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, such as all state variables have taken on every value, and all decisions in the model have evaluated to both true and false, etc.

Several research groups are actively pursuing model checking techniques as a means for test-case generation. In our previous work in the Critical Systems Research Group at the University of Minnesota we provided a formalism suitable for structural test-case generation using model checkers (Rayadurgam and Heimdahl, 2001a) and illustrated how this approach can be applied to a formal specification language (Rayadurgam and Heimdahl, 2001b). We also presented a framework for specification centered testing in Heimdahl et al. (2001). This is the framework we have used in several experiments (Heimdahl et al., 2003; Heimdahl and Devaraj, 2004), including the one described in this report.

Gargantini and Heitmeyer (1999) 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.

Ammann and Black (1999) and Ammann et al. (1998), 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.

Hong et al. (2002) formulate a theoretical framework for using temporal logic to specify data-flow test coverage criteria. They also discuss various techniques for reducing the size of the test set generated by the model checker (Hong et al., 2003).

2.3 Previous test-reduction experiments

Several studies have investigated the effect of test-set reduction on the size and fault finding capability of a test-set. In an early study, Wong et al. address the question of the effect on fault detection of reducing the size of a test set while holding coverage constant (Wong et al., 1997, 1998). Their experiments were carried out over a set of commonly used UNIX utilities implemented in C. These programs were manually seeded with faults, producing variant programs each of which contained a single fault. They randomly generated a large collection of test sets that achieved block and all-uses data flow coverage for each subject program. For each test set they created a minimal subset that preserved the coverage of the original set. They then compared the fault finding capability of the reduced test-set to that of the original set. Their data shows that test minimization keeping coverage constant results in little or no reduction in its fault detection effectiveness. This observation leads to the conclusion that test cases that do not contribute to additional coverage are likely to be ineffective in detecting additional faults (of the fault types injected).

To confirm or refute the results in the Wong study, Rothermel et al. performed a similar experiment using seven sets of C programs with manually seeded faults (Rothermel et al., 1998). For their experiment they used edge-coverage (Frankl and Weiss, 1991) adequate test suites containing redundant tests and compared the fault finding of the reduced sets to the full test sets. In this experiment, they found that (1) the fault-finding capability was significantly compromised when the test-sets were reduced and (2) there was little correlation between test-set size and fault finding capability. The results of the Rothermel study were also observed by Jones and Harrold in a similar experiment (Jones and Harrold, 2003).

These radically different results are difficult to reconcile and the relationship between coverage criteria, test-suite size, and fault finding capability clearly needs more study.

In the experiment discussed in this paper we attempt to shed some additional light on this issue. Our work is different in some respects, however. First, we are not studying testing of traditional programs, we are interested in test-case generation from formal models and conformance testing of implementations derived from formal specifications. In particular, formal specifications expressed in synchronous data-flow languages commonly used in model-based development, for example, Simulink Mathworks Inc., http://www.mathworks.com, SCADE (Technologies, 2004), and Statecharts (Harel et al., 1990).

Second, we are addressing a wide spectrum of coverage criteria ranging from the very weak, for example, transition coverage, to the very strong, for example MC/DC. The previous experiments addressed either rather weak criteria such as block-coverage (Wong et al., 1998) or used test-suites that did not fully provide the desired strong coverage (Jones and Harrold, 2003). This issue will be further addressed in the discussion of our results.

These differences make a direct comparison of our results with related work difficult, but our findings seem to reinforce the observations in the Rothermel et al., and Jones and Harrold studies; although test-suite reduction can dramatically reduce the size of a test-suite without affecting coverage, test-suite reduction has a detrimental effect on the test-suite's fault finding capability.

2.4 Case example: The FGS

In a related project we have worked with Rockwell Collins Inc. on research challenges in model-based development and verification of large models in the avionics domain. In the course of that project, Rockwell Collins Inc. has developed several large formal models. To provide realistic results for the experiment described in this report, we used one of these models—a close to production model of a Flight Guidance System (FGS).[1] Engineers at Rockwell Collins Inc. were the primary developers of the model using the RSML^{-e} notation and the NIMBUS tool (a description follows in Section 2.5).

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) in a commercial aircraft. The FGS was developed using the RSML^{-e} language. 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 can be broken down to 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 case study we have used the mode logic.

Figure 2 illustrates a graphical view of the FGS state variables in the NIMBUS environment. The definitions of when and how the state variables change value are described in the RSML^{-e} tabular notation (briefly discussed in Section 2.5). 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,

[1] We thank Dr. Steve Miller, Dr. Alan Tribble, and Dr. Mike Whalen of Rockwell Collins Inc. for the information on flight control systems and for letting us use the models they developed.

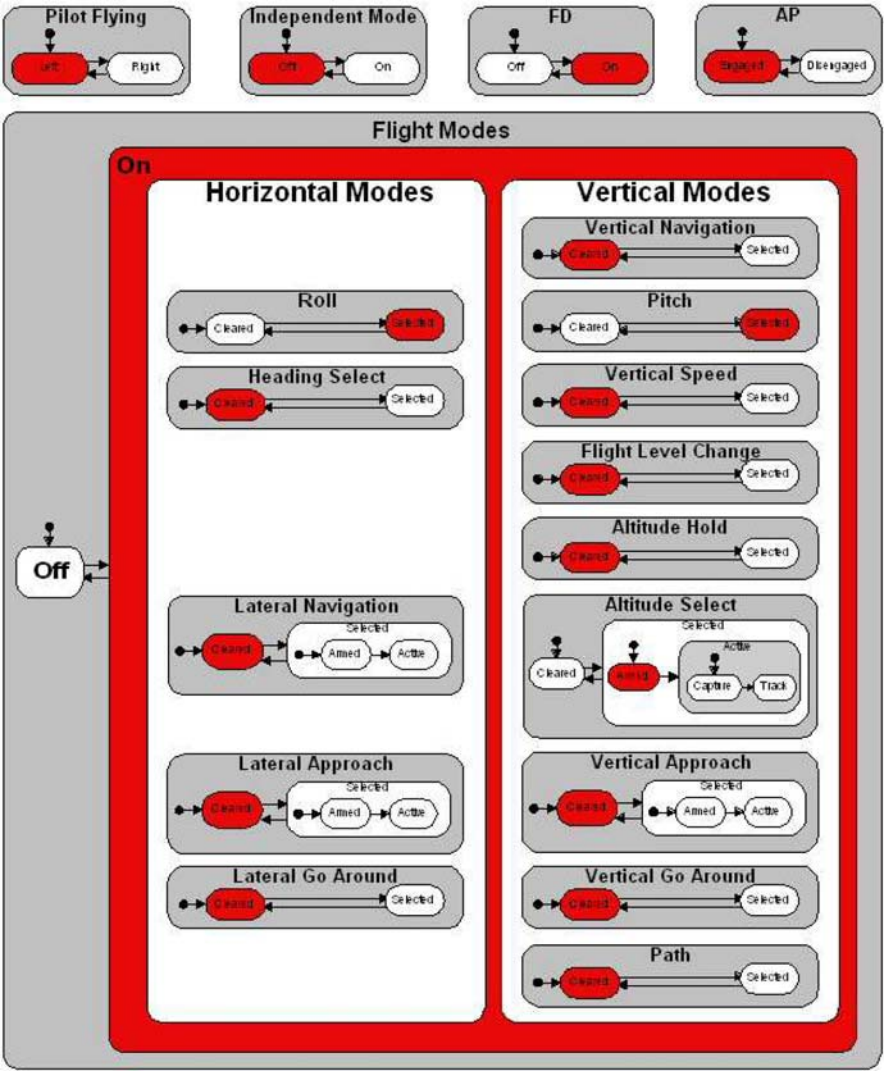

Fig. 2 Flight guidance system

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.

The FGS mode-logic model we have used in the experiment is production sized, but does not represent any actual Rockwell Collins product. The model consists of 2564 lines of code in RSML^{-e} and consists of 142 state variables. When automatically translated to the input language for the model checker NuSMV (NuS, 2005) it consists of 2902 lines of code and required 849 BDD variables for encoding (the BDD variables are Boolean variables and several may be needed to encode each enumerated variables in the model). The FGS is ideally suited for test case generation using model checkers

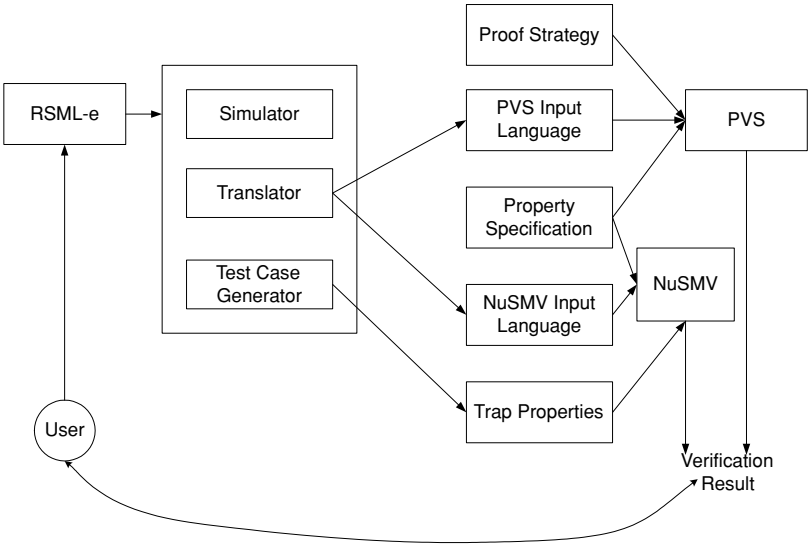

Fig. 3 Verification framework

since it is discrete—the mode logic consists entirely of enumerated and Boolean variables.

2.5 Nimbus and RSML^{-e}

Figure 3 shows an overview of the NIMBUS tools framework we have used as a basis for our test-case generation engine. The user builds a behavioral model of the system in the fully formal and executable specification language RSML^{-e} (see below). After evaluating the functionality and behavioral correctness of the specification using the NIMBUS simulator, users can translate the specifications to the PVS (Owre et al., 1993) or NuSMV input languages for verification (or test-case generation as is the case in this report). The set of LTL trap properties required to use NuSMV to generate test sequences are obtained by traversing the abstract syntax tree in NIMBUS and then outputting sets of properties whose counterexamples will provide the correct coverage.

To generate test cases in NIMBUS, the user would invoke the following steps. First, the model is automatically translated to the input language of NuSMV and the tools automatically generate the set of trap properties of interest to achieve desired coverage. Second, the trap properties are automatically merged with the NuSMV model and the NuSMV model checker is invoked to collect the counterexamples. Third, the counterexamples are processed to extract test sequences in a generic intermediate test representation. The intermediate test representation contains (1) the input in each step, (2) the expected state changes (to state variables internal to the RSML^{-e} model), and (3) the expected outputs (if any). Finally, the intermediate test representation is translated to the input format for whatever testing tool is used to test the system under test.

```
STATE_VARIABLE ROLL : Base_State
   PARENT           : Modes.On
   INITIAL_VALUE    : UNDEFINED
   CLASSIFICATION   : State

   TRANSITION UNDEFINED TO Cleared  IF NOT Select_ROLL()
   TRANSITION UNDEFINED TO Selected IF Select_ROLL()
   TRANSITION Cleared   TO Selected IF Select_ROLL()
   TRANSITION Selected  TO Cleared  IF Deselect_ROLL()

END STATE_VARIABLE


MACRO Select_ROLL() :
   TABLE
      Is_No_Nonbasic_Lateral_Mode_Active() : T;
      Modes = On                           : T;
   END TABLE
END MACRO

MACRO Deselect_ROLL() :
   TABLE
      When_Nonbasic_Lateral_Mode_Activated() : T *;
      When(Modes = Off)                      : * T;
    END TABLE
END MACRO
```

Fig. 4 A small portion of the FGS specification in RSML^{-e}

The NIMBUS tools discussed above all operate on the RSML^{-e} notation—RSML^{-e} is based on the Statecharts (Harel, 1987) like language Requirements State Machine Language (RSML) (Leveson et al., 1994). RSML^{-e} is a fully formal and synchronous data-flow language without any internal broadcast events (the absence of events is indicated by the^{-e}).

An RSML^{-e} specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants. *Input variables* are used to record the values observed in the environment. *State variables* are organized in a hierarchical fashion and are used to model various states of the control model. *Interfaces* act as communication gateways to the external environment. *Functions and macros* encapsulate computations providing increased readability and ease of use.

Figure 4 shows a specification fragment of an RSML^{-e} specification of the Flight Guidance System.[2] The figure shows the definition of a state variable, `ROLL`. `ROLL` is the default lateral mode in the FGS mode logic.

The conditions under which the state variable changes value are defined in the `TRANSITION` clauses in the definition. The condition tables are encoded in the

[2] We use here the ASCII version of RSML^{-e} since it is much more compact than the typeset version.

macros, `Select_ROLL` and `Deselect_ROLL`. The tables are adopted from the original RSML notation—each column of truth values represents a conjunction of the propositions in the leftmost column (F represents the negation of the proposition and a '$*$' represents a "don't care" condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form.

3 The experiment

To investigate the relationship between test reduction and fault finding capability in the domain of model-based tests, we designed our experiment to test two hypotheses:

Hypothesis 1. Test reduction of a naïvely generated specification based test-set can produce significant savings in terms of test-set size.

Hypothesis 2. Test reduction will adversely affect the fault finding capability of the resulting test set.

We formulated our hypotheses based on two informal observations. First, in a previous study we got an indication that one could achieve equivalent transition and state coverage with approximately 10% of the full test-set generated (Heimdahl et al., 2003), we believe this generalizes to other criteria as well. (A discussion of the various specification coverage criteria will follow in Section section:criteria below.) Second, intuitively, more tests-cases ought to reveal more faults. Only an extraordinarily good test adequacy criterion would provide a fault finding capability that is immune to variations in test-suite size, and we speculate that none of the known coverage criteria posses this property.

3.1 Experimental setup

In our experiment the aim was to determine how well a test-suite generated to provide a certain structural or condition based coverage of a specification model reveals faults in the implementation of that model as compared to a reduced test-suite providing the same coverage of the original model. To provide realistic results, we conducted the experiment using the close to production model of the Flight Guidance System (FGS) discussed in Section 2.4.

We conducted the experiment through the five the steps outlined below. Each step is elaborated in detail in the sections following this short overview.

1. We used the original FGS specification to generate test-suites to various coverage criteria of interest, for example, transition coverage or MC/DC. Note here that we did this naïvely in that we generated a test-case *for each* construct we needed to cover. Thus, the test-suites were straight forward to generate and easily traceable back to the constructs of the model they were intended to cover, but the tests in each test-suite were also highly redundant in terms of the coverage they provided.

2. To provide targets for our conformance testing experiment, we used fault seeded models to emulate the actual implementation. We generated 100 faulty models of the FGS by randomly seeding one fault per faulty specification. The fault classes we seeded were derived based on the change history of the model and faults likely to be inserted when developing an implementation based on the specification; they are discussed further in Section 3.2. By manually inspecting the resultant mutant models, we found that 28 of them were semantically identical to the original FGS model. Therefore, our study uses the remaining 72 mutants that contain real faults as targets of the testing effort.
3. We ran the full (non-reduced) test suite on the 72 faulty implementations and recorded the number of faults revealed.
4. We generated and ran five reduced test suites for each full test-suite, ensuring that the desired coverage criterion was maintained. As discussed below, since the test reduction algorithm picks test-cases at random from the complete test-suite we generated several reduced sets for each full test-suite to avoid skewing our results because we were lucky (or unlucky) in the selection of tests for a reduced test-suite. We arbitrarily chose to generate five reduced test-suites to provide higher confidence in our results while at the same time keep the effort conducting the experiments manageable.
5. Given the results of the previous steps, we compared the relative fault finding capability of the full test-suites versus the reduced test-suites.

In the remainder of this paper we provide a detailed description of activities involved in the experiment and discuss our findings.

3.2 Fault injection and detection

To provide targets for our testing effort we needed a collection of fault implementations. Since the structure of an RSML^{-e} model is basically a collection of nicely formatted nested if-statements where each condition guards the assignment of a state variable, an implementation of the model will have a very similar structure to the model from which it was developed. Consequently, the fault classes applicable in the implementation domain are similar to the ones in the model domain. In addition, we have extensive support to execute, manipulate, and measure test coverage over RSML^{-e} models. Therefore, we decided to use a collection of fault seeded RSML^{-e} models to emulate the implementations rather than implementing the models in, for example, C, and seed faults in the resultant implementations (we would simply seed faults from the same fault classes in an implementation in C rather than an "implementation" in RSML^{-e}).

To create the faulty specifications, we first reviewed the revision history of the FGS model to understand what types of faults were commonly made when developing models. We also studied how the models would be implemented in source code to determine which faults seemed likely to be inserted in a manual implementation. Note here that our focus has been on faults inserted during manual coding, not the types of faults that might be inserted by a code-generator. Nevertheless, given the limited types of faults possible when deriving an implementing from a model expressed in a language such as RSML^{-e}, we believe that the faults injected by a code-generator

```
MACRO When_LGA_Activated() :
 TABLE
  Select_LGA()                : T;
  PREV_STEP(..LGA) = Selected : F;
  Is_This_Side_Active         : *; /* Was T */
 END TABLE
END MACRO
```

Fig. 5 An example fault seeded into the FGS model

would come from the same fault classes—the main difference from manual coding would be that a code-generator would presumably inject the faults systematically rather than spuriously as injected by a programmer.

The faults that we identified as as commonly inserted during the FGS development effort as well as likely faults to be injected during implementation fell into the following four categories:

Variable Replacement (VR): A variable reference was replaced with a reference to another variable of the same type.
Condition Insertion (CI): A condition that was previously considered a "don't care" (*) in one of the tables was changed to T (the condition is required to be true).
Condition Removal (CR): A condition that was previously required to be true (T) or false (F) in a table was changed to "don't care" (*).
Condition Negation (CN): A condition that was previously required to be true (T) in a table was changed to false (F), or vice versa.

As an example, Fig. 5 shows a missing condition fault contained in macro `When_LGA_Activated`, the fault was created by changing the table from requiring that the Boolean variable `Is_This_Side_Active` was true (*T*) to a "don't care" (*).

Since we focused on likely faults, we have omitted some possible fault classes, for example, missing state variables and misdirected or missing transitions. In our experience, such severe faults were uncommon and so obvious they would be quickly eliminated and, thus, would not linger until testing commenced.

We used our fault seeder to generate 100 faulty models to use as test targets (25 for each fault class). By manually inspecting the resulted faulty models, we found that 28 of them were semantically identical to the original FGS model. Therefore, our study uses the remaining 72 mutants that contain faults detectable through testing.

During our testing experiment, we used a quite sensitive oracle to determine if a test-case revealed a fault. Given the input sequence of a test-case, we compared both the generated output as well as the internal state of the model to determine if a fault was present. In general, the internal state information of the system under test may not be available to the oracle. Thus, our oracle was able to detect faults that may not have manifested themselves as erroneous outputs, but only as a corrupt system state. We chose this approach since we expect this to be the type of oracle used when performing conformance testing of auto-generated code; we have long advocated that the auto-generated code should conform to coding standards that make this type of traceability between specification and implementation feasible (Whalen and Heimdahl, 1999).

3.3 Specification based test criteria

Adequacy criteria are used by testers to decide when to stop testing by helping them determine if the software has been adequately tested. In this paper, these criteria are defined on a synchronous data-flow specification language. We are using the specification language RSML^{-e} (Thompson et al., 1999) in our study, but the criteria are applicable without modification to a broad class of languages. As mentioned in the background section, an RSML^{-e} model consists of state variables and a next state relation for these state variables (this can be viewed as state machines with transitions between the states). The next state relation defines under which conditions the state variables change value (the state machines change state), and are given in terms of Boolean expressions involving variables and arithmetic, relational, or boolean operators.

We use Γ to represent a test-suite and Σ for the formal model. In the following definitions, a *test-case* is to be understood as a sequence of values for the input variables in the model Σ and the expected outputs and state changes caused by these inputs. The sequence of inputs will guide Σ from its initial state to the structural element, for example, a transition, the test-case was designed to cover. A *test-suite* is simply a set of such test cases. In this paper we use the following six specification coverage criteria. Note that for the condition based coverage criteria, a *condition* is defined as a Boolean expression that contains no Boolean operators and a *decision* is Boolean expression consisting of conditions and zero or more Boolean operators. This definition of conditions and decisions follow the standard DO-178B (RTCA, 1992) governing civilian airborne software systems deployed in in the United States.

Variable Domain Coverage: (Often referred to as state-coverage.) Requires that the test set Γ has test-cases that force each control variable defined in the model Σ to take on all possible values in its domain at least once.

Transition Coverage: Analogous to the notion of branch coverage in code and requires that the test set Γ has test-cases that exercise every transition definition in Σ at least once. Note here that transition definition is the definition of a state transition at the level of the surface syntax of the modeling language, not the notion of a transition in the representation of the global state space.

Decision Coverage: Each decision occurring in Σ evaluates to true at some point in some test-case and evaluates to false at some point in some other test case. Note that if the decision is, for example, in a function, there is no requirement that the function is actually invoked—this criterion only requires that the decision *would* have evaluated to true/false *if* it was evaluated during the execution of the test-case. We chose to include this criterion since the published versions of decision coverage could be interpreted this way by a careless analyst. When instrumenting a model (or program) to measure test coverage the evaluation of the constructs of interest (in this case a decision) is implicit; when generating test-cases using model-checking techniques, the fact that the construct of interest must be evaluated must be explicit. In fact, we made this exact mistake ourselves in a previous study (Heimdahl et al., 2004).

Decision Coverage with Single Uses: Analogous to decision coverage, but the decision must actually be evaluated. For example, for a condition in a function, the

condition must evaluate to true/false while the function is invoked from some point in the model.

Modified Condition and Decision Coverage (MC/DC): MC/DC was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications (Chilenski and Miller, 1994b). MC/DC requires us to show the following:

- Every condition within the decision has taken on all possible outcomes at least once, and
- Every condition has been shown to independently affect the outcome of the decision

  Note again that invocation of the decision is not required because of the same reason discussed for decision coverage.

MC/DC with Single Uses: Analogous to modified condition and decision coverage, but the decision must actually be evaluated.

A more formal treatment of these coverage criteria can be found in Rayadurgam (2003), Rayadurgam and Heimdahl (2001a), Hayhurst et al. (2001), and FAA (2002).

3.4 Test set generation and reduction

We generated full test-suites using the approach discussed in Section 2.2. We used the NIMBUS to translate to the input language of the model-checker NuSMV (NuS, 2005) and also to generate the trap properties corresponding to the test coverage criteria discussed above. The model and the trap properties are then given to the NuSMV tool to create the full test-suites.

A single test-case usually satisfies more than one test obligation. For instance, a test-case used to cover a certain state of interest may also cover other states during its execution. This then provides for a way to reduce the size of the final test-suite by choosing a subset of test-cases that preserves the coverage obtained by the full test-suite.

Finding a minimal test-suite that satisfies the test requirements is in general an NP problem (Garey and Johnson, 1979), but often greedy heuristics suffice to generate significantly reduced test-suites. The method we use begins with an empty set of test cases and initializes the coverage to zero (Fig. 6). The greedy algorithm then randomly picks a test-case from the full test-suite, runs the test, and determines if the test-case improved the overall coverage (for whatever criterion in which we are interested). Any test-case that improves the coverage is added to the reduced set. This continues until we have exhausted all the test-cases in the full test-suite—we now have a, hopefully, much smaller suite that has the same coverage as the full test-suite.

The test case minimization approach we employ is a post processing technique to eliminate redundant test-cases. This means that we first generate the entire test-suite that satisfies a particular test criterion and then perform minimizations on it. If the goal was to get small test-suites it would be better to perform this filtering as the test-cases are generated. Nevertheless, since we also need the full test-suite for our experiment we choose this post processing approach.

Fig. 6 Algorithm for test-suite reduction

Algorithm 3.1: TEST-REDUCE(Σ, Γ, η)

INPUTS :
Model Σ, test suite Γ, and test criterion η
OUTPUT :
Reduced test set Ω

$\Omega \leftarrow \emptyset$; ReducedTest set
$AC \leftarrow 0$; Actual Coverage
$PC \leftarrow 0$; Previous Coverage
$shuffle(\Gamma)$;
repeat
choose a test case f from Γ;
run f against the model Σ;
Measure actual coverage AC;
if $AC \neq PC$
then $\Omega \leftarrow \Omega \cup \{f\}$;
$PC \leftarrow AC$;
until *Γ is exhausted*
return (Ω);

Note that we randomly select test-cases from the full set to create a reduced test-suite. We then generate five separate reduced test-suites for each full test-suite. We choose to generate five separate reduced sets to reduce problems related to skewing the results by accidentally picking a "very good" (or bad) set of test-cases. The results for all test runs are included in this report.

4 Experimental results and analysis

As a baseline for our experiments, we ran the full test-suites to determine their fault finding capability. To get a basic idea of their fault finding capability, we also created a collection of randomly generated tests to us as a comparison. We expended the same amount of time automatically generating and running the random tests as we did running the tests providing transition coverage. Thus, the randomly generated tests serve as a simple baseline for the other test suites; one would expect the tests carefully crafted to provide a certain coverage to perform better than the randomly generated test-set. The results are summarized in Table 1. The table shows the number of test-cases in each test-suite and their fault finding capability (total fault finding capability as well as broken down per fault-class).

Table 1 Full test set generation for various criteria along with their fault detection capability

Test criteria	Size	VR	CN	CI	CR	Total
Random	100	21	25	5	15	66 (92%)
Variable domain	115	14	15	2	4	32 (44%)
Transition	313	20	24	5	15	64 (89%)
Decision	435	23	24	5	15	67 (93%)
Decision usage	478	23	24	7	15	69 (96%)
MC/DC	537	22	25	7	16	70 (97%)
MC/DC usage	334	23	25	8	16	72 (100%)

Table 2 Reduced test set sizes for various test reduction runs

Criteria	Full	r.1	r.2	r.3	r.4	r.5	Avg.	Red.
Var. domain	115	19	22	18	21	21	20.2	82%
Transition	313	35	43	29	38	43	37.6	88%
Decision	435	45	44	44	45	42	44.0	90%
Decision usage	478	37	43	47	43	38	41.6	91%
MC/DC	537	34	33	29	34	32	32.4	94%
MC/DC usage	334	30	30	33	32	33	31.6	91%

r.1–r.5 indicate the size of the five reduced test-suites. Avg. denotes the average reduction of the five

As can be seen, the randomly generated test perform surprisingly well compared to the test-suites providing structural coverage. Note here that the randomly generated tests were of the length 100 input steps whereas the tests generated to a specific coverage were typically very short (1–3 steps). Nevertheless, the comparison is still relevant since we were not interested in comparing the number of test-cases but rather the effort involved in generating the tests; given equivalent effort in test-generation and test-execution, random testing performed very well. We have discussed the reasons behind the poor performance of Variable Domain and Transition Coverage in a previous study (Heimdahl et al., 2004) and a discussion of this topic is outside the scope of this paper.

From the results in Table 1 one can also observe that the more rigorous the test criteria, the better the fault finding capability. For instance, for this particular case-example MC/DC with usage detects all faults yielding a behaviorally different model.

4.1 Test-suite reduction

As mentioned earlier, we generated five different reduced test-suites for each coverage criterion to control the possibility that we by chance got a very "good" or very "poor" reduced test-suite. The results of the reduction algorithm can be seen in Table 2.

The results support our first hypothesis that test reduction results in significant savings in terms of test-suite size. In all cases there was at least an 80% average reduction in the size of the test-suite. This reduction reinforces the findings in Wong et al. (1997, 1998), Rothermel et al. (1998), Jones and Harrold (2003) and is to be expected since our test-case generation method produces one test-case per construct of interest (variable value, transition, decision value, or MC/DC value). This generation approach is desirable since it makes traceability of the test-cases to their test-objective straightforward (one-to-one mapping). On the other hand, as is evident from Table 2, it leads to a significant number of overlapping (with respect to coverage) test-cases that may or may not add to the fault-finding ability of the test-suite; this is the topic of the next section.

4.2 Effect on fault detection effectiveness

The fault finding capability of the full as well as reduced test-suites is summarized in Table 3. The results are in agreement with our second hypothesis that test-suite

Table 3 Fault finding capability of the reduced test-sets

Criteria	Full set	r.1	r.2	r.3	r.4	r.5	Avg	Red
Var. domain	32	28	29	25	28	25	27.0	15.6%
Transition	64	58	58	58	59	57	58.0	9.38%
Decision	67	62	61	62	62	61	61.6	8.06%
Dec. usage	69	62	63	63	62	63	62.6	9.28%
MC/DC	70	64	63	63	63	63	63.2	9.71%
MC/DC usage	72	67	66	67	67	67	66.8	7.22%

r.1–r.5 indicate the number of faults found with each of the five reduced test-suites. Avg. denotes the average number of faults found with the five reduced test-suites.

reduction will adversely impact the fault finding ability of test-suites that are derived from synchronous data-flow models.

As shown in Table 3, the number of faults detected by the reduced test-suites is significantly less for all coverage criteria that were examined in our experiment; in all cases there was at least a 7% reduction in the fault detection effectiveness. One may argue that a 7% reduction is rather small, but for our domain of interest, automated code generation in critical systems, any reduction in fault finding ability is unacceptable.

From our results we can also observe that the most rigorous coverage criteria, MC/DC with Usage, seems to be the least sensitive to the effect of test-suite reduction. We speculate that this is because it is simply harder to come up with a test-suite that provides this high level of coverage without finding faults—MC/DC with Usage is simply a "better" coverage criterion than the other ones we used in our experiment. We hypothesize that MC/DC with Usage is better than the other criteria in two respects. First, it seems to find more faults than any other criteria. Second, it seems to be less sensitive to the effect of test-suite reduction. Thus, MC/DC with Usage is the closest to the *ideal coverage criterion* in this domain we have seen to date; a test-suite generated to the ideal criterion would detect *all* faults in the system under test and *any* test-suite, large or small, providing this coverage would reveal the same faults.

Our results are markedly different than the results reported in previous studies (Wong et al., 1998; Jones and Harrold, 2003; Rothermel et al., 1998); one of the studies reports no reduction in fault finding and two studies report a dramatic and highly varied reduction in the fault finding capability of the reduced test-suites. In our study we observe a modest, but notable, reduction in the fault-finding capability. In our experiment, however, that reduction in fault-finding seems to be reasonably *predictable*; each of the five reduced test-suites we randomly generated for each coverage criterion have approximately the same fault-finding capability. This stands in stark contrast to the results in the Rothermel et al., and Jones and Harrold studies where the reduction in fault finding varied between 0% and 100% (Jones and Harrold, 2003; Rothermel et al., 1998).

We do not have a ready explanation for this phenomenon, but we speculate that it may be related to three factors; (1) the coverage criteria used in the experiment, (2) the actual coverage provided by the test-suites, and (3) the classes of faults considered. The Rothermel et al. study (1998) used edge-coverage of the control flow graph (roughly equivalent to the transition coverage in our domain) and most of our criteria are more

rigorous than edge-coverage. Since there seems to be a correlation between the rigor of the coverage criterion and the variability in fault-finding of the reduced test-suites, this may be part of the explanation for our results. The Jones and Harrold study (Jones and Harrold, 2003) used MC/DC as the coverage criterion in their experiment, but the test-suites they used did not provide complete MC/DC coverage. Their reduced test-suites provided the *same* coverage of the code as the full suite, but the full suite did not provide coverage up to 100% of achievable coverage of the criterion of interest. In our case, we provided full coverage of every criterion. The fact that we worked from complete test-suites may have made our test-suites less susceptible to the variations if fault finding observed in their study. Finally, since we are working with languages with limited expressiveness (no iteration and dynamic memory for example), the fault classes under consideration most likely affect the results. Needless to say, further study is clearly needed to understand these issues better.

To summarize the findings, reduction of test-suite size has an unacceptable effect on the suite's fault finding capability. Should there be an urgent need to reduce the test-suite size because of resource limitations (in terms of, for example, time), we speculate that *test-case prioritization* (Jones and Harrold, 2003) would be a better approach than test-suite reduction (or minimization). In test-case prioritization, we would not eliminate any test-cases from our test-suite; we would instead attempt to *sort* the test-cases based on expected fault finding potential and execute the ones deemed to be most likely to reveal faults first. We would terminate the testing when our resources are depleted. Naturally, more work is needed to determine how to prioritize test cases and also empirically evaluate if the test-case prioritization approach in fact performs better than reduced or minimized test-suites.

4.3 Threats to validity

There are four obvious threats to the external validity that prevents us from generalizing our observations. First, and most seriously, we are using only one instance of a formal model in our experiment. Although the FGS is an ideal example—it was developed by an external industry group, it is large, it represents a real system, and is of real world importance—it is still only one instance. The characteristics of the FGS model, for example, it is entirely modelled using Boolean and enumerated variables, most certainly affects our results and makes it impossible to generalize the results to systems that, for example, contain numeric variables and constraints.

Second, we are using fault seeded models to emulate an implementation in conformance testing. As mentioned earlier, given the structure and semantics of the modelling language RSML^{-e}, we assert that any reasonable implementation derived from such a model would be susceptible to the same types of faults as the model. Nevertheless, the fact that we are not using hand generated code in the experiment might bias the results and is, therefore, a concern.

Third, we are using seeded faults in our experiment. Although we took great care in selecting fault classes that represented actual faults we observed during the development of the FGS model as well as faults we deem highly likely to be introduced during implementation, fault seeding always leads to a threat to external validity—we simply do not know if our faults represent what we would see in practice.

Finally, we only considered a single fault per model. Using a single fault per specification makes it easier to control the experiment. Nevertheless, we cannot account for the more complex fault patterns that may occur in practice and the effect multiple faults may have on our ability to reveal them; for example, fault masking may affect the effectiveness of test suites.

Although there are several threats to the external validity of our experiment, believe the results generalize to a large class of models in the critical systems domain and our results raise serious doubts about the use of any test-suite reduction techniques in this domain. Note here that test-case generation techniques that are optimized—optimized in the sense that they only generate tests needed to provide the desired coverage—would be subject to the same potential drawbacks as the test-set reduction technique we evaluated in this paper.

5 Summary and conclusion

We have described an experiment in which we investigated the effect of test-suite reduction in the domain of automatically generated conformance test-suites. As a system-under-test, we used a model of a production sized Flight Guidance System seeded with "representative" faults. Our results confirm our two hypotheses; one can dramatically reduce the size of the automatically generated conformance test-suites while maintaining desired coverage, and the fault finding of the reduced test-suites was significantly adversely affected. Although we cannot broadly generalize our results and further studies are needed, the experiment indicates that test-suite reduction of test-suites providing structural coverage may not be an effective means of reducing testing effort—the cost is terms of lost fault finding capability is simply too high; especially in the critical systems domain in which we are mainly interested. On a related note, the results cast doubts on the effectiveness of structural coverage criteria in general. Small (or minimal) test-suites that provide structural coverage criteria do not seem to be effective; we must understand better the reasons behind this problem and either discover ways to somehow augment the test-suites to enhance their effectiveness or define structural coverage criteria that are not sensitive to test-suite size. Our results indicate that more rigorous criteria, such as MC/DC, provide a better fault finding capability both for the full-test suites as well as the reduced test suites as compared to less rigorous criteria, such as variable domain and transition coverage. This hints that MC/DC is more robust to the detrimental effect of test-suite size. Nevertheless, the characteristics of coverage criteria in both the specification and implementation domain have not been well investigated and more work in this area is clearly needed.

Based on our results, we are skeptical towards any test-suite reduction techniques that aim solely to maintain structural coverage, because, in our opinion, there is an unacceptable loss in terms of test-suite quality. Thus, we advocate research into test-case prioritization techniques and experimental studies to determine if such techniques can more reliably lessen the burden of the testing effort by running a subset of an ordered test-suite, as opposed to a reduced test-suite, without little if any loss in fault finding capability.

References

NuS: The NuSMV Toolset. Available at http://nusmv.irst.itc.it/ (2005)

Ammann, P.E., Black, P.E.: 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 (1999)

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

Archer, M., Heitmeyer, C., Simsm S.: TAME: A PVS interface to simplify proofs for automata models. In: User Interfaces for Theorem Provers (1998)

Bensalem, S., Caspi, P., Parent-Vigouroux, C., Dumas, C.: 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), pp. 89–107. IEEE Computer Society, San Jose, CA (1999)

Blackburn, M.R., Busser, R.D., Fontaine, J.S.: Automatic generation of test vectors for SCR-style specifications. In: Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97 (1997)

Callahan, J., Schneider, F., Easterbrook, S.: Specification-based testing using model checking. In: Proceedings of the SPIN Workshop (1996)

Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.: Model checking large software specifications. IEEE Trans. Softw. Eng. **24**(7), 498–520 (1998)

Chilenski, J., Miller, S.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **9**, 193–200 (1994a)

Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. 193–200 (1994b)

Choi, Y., Heimdahl, M.: Model checking RSML^{-e} requirements. In: Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering, pp. 109–118. Tokyo, Japan (2002)

Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

Engels, A., Feijs, L.M.G., Mauw, S.: Test generation for intelligent networks using model checking. In: Proceedings of TACAS'97, LNCS 1217, pp. 384–398. Springer (1997)

Esterel Technologies: SCADE suite product description. http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html (2004)

FAA, C.A.S.T.: What is a "decision" in application of modified condition/decision coverage and decision coverage (DC)? Technical Report position paper (2002)

Frankl, P., Weiss, S.N.: An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In: Proceedings of the symposium on Testing, analysis, and verification (1991)

Garey, M., Johnson, D.: Computers and Intractability. Freeman, New York (1979)

Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. Softw. Eng. Notes **24**(6), 146–162 (1999)

Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Trans. Program. Lang. Syst. **16**(3), 843–871 (1994)

Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M. STATEMATE: A working environment for the development of complex reactive systems. IEEE Trans. Softw. Eng. **16**(4), 403–414 (1990)

Hayhurst, K., Veerhusen, D., Rierson, L.: A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA (2001)

Heimdahl, M.P., Devaraj, G.: 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 (2004)

Heimdahl, M.P., Devaraj, G., Weber, R.J.: 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 (2004)

Heimdahl, M.P., Rayadurgam, S., Visser, W.: Specification centered testing. In: Second International Workshop on Analysis, Testing and Verification (2001)

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

Heimdahl, M.P.E., Leveson, N.G.: Completeness and consistency in hierarchical state-base requirements. IEEE Trans. Softw. Eng. **22**(6), 363–377 (1996)
Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: SCR*: A toolset for specifying and analyzing requirements. In: Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95 (1995)
Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. **5**(3), 231–261 (1996)
Heitmeyer, C., Jr, J.K., Labaw, B., Archer, M., Bharadwaj, R.: Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Trans. Softw. Eng. **24**(11), 927–948 (1998)
Hong, H.S., Cha, S.D., Lee, I., Sokolsky, O., Ural, H.: Data flow testing as model checking. In: Proceedings of the International Conference on Software Engineering. Portland, Oregon (2003)
Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: 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 (2002)
Jasper, R., Brennan, M., Williamson, K., Currier, B., Zimmerman, D.: Test data generation and feasible path analysis. In: Proceedings of International Symposium on Software Testing and Analysis (1994), pp. 95–107.
Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. **29**(3), 195–209 (2003)
Lee, E.A.: Overview of the Ptolemy Project. Technical Report Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA (2003)
Leveson, N., Heimdahl, M., Hildreth, H., Reese, J.: Requirements specification for process-control systems. IEEE Trans. Softw. Eng. **20**(9), 684–706 (1994)
Mathworks Inc.: Mathworks Inc. Simulink Product Web Site. via the world-wide-web: http://www.mathworks.com.
Offutt, A.J., Xiong, Y., Liu, S.: Criteria for generating specification-based tests. In: Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99) (1999)
Owre, S., Shankar, N., Rushby, J.: The PVS specification language. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta relese edition (1993)
Pnueli, A.: Applications of temporal logic to specification and verification of reactive systems: A survey of current trends. Lecture Notes in Computer Science Number 224, pp. 510–584 (1986)
Rayadurgam, S.: Automatic test-case generation from formal models of software. Ph.D. thesis, University of Minnesota (2003)
Rayadurgam, S., Heimdahl, M.P.: 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), pp. 83–91. IEEE Computer Society (2001a)
Rayadurgam, S., Heimdahl, M.P.: 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 (2001b)
Rothermel, G., Harrold, M., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Proceedings of the International Conference on Software Maintenance, pp. 34–43 (1998)
RTCA: Software Considerations In Airborne Systems and Equipment Certification. RTCA (1992)
Thompson, J.M., Heimdahl, M.P.: An integrated development environment prototyping safety critical systems. In: Tenth IEEE International Workshop on Rapid System Prototyping (RSP) 99, (1999), pp. 172–177.
Thompson, J.M., Heimdahl, M.P., Miller, S.P.: Specification based prototyping for embedded systems. In: Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering, pp. 163–179 (1999)
Whalen, M.W., Heimdahl, M.P.: On the requirements of high-integrity code generation. In: 4th IEEE International Symposium on High Assurance Systems Engineering, Vol. LNCS yyy, (1999), pp. 217–226.
Wong, W., Horgan, J., Mathur, A., Pasquini, A.: Test set size minimization and fault detection effectiveness: a case study in a space application. In: Proceedings of the 21st Annual International Computer Software and Applications Conference, (1997), pp. 522–528.
Wong, W., Horgan, J., London, S., Mathur, A.: Effect of test set minimization on fault detection effectiveness. Softw. Pract. Exp. **28**(4), 347–369 (1998)