

Generating Regression Tests using Model Checking

Lihua Xu and Debra Richardson

{lihuax, djr} @ ics.uci.edu

Department of Informatics, School of Information and Computer Science,
University of California at Irvine, CA, USA

Abstract

During the software maintenance phase, regression testing is certainly an expensive but necessary activity to make sure the new versions of the system do not “regress”. As software evolves, chances are that not only the implementation changes, but that the specification of the system changes too. We argue that guiding regression testing by the system specifications will be more accurate and cost effective.

Model checking is used to reason about the truth of statements about the system specification. In this paper, we use a model checker as part of a highly-automated regression test generation tool, providing a practical approach to specification based regression testing. Features of our approach that support different specification description styles and different test coverage criteria are also presented.

1 Introduction

Regression Testing is a maintenance activity that attempts to validate modified software and ensure that modifications are correct and have not inadvertently affected the software. Selective regression testing attempts to identify test cases for reuse in testing modified portions of the system. Specification-based regression testing techniques select reusable test cases from the original test suite with respect to formal specifications of the system. One should make sure that the cost of selecting the regression tests is lower than retesting with the entire original test suite.

Most specification-based regression testing approaches extract the “specification” from the implementation and then use this extracted specification to guide their test case selection ([1], [2], [3], [4]). One drawback of these approaches is that they not only increase the cost of regression testing by adding the cost of extracting the specification, but also the derived specification may suffer the same wrong assumptions as did the implementation. We solve this particular problem by combining regression testing with model checking, which uses the formal specifications developed in the early stage of the software lifecycle, instead of extracting “specifications” from source code, to guide our regression test generation.

Our work is also motivated by the ability of model checkers to generate counterexamples, where each counterexample can be considered as a complete test sequence. Thus, our regression test generation toolkit is

able to benefit from most of the technology of existing model checkers, which in turn, motivates the developers to formalize their development process by making wider use of formal methods, since practitioners could validate the specification and verify the implementations with only a little extra effort.

The general idea of our work is to use the difference between two versions of software specifications – the original one and the modified version to guide our regression test generation. Our approach is able to support different specification description styles, such as model checker languages, architecture descriptions, FSM, scenarios, LTS, and so on.

Even though regression testing is aimed at reducing testing costs, different users may have different requirements and cost constraints for the level of granularity of regression testing. Therefore, our approach supports different test coverage criteria. Currently, our work allows users to choose from a limited set of FSM-based test coverage criteria to generate regression test suites, including state transition (STL), syntactic trigger events (SynL) and semantic trigger events (SemL).

2 Background

This section reviews some background for our work.

2.1 Model Checking

A model checker takes a formal description of the system as input and effectively analyzes it with respect to given properties. Therefore, a model checking specification consists of two parts. One part describes the functional behavior of system as a state machine defining variables and conditions under which variables may change value. The other part constrains execution as named properties that are expressed as invariant conditions by temporal logic constraints. Conceptually, a model checker visits all reachable states to verify the properties, and in cases where a property is not satisfied, it attempts to generate a counterexample in the form of a sequence of states.

2.2 Regression Testing

Regression testing is a necessary though expensive maintenance activity that attempts to validate modified software and ensure that modifications are not only correct but also have not inadvertently affected the software so that portions that used to work no longer work correctly. The simplest regression testing strategy, *retest all*, tends to rerun all of the test cases in the original test suite on a modified version, and is therefore very time-consuming and expensive. An alternative,

selective retest, chooses only those tests that are associated with the modified portions. In either case, it is necessary to generate some new tests to cover untested modified portions of the system.

Traditionally, the process of regression testing is conceptualized as follows. Assume as available: a program P , a modified version P' , and a test suite T which is used to test P together with the expected test results. In order to reduce costs and to reuse T as effectively as possible, selective-retesting techniques first select a subset of T , T' , based on the modifications of P by P' . Second, if necessary, a set of new tests T'' is created, to test the new, modified and untested portion of P' . Thus, the test suite for P' is $T' + T''$. The main focus of this paper is on the first step.

3 Research Goals

To support cost-effective specification-based regression testing, we propose a novel approach of combining regression testing via model checking with the following research goals.

- Different specification description style support:** Rather than using a single description style to represent the system, users may have preferences for using different specification description styles (i.e., different languages for different model checkers, FSM, scenarios, software architecture descriptions, LTS) to represent different abstract levels of systems. Similarly, specification-based regression test generation should also enable users to generate regression tests from potentially different specification styles.
- Different test coverage criteria support:** To provide support for a variety of users, different requirements and cost constraints need to be considered. Thus, our approach should provide hierarchically layered property-based testing criteria where users are able to cover different levels of regression testing. We will describe our solution details in Section 4.
- Automation:** The creation of regression test cases for the modified system should be automated. The capability to transition from different specification description styles to the language fed into the model checker and property extraction mechanism should also occur without the need for user intervention.
- Safe and Minimal:** Regression testing should be as safe as possible: that is, users should be able to select every test from the original test suite that can expose faults in the modified program under certain coverage criteria. On the other hand, the purpose of regression testing is to reduce testing cost, thus, the resulting regression test cases should not include those that cannot expose change in the modified system. Our work handles this problem by

generating the regression test cases from the differences between two versions of the system, instead of only selecting from original test suite; therefore, under certain coverage criteria; the approach covers those and only those changes, including the added parts, made for the modified specification.

- Quality of Test Case Guarantees:** The specification used to guide regression testing should be correct in the sense of being consistent with users' requirements. Our approach includes two phases: phase 1 checking the correctness of the modified specification in terms of application-independent properties using a model checker; and phase 2 constructing regression test sequences based on the differences between two versions of the specification. These two phases help us to increase our confidence that the implementation matches the users' requirements.

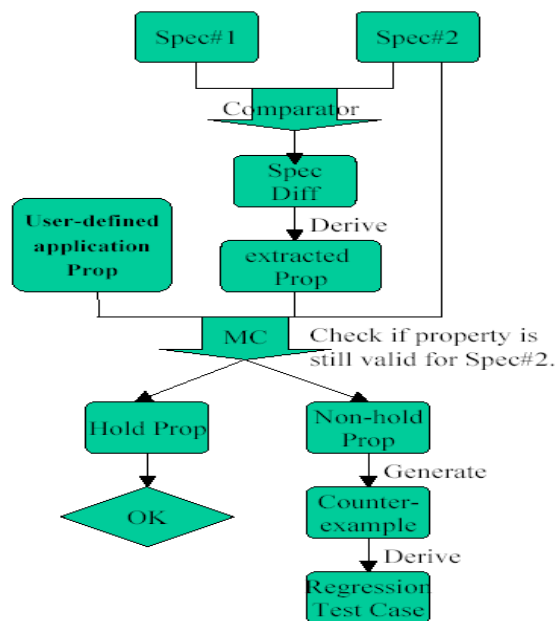


Figure 1. Overall Approach of RTMC

4 Approach

We propose our overall approach shown in Figure 1 to achieve the above research goals. We name our approach RTMC (*Regression Testing via Model Checking*), which relies on the existence of two different specification models – Spec#2 is the specification of a modified version of the same system described by Spec#1. The idea is to use the difference between the two models to guide regression test case selection.

First, Spec#1 and Spec#2 are fed into a *comparator*; the differences between the specifications based on different test coverage criteria (Spec Diff) will

be generated and used to derive properties (extracted prop) such that each property covers one test path of the specification that has been modified and we are interested in retesting. Second, Spec#2 is run through a model checker with respect to these properties, together with other user-defined application properties (i.e., safety properties). For a property that holds (a Hold Prop), this branch is considered not to have changed. For a property that does not hold (a Non-Hold Prop), the property is considered to have changed and hence must be retested – that is, test cases are generated from each counterexample and retested over the modified source code.

Our approach is based on the following underlying ideas:

1. For each hold property, the branch is considered correct (that is, if it was correct before, it remains correct);
2. The model checker will generate at least one counterexample for each non-hold property.

It is worth noting that we are aware of the case when hold properties should not hold– that is, it might be the case that modifications are made because properties have changed. But the including of user-defined application properties certainly weaken the likelihood of this problem. It is also possible that counterexamples cannot be generated with respect to certain properties. For instance, suppose that a property specifies that a possible execution path leads to a certain state, while in fact no such execution path exist – no counterexample will exhibit this property [5]. Such cases are beyond the scope of this paper.

Below we describe various components and important decisions in our approach.

Comparator: Our approach provides *comparators*, that are able to take a variety of different specification types as input, and can output FSM-like tables for Spec#2, with specific descriptions of differences between the two versions of the specification.

Property Derivation: Before results from the *Comparator* are fed into the model checker, the differences between the two versions of the specification are derived into properties such that every property is actually a testing path/branch of the system. Here, our approach uses assertions to represent our extracted properties, since we chose SPIN [6] to implement our approach.

It is worth noting that we only use SPIN at a level that allows us to generate counterexamples, the approach can be used with any model checker. We specifically do not wish to limit the application of our technique to a single model checker.

Hierarchical Test Coverage Criteria: Our approach provides hierarchical regression test criteria by deriving property abstractions to test at different levels of

granularity (see Figure 2) – recall that properties are really testing paths/branches of the models. At the *state transition level (STL)*, coarse-grained properties that only label high-level branches are derived. Thus, the regression test suite will include cases in which properties would cover state transitions, whose variables’ initial and/or final states have been changed. At the *syntactic trigger events level (SynL)*, differences in trigger events are noted, in which case properties would cover not only state transitions but also events related to each transition. Thus, the test paths whose variables’ initial and/or final states have not been changed, but their related events have been syntactically changed, will be added into the regression test suite. At the *semantic trigger events level (SemL)*, only semantic differences in trigger events are considered. For instance, a name change is an insignificant change, although semantic changes in which the value of an event variable has changed or an event has been totally modified are the ones that must be retested.

Our initial exploration of this approach only implements the state transition level.

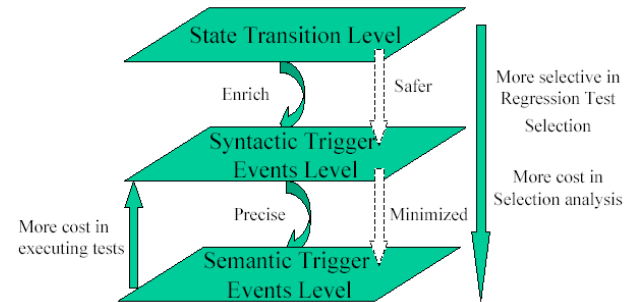


Figure 2. Coverage Criteria of our approach

Our attempt to build a hierarchical structure to guide specification-based regression testing is based on the intuition that increased test selection cost is offset by test execution savings ([7], [8]). As can be seen in Figure 2, regression testing at a lower level of granularity requires additional test selection cost but results in a smaller test sets and hence lower test execution costs. Harrold and Rothermel categorize regression test strategies from *safe* to *minimizing* [8]; very reliable results require a *safe* test selection strategy, whereas a *minimization* technique should be chosen if retesting time is important regardless of cost. Clearly it is not always wisest to select the smallest regression test suite; thus, sometimes the most coarse-grained derivation will satisfy the goal. In our approach, from STL to SynL, the approach makes the test suite more *safe*, while from SynL to SemL, the method *minimizes* the test suite.

5 Case Study

To demonstrate our approach more clearly, we use the Safety Inject System (SIS) [9] as a small example with some complexity. Many variations on this example have been presented in the literature. As our original model, we use the one from Heitmeyer et.al. ([10], [11]), which is a simplified version of a control system for safety injection in a nuclear plant.

5.1 Two Versions of Model

The SIS system monitors water pressure. When the pressure is too low (lower than some threshold), the system injects coolant into the reactor core. For the sake of simplicity, suppose we only interested in testing one component of the system, *mcPressure*, which monitors the state of pressure with three possible values *TooLow*, *Permitted*, and *High*. At any given time, the system must be in one and only one of these states. Variable *mWaterPres* represents the actual value of water pressure.

Although there may exist different description style for the specification, in our example, we chose to use event tables from SCR specification to illustrate the difference of two models. Our approach works same for other descriptions styles. Table 1 shows the initial/final state with their trigger events of the original model (Spec#1), while Table 2 shows the event table of the modified model (Spec#2), with the changes shown shaded.

| Old Mode | Event | New Mode |
|-----------|--------------------------|-----------|
| TooLow | @T(mWaterPres >= Low) | Permitted |
| Permitted | @T(mWaterPres >= Permit) | High |
| Permitted | @T(mWaterPres < Low) | TooLow |
| High | @T(mWaterPres < Permit) | Permitted |

Table 1. Event Table for *mcPressure*

| Old Mode | Event | New Mode |
|-----------|---|-----------|
| TooLow | @T(mWaterPres' >= Permit) | High |
| TooLow | @T(mWaterPres' >= Low) & @T(mWaterPres' < Permit) | Permitted |
| Permitted | @T(mWaterPres' >= Permit) | High |
| Permitted | @T(mWaterPres' < Low) | TooLow |
| High | @T(mWaterPres' < Permit) & @T(mWaterPres' >= Low) | Permitted |
| High | @T(mWaterPres' < Low) | TooLow |

Table 2: Modified Event Table for *mcPressure*

5.2 Different Test Coverage Criteria

Using our defined test coverage criteria, one can easily categorize the changes in Table 2 into three:

1. Two more state transitions have been added. (shaded rows).
2. Variable *mWaterPres* changed its name to *mWaterPres'*.
3. Two trigger events have been changed semantically (shaded events).

Therefore, as shown in Figure 3, shaded arrows show test paths determined to have changed using different test coverage criteria. To be specific, the first categorized group of changes belong to the state transition level (STL), where only the changes of initial and final states are covered in the properties. The syntactic trigger events level (SynL), where if the events change in any way then the test path would be considered to have changed, include all three groups of changes. The semantic trigger events level (SemL), where the modified trigger events have changed in a meaningful way, include the first and third groups of change. Thus, the regression tests generation results from STL would be subsumed by SemL, which is subsumed by SynL.

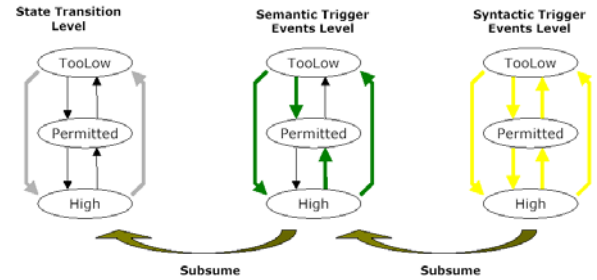


Figure 3. Different Coverage Criteria

5.3 Property Derivation

Based on the coverage criteria chosen by the regression tester, the comparator generates an FSM-like table for Spec#2, with specific notice and numbering of the testing paths needing to be retested.

Take STL as an simple example, the FSM-like table generated for modified version of *mcPressure* is shown in Table 3, with Promela, the language for SPIN, used to describe the trigger events, since we are using SPIN to partly implement our approach.

| No | Initial State | Trigger Events | Final State |
|----|---------------|--|-------------|
| 1 | TooLow | ((!(Permit>mWaterPres_NEW)) &&(Permit>mWaterPres_OLD)) | High |
| - | TooLow | ((!(Low > mWaterPres_NEW)) && (Low > mWaterPres_OLD)) &&((Permit> WaterPres_NEW) &&(!(Permit> aterPres_OLD)))) | Permitted |
| - | Permitted | ((!(Permit> mWaterPres_NEW)) &&(Permit>mWaterPres_OLD)) | High |

| | | | |
|---|-----------|---|-----------|
| - | Permitted | ((Low > mWaterPres_NEW) && !(Low > mWaterPres_OLD)) | TooLow |
| - | High | ((!(Permit > mWaterPres_NEW) && !(Permit > mWaterPres_OLD)) && !(Low > mWaterPres_NEW) && (Low > mWaterPres_OLD)) | Permitted |
| 2 | High | ((Low > mWaterPres_NEW) && !(Low > mWaterPres_OLD)) | TooLow |

Table 3. FSM-like Table for Modified *mcPressure*

Therefore, when Spec#2 is translated to Promela to be run through the SPIN model checker, two assignments are inserted into these two numbered test paths: `assertion_1 = 1`, and `assertion_2 = 1`, respectively (see Figure 4). Since in the state transition level, differences in trigger events are not detected, rows 2 and 5 in Table 2 will not be considered. Thus, the test path associated with these two lines will not be tested at this level.

```

if
  :: (!(Permit > mWaterPres_NEW) && (Permit >
    mWaterPres_OLD) && (mcPressure_OLD == TooLow) ->
    mcPressure_NEW = High; assertion_1 = 1;

  :: (!(Low > mWaterPres_NEW) && (Low >
    mWaterPres_OLD) && ((Permit > mWaterPres_NEW) &&
    (!(Permit > mWaterPres_OLD)))) && (mcPressure_OLD ==
    TooLow) -> mcPressure_NEW = Permitted;

  :: (!(Permit > mWaterPres_NEW) && (Permit >
    mWaterPres_OLD) && (mcPressure_OLD == Permitted) ->
    mcPressure_NEW = High;

  :: ((Low > mWaterPres_NEW) && !(Low >
    mWaterPres_OLD)) && (mcPressure_OLD == Permitted) ->
    mcPressure_NEW = TooLow;

  :: ((Permit > mWaterPres_NEW) && !(Permit >
    mWaterPres_OLD)) && ((!(Low > mWaterPres_NEW) &&
    (Low > mWaterPres_OLD)) && (mcPressure_OLD == High))
    -> mcPressure_NEW = Permitted;

  :: ((Low > mWaterPres_NEW) && !(Low >
    mWaterPres_OLD)) && (mcPressure_OLD == High) ->
    mcPressure_NEW = TooLow; assertion_2 = 1;

:: else skip;
fi;

```

Figure 4. Promela Code for Event Table 2

Next, for each assignment inserted into the Promela code, a corresponding Promela assert statement is inserted at the end of the Promela specification as properties to be checked:

```

assert (assertion_1 != 1);
assert (assertion_2 != 1);

```

5.4 Regression Tests Generation

Each time SPIN checks the assertions against the modified specification, it will generate corresponding

counterexamples, which trace the violation of these properties, since the assert statement will not hold. In this way, the regression test sequences are derived.

For the sake of simplicity, we define the boundaries of the water pressure: *Low*=1 and *Permitted*=2, and the legal range of variable *mWaterpres* is [0, 2000]. Figure 5 shows one of the counterexamples derived from SPIN. Test sequences corresponding to these counterexamples are the traces whose initial values of the variables are described in the first step, and then follow the changes of variables in subsequent steps.

In our experiment, we found that SPIN tends to generate a set of counterexamples for each property. We also realize that by eliminating syntactic duplicates and ignoring the ones that are really part of the others, “prefixes” of others [12], or other test criteria, there can be further reduction of the test set size. Therefore, finite, reasonably sized test sets will be automatically derived, and our approach appears scalable.

```

Step 1:
mBlock = off
mReset = off
mWaterPres = 0
mcPressure = TooLow

Step 2:
mWaterPres = mWaterPres + 2

Step 3:
tOverridden = False
mcPressure = High
cSafety_Injection = off

```

Figure 5: Counterexample for assert (assertion_1 != 1)

It is worth noting that the SCR specification can be formally transformed to a source code implementation in which each property covers a corresponding test path in the implementation [11]. Therefore, the test cases we construct from counterexamples can actually be used to test the implementation.

6 Related Work

The general idea of using model checking in software testing is not new; the ability to construct counterexamples by a model checker has been proposed as a way of deriving test cases. This section presents some of the previous work that has influenced our own research.

Gargantini and Heitmeyer present an approach for constructing test sequences from SCR specifications [10]. In each SCR notation, each operational specification is translated into the language of a model checker by inserting assertions into every possible execution branch. Test sequences are then derived from the counterexamples generated by the model checker.

Although we also use assertions to construct counterexamples, their approach is focused on generating test sequences only; as discussed in Section 5.4, our approach is concentrating on deriving regression test sequences.

Black, Ammann and their colleagues present an approach to apply model checkers for evaluating and generating test sets based on mutation analysis ([5], [12]). Their test evaluation is done by measuring test coverage while running the tests on the specification, with respect to mutation test coverage criteria. Test generation is done by applying mutations to both specifications and properties, where the test sets derived from the mutation of specification should be failing tests and ones derived from mutations of properties should be passing tests.

Callahan and colleagues [13] verify execution traces, which are generated by simulating the program, using a model checker as a semantic tableau, and generate test cases by constructing equivalence partitions based on the conjunction of all requirements and their logical complements such that each combination is a coverage property.

Eagels and colleagues also use properties, which they assume are “test purposes” defined by the developer, to derive test sequences [14]. This approach, however, may suffer from the fact that the properties may be incomplete, and sometimes even not available.

Krishnan proposes generating test sequences directly from the specification of properties, instead of using model checker itself [15]. One drawback of it is the approach use decision tables or LTL as starting points, thus, it may suffer the fact that the “model” they are using to guide test generation is actually need to be defined by tester, the results may not complete.

7 Conclusion and Future Work

In summary, the contributions of RTMC are the concept of using a model checker as part of the regression test generation tool, a variety of specification description support, and different test coverage criteria support. Most importantly, it offers a significant opportunity to reduce testing costs and automatically produce regression tests from formal specifications with greater confidence

The SIS specification used in this paper is a fairly simple application. To demonstrate scalability of the method and applicability to more realistic, complicated software systems, we plan to use larger specifications taken from real life examples. We intend to implement tool support so as to gather hard data of the efficiency of applying our approach as well as to explore potential new criteria.

8 Acknowledgments

The authors appreciate the many discussions with members of the ROSATEA research group, and in particular the suggestions provided by Thomas Alspaugh,, Henry Muccini, Tim Standish, and Hadar Ziv.

9 Reference

1. D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. Kim, & Y. Song, Developing an object-oriented software testing and maintenance environment, *Communications of the ACM*, 1995
2. Y. Wu, D. Pan & M. H. Chen, Techniques of maintaining evolving component-based software, *Proceedings of the International Conference on Software Maintenance*, 2000, IEEE,
3. Y. Wu, M. H. Chen & H. M. Kao, Regression testing of object-oriented programs, *Proceeding of the International Symposium on Software Reliability*, 1999
4. S. Beydeda and V. Gruhn, Integrating White- and Black-Box Techniques for Class-Level Regression Testing, *COMPSAC 2001*.
5. P. E. Ammann, P. E. Black and W. Majurski, Using Model Checking to Generate Tests from Specifications, *Proceedings of 2nd ICFEM'98*, IEEE, 1998.
6. G. Holzmann, The Model Checker SPIN, *IEEE Trans. On Software Engineering*, Vol.23, No. 5, May 1997
7. D. S. Rosenblum and E. J. Weyuker. Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies. *IEEE Transactions on Software Engineering*; 23(3):146-156, March 1997
8. G. Rothermel and M. J. Harrold, Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529-551, August 1996
9. P.J. Courtois and David L. Parnas. Documentation for safety critical software. *Proc. 15th Int'l Conf. On Softw. Eng. (ICSE '93)*, Baltimore, MD, 1993.
10. A. Gargantini, C. Heitmeyer: Using Model Checking to Generate Tests from Requirements Specifications, *Proc., Joint 7th Eur. Software Engineering Conf. and 7th ACM SIGSOFT Intern. Symp. on Foundations of Software Eng. (ESEC/FSE99)*, Toulouse, FR, Sept. 6-10, 1999
11. Elizabeth I. Leonard, Constance L. Heitmeyer, Program Synthesis from Formal Requirements Specifications using APTS, *Higher-Order and Symbolic Computation*, Kluwer Academic Publishers, to appear.
12. P. Ammann, Paul E. Black and W. Ding, Model Checkers in Software Testing, *National Institute of Standards and Technology*, 2002. NIST-IR 6777.
13. J. Callahan, F. Schneider and S. Easterbrook. Automated software testing using model checking, *Proceedings 1996 SPIN workshop*, Rutgers, NJ, 1996
14. A. Engels, L. Feijis and S. Mauw, Test Generation for Intelligent Networks Using Model Checking, *Proc. TACSS'97*, 1997
15. P. Krishnan, Uniform Descriptions for Model Based Testing, *Australian Software Engineering Conference 2004*, to be appear.