Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 33–46

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Handling Model Changes: Regression Testing and Test-Suite Update with Model-Checkers

Gordon Fraser[1] ,[2] , Bernhard K. Aichernig[3] and Franz Wotawa[4]

Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz, Austria

Abstract

Several model-checker based methods to automated test-case generation have been proposed recently. The performance and applicability largely depends on the complexity of the model in use. For complex models, the costs of creating a full test-suite can be significant. If the model is changed, then in general the test-suite is completely regenerated. However, only a subset of a test-suite might be invalidated by a model change. Creating a full test-suite in such a case would therefore waste time by unnecessarily recreating valid test-cases. This paper investigates methods to reduce the effort of recreating test-suites after a model is changed. This is also related to regression testing, where the number of test-cases necessary after a change should be minimized. This paper presents and evaluates methods to identify obsolete test-cases, and to extend any given test-case generation approach based on model-checkers in order to create test-cases for test-suite update or regression testing.

Keywords: Testing with model-checkers, regression testing, test-suite update, model change, automated test-case generation

1 Introduction

The need for efficient methods to ensure software correctness has resulted in many different approaches to testing. Recently, model-checkers have been considered for test-case generation use in several works. In general, the counter example mechanism of model-checkers is exploited in order to create traces that can be used as test-cases.

[1] This work has been supported by the FIT-IT research project "Systematic test case generation for safety-critical distributed embedded real time systems with different safety integrity levels (TeDES)"; the project is carried out in cooperation with Vienna University of Technology, Magna Steyr and TTTech.

[2] Email: fraser@ist.tugraz.at

[3] Email: aichernig@ist.tugraz.at

[4] Email: wotawa@ist.tugraz.at

doi:10.1016/j.entcs.2007.08.004

If the model used for test-case generation is changed, this has several effects. Test-cases created with a model-checker are finite execution paths of the model, therefore a test-suite created previously to the model change is likely to be invalid. As test-case generation with model-checkers is fully automated the obvious solution would be to create a new test-suite with the changed model. This is a feasible approach as long as the model complexity is small. If the model complexity is significant, the use of a model-checker can lead to high computational costs for the test-case generation process. However, not all of the test-cases might be invalidated by the model change. Many test-cases can be valid for both the original and the changed model. In that case, the effort spent to recreate these test-cases would be wasted. There are potential savings when identifying invalid test-cases and only creating as many new test-cases as necessary. If a model is changed in a regression testing scenario, where a test-suite derived from the model before the change fails to detect any faults, running a complete test-suite might not be necessary. Here it would be sufficient to run those tests created with regard to the model change.

In this paper, we present different approaches to handle model changes. Which approach is preferable depends upon the overall objectives in a concrete scenario — should the costs be minimized with regard to test-case generation or test-case execution, or is it more important to ensure that the changes are correctly implemented? The contributions of this paper are as follows:

- We present methods to decide whether a test-case is made obsolete by a model change or if it remains valid. The availability of such a method allows the reuse of test-cases of an older test-suite and is a necessary prerequisite to reduce the costs of the test-case generation process for the new model.
- We present different methods to create new test-cases after a model change. These test-cases can be used as regression tests if the number of test-cases executed after a model change should be minimized. They are also used in order to update test-suites created with older versions of the model.
- An empirical evaluation tries to answer two research questions: (1) What is the impact of test-suite update on the overall quality compared to newly created test-suites? (2) Is there a performance gain compared to completely re-creating a test-suite after a model change?

This paper is organized as follows: Section 2 identifies the relevant types of changes to models and relates them to a concrete model-checker input language, and then presents our solutions to the tasks of identifying invalid test-cases and creating new ones. Section 3 describes the experiment setup and measurement methods applied to evaluate these approaches, and presents the results of these experiments. Finally, Section 4 discusses the results and concludes the paper.

2 Handling Model Changes

As this paper considers test-cases created with model-checkers, this section recalls the basic principles of such approaches. After a short theoretical view of model changes we use the input language of a concrete model-checker to present methods to handle model changes.

2.1 Preliminaries

A model-checker is a tool used for formal verification. It takes as input an automaton model and a temporal logic property and then effectively explores the entire state space of the model in order to determine whether model and property are consistent. If inconsistency is detected, the model-checker returns an example execution trace (counter example) that illustrates how a violating state can be reached. The idea of model-checker based testing is to use these counter examples as test-cases. Several different approaches of how to force the model-checker to create traces have been suggested in recent years. Model-checkers use Kripke structures as model formalism:

Definition 2.1 [Kripke Structure] A Kripke structure M is a tuple $M = (S, s_0, T, L)$, where S is the set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times S$ is the total transition relation, and $L : S \rightarrow 2^{AP}$ is the labeling function that maps each state to a set of atomic propositions that hold in this state. AP is the countable set of atomic propositions.

Properties are specified with temporal logics. In this paper we use Linear Temporal Logic (LTL) [12]. An LTL formula consists of atomic propositions, Boolean operators and temporal operators. X refers to the *next* state. For example, $X\ a$ expresses that a has to be true in the next state. U is the *until* operator, where $a\ U\ b$ means that a has to hold from the current state up to a state where b is true. Other operators can be expressed with the operators U and X. E.g., $G\ x$ is a shorthand for $\neg(true\ U\ \neg x)$ and requires x to be true at all times. The syntax of LTL is given as follows, with $a \in$ AP: $\phi ::= a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid X\phi \mid \phi_1 U \phi_2 \mid G\phi$.

If the model-checker determines that a model M violates a property ϕ then it returns a trace that illustrates the property violation. The trace is a finite prefix of an execution sequence of the model (path):

Definition 2.2 [Path] A path $p := \langle s_0, ...s_n \rangle$ of Kripke structure M is a finite or infinite sequence such that $\forall\ 0 \leq i < n : (s_i, s_{i+1}) \in T$ for M.

A test-case t is a finite prefix of a path p. We consider such test-cases where the expected correct output is included. This kind of test-cases is referred to as *passing* or *positive* test-cases. The result of the test-case generation is a *test-suite*. As test-cases created by model-checkers are linear sequences, they cannot account for non-deterministic behavior of the system under test. We therefore restrict the presented results to deterministic models. The main application area of model-checker based testing is reactive systems, where inputs are processed in a cyclic way and output values are set accordingly.

Definition 2.3 [Invalid Test-Case] A test-case t for model $M = (S, s_0, T, L)$ is *invalid* for the altered model $M' = (S', s'_0, T', L')$, if any of the following conditions is true:

$$\exists\, i : <\ldots, s_i, s_{i+1}, \ldots>= t \wedge (s_i, s_{i+1}) \notin T' \tag{1}$$
$$\exists\, i : <\ldots, s_i, \ldots>= t \wedge L(s_i) \neq L'(s_i) \tag{2}$$
$$\exists\, i : <\ldots, s_i, \ldots>= t \wedge (s_i \notin S') \tag{3}$$

In practice, the Kripke structure is described with the input language of the model-checker in use. Such input languages usually describe the transition relation by defining conditions on AP, and setting the values of variables according to these conditions. A transition condition C describes a set of states S_i where C is fulfilled. In all successor states of these states the variable v has to have the next value n: $\forall s \in S_i : L(s) \models C \wedge \forall s' : (s, s') \in T \rightarrow {}"v = n" \in L(s')$. In this paper, we use the syntax of the model-checker NuSMV [6]. Listing 1 shows how a transition relation looks like in NuSMV models. A change in the Kripke structure is represented by a syntactical change in the model source. Such changes can be automatically detected, e.g., by a comparison of the syntax trees. We are only interested in changes that do not invalidate a complete test-suite. Traces created by a model-checker consist only of states s such that for each variable v defined in the model source there exists the proposition $"v = n" \in L(s)$, where n is the current value of variable v in state s. For example, addition or removal of a variable in the model source would result in a change of L for every state in S, and would therefore invalidate any test-suite created before the change. Consequently, the interesting types of changes are those applied to the transition conditions or the values for the next states of variables in the model description.

```
ASSIGN
  next(var) := case
    condition1: next_value1;
    condition2: next_value2;
  esac;
```

Listing 1: ASSIGN section of an SMV file.

```
init(x):= 1;
next(x):= case
  State = 0: 0;
  State = 1: 1;
  1: x;
esac;
```

Listing 2: Transition relation of $t := \langle (x = 1), (x = 0), (x = 1) \rangle$.

2.2 Identifying Obsolete Test-Cases

In order to use a model-checker to decide if a test-case is valid for a given model, the test-case is converted to a verifiable model. The transition relations of all variables are given such that they depend on a special state counting variable, as suggested by Ammann and Black [1]. An example transition relation is modeled in Listing 2, where `State` denotes the state counting variable. There are two methods to decide whether a test-case is still valid after a model change. One is based on an execution of the test-case on the model, and the other verifies change related properties on the test-case model.

Symbolic Test-case Execution:

A model-checker is not strictly necessary for symbolic execution. In a scenario of model-checker based test-case generation, however, the possibility to use a model-checker is convenient, as it avoids the need for an executable model. Symbolic execution of a test-case with a model-checker is done by adding the actual model as a sub-model instantiated in the test-case model. As input variables of the sub-model the values of the test-case are used. Finally, by verifying a property that claims that the output values of the test-case and the sub-model equal for the length of the test-case, the model-checker determines whether this is indeed the case:

```
MODULE changed_model(input variables)
Transition relation of changed model

MODULE main
test-case model
VAR
  SubModel: changed_model(input vars);

SPEC G(State < max_state -> output = SubModel.output)
```

Listing 3: Symbolic execution of a test-case.

Now the problem of checking the validity of a test-suite with regard to a changed model reduces to model-checking each of the test-cases combined with the new model. Each test-case that results in a counter example is obsolete. The test-cases that do not result in a counter example are still valid, and thus are not affected by the model change. A drawback of this approach is that the actual model is involved in model-checking. If the model is complex, this can have a severe impact on the performance.

Change Properties:

In many cases, test-case models can simply be checked against certain properties in order to determine whether a change has an influence on a test-case's validity. This avoids the inclusion of the new model in the model-checking process. If a transition condition or target is changed, then the changed transition can be represented as a temporal logic property, such that any test-case model that is valid on the new model has to fulfill the property:

$$G(changed\_condition \rightarrow X\ variable = changed\_value)$$

Such change properties can be created automatically from the model-checker model source file. The concrete method depends on the syntax used by the model-checker.

If a variable transition is removed, it can only be determined whether a test-case takes the old transition using a negated property:

$$G(old\_condition \rightarrow X\ \neg(variable = old\_value))$$

Any test-case that takes the old transition results in a counter example.

Theoretically, the latter case can report false positives, if the removed transition is subsumed or replaced by another transition that behaves identically. This is conceivable as a result of manual model editing. Such false positives can be avoided by checking the new model against this change property. Only if this results in a counter example the removal has an effect and really needs to be checked on test-cases. Although verification using the full model is necessary, it only has to be done once in contrast to the symbolic execution method.

Test-cases that are invalidated by a model change can be useful when testing an implementation with regard to the model change. Obsolete positive test-cases can be used as negative regression test-cases. A negative test-case is such a test-case that may not be passed by a correct implementation. Therefore, an implementation that passes a negative regression test-case adheres to the behavior described by the old model.

2.3 Creating New Test-Cases

Once the obsolete test-cases after a model change have been identified and discarded, the test-cases that remain are those that exercise only unchanged behavior. This means that any new behavior added through the change is not tested. Therefore, new test-cases have to be created.

Adapting Obsolete Test-Cases:

Analysis of the old test-suite identifies test-cases that contain behavior that has been changed. New test-cases can be created by executing these test-cases on the changed model, recording the new behavior. This is done with a model-checker by combining test-case model and changed model together as described in Section 2.2. The test-case model contains a state counter `State`, and a maximum value `MAX`. The model-checker is queried with the property $G(\texttt{State}\neg\texttt{MAX})$. This achieves a trace where the value of `State` is increased up to `MAX`. The adapted test-case simply consists of the value assignments of the changed model in that trace.

Alternatively, when checking test-cases using the symbolic execution method, the counter examples in this process can directly be used as test-cases. In contrast to the method just described resulting test-cases can potentially be shorter, depending on the change. This can theoretically have a negative influence on the overall coverage of the new test-suite.

The drawback of this approach is that the changed model might contain new behavior which cannot be covered if there are no related obsolete test-cases. In the evaluation we refer to this method as *Adaptation*.

Selectively Creating Test-Cases:

Xu et al. [13] presented an approach to regression testing with model-checkers, where a special comparator creates trap properties from two versions of a model. In general, trap property based approaches to test-case generation express the items that make up a coverage criterion as properties that claim the items cannot be reached [9]. For example, a trap property might claim that a certain state or transition is never reached. When checking a model against a trap property the model-checker returns a counter example that can be used as a test-case. We generalize the approach of Xu et al. in order to be applicable to a broader range of test-case generation techniques. The majority of approaches works by either creating a set of trap properties or by creating mutants of the model.

For all approaches using trap properties we simply calculate the difference of the sets of trap properties, as an alternative to requiring a special comparator for a specific specification language and coverage criterion. The original model results in a set of properties P, and the changed model results in P'. New test-cases are created by model-checking the changed model against all properties in $P' - P$. The calculation of the set difference does not require any adaptation of given test-case generation frameworks. In addition, it also applies to methods that are not based on coverage criteria, e.g., the approach proposed by Black [4]. Here, properties are generated by "reflecting" the transition relation of the SMV source file as properties similar to change properties presented in Section 2.2. The resulting properties are mutated, and the mutants serve as trap properties.

It is conceivable that this approach might not guarantee achievement of a certain coverage criterion, because for some coverable items the related test-cases are invalidated, even though the item itself is not affected by the change. If maximum coverage of some criterion is required, then an alternative solution would be to model-check the test-case models against the set of trap properties for the new model instead of selecting the set difference. For reasons of simplicity, we consider the straight forward approach of using set differences in this paper. In the evaluation we refer to this method as *Update*.

The second category of test-case generation approaches uses mutants of the model to create test-cases (e.g., [3, 2, 11, 8]). For example, state machine duplication [11] combines original and mutant model so that they share the same input variables. The model-checker is then queried whether there exists a state where the output values of model and mutant differ. Here, the solution is to use only those mutants that are related to the model change. For this, the locations of the changes are determined (e.g., in the syntax tree created by parsing the models) and then the full set of mutants for the changed model is filtered such that only mutants of changed statements in the NuSMV source remain. Test-case generation is then performed only using the remaining mutants.

Testing with Focus on Model Changes

As a third method to create change related test-cases we propose a generic extension applicable to any test-case generation method. It rewrites both the model (or mutants thereof) and properties involved in the test-case generation just before the model-checker is called. This rewriting is fully automated. The model is extended by a new Boolean variable `changed`. If there is more than one change, then there is one variable for each change: `change`i. These variables are initialized with the value false. A change variable is set to true when a state is reached where a changed transition is taken. Once a change variable is true, it keeps that value. The transition relation of the change variable consists of the transition condition of the changed variable is shown in Listing 4.

```
MODULE main
VAR
  changed: boolean;
...
ASSIGN
  init(changed) := FALSE;
  next(changed) := case
    changed_condition: TRUE;
    1: changed; -- default branch
  esac;
  next(changed_var) := case
    changed_condition: changed_value;
...
```

Listing 4: Transition relation with special variable indicating changes.

The properties involved in the test-case generation approach are rewritten in order to create test-cases with focus on the model change. As an example we use LTL, although the transformation can also be applied to computation tree logic (CTL) [7].

Definition 2.4 [Change Transformation] The change transformation $\phi' = \alpha(\phi, c)$ for an LTL property ϕ with respect to the change identified with the Boolean variable c, with $a \in AP$ being a propositional formula, is recursively defined as:

$$\alpha(a, c) = a \tag{4}$$

$$\alpha(\neg\phi, c) = \neg\alpha(\phi, c) \tag{5}$$

$$\alpha(\phi_1 \wedge \phi_2, c) = \alpha(\phi_1, c) \wedge \alpha(\phi_2, c) \tag{6}$$

$$\alpha(X\phi, c) = X(c \rightarrow \alpha(\phi, c)) \tag{7}$$

$$\alpha(\phi_1 U \phi_2, c) = \alpha(\phi_1, c)\ U\ (c \rightarrow \alpha(\phi_2, c)) \tag{8}$$

Basically, all temporal operators are rewritten to include an implication on the change variable. This achieves that only such counter examples are created that include the changed transition. For multiple changes there has to be one modified version of each property for each change in order to make sure that all changes are equally tested. In the evaluation we refer to this method as *Focus*.

3 Empirical Results

The previous section presented different possibilities for different aims to cope with model changes in a scenario of model-checker based test-case generation. This section tries to evaluate the feasibility of these ideas. First, the experiments conducted are described, and then the results are presented and discussed.

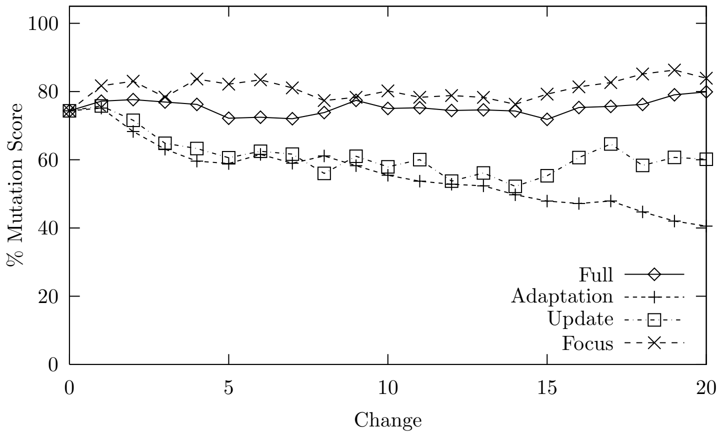
3.1 Experiment Setup

The methods described in this paper have been implemented using the programming language Python and the model-checker NuSMV [6]. All experiments have been run on a PC with Intel Core Duo T2400 processor and 1GB RAM, running GNU/Linux. We automatically identify changes between two versions of a model by an analysis of the abstract syntax trees created from parsing the models. We use a simple example model of a cruise control application based on a version by Kirby et al. [10]. In order to evaluate the presented methods, the mutation score and creation time of new and updated test-suites were tracked over several changes. There is a threat to the validity of the experiments by choosing changes that are not representative of real changes. Therefore the experiments were run several times with different changes and the resulting values are averaged.

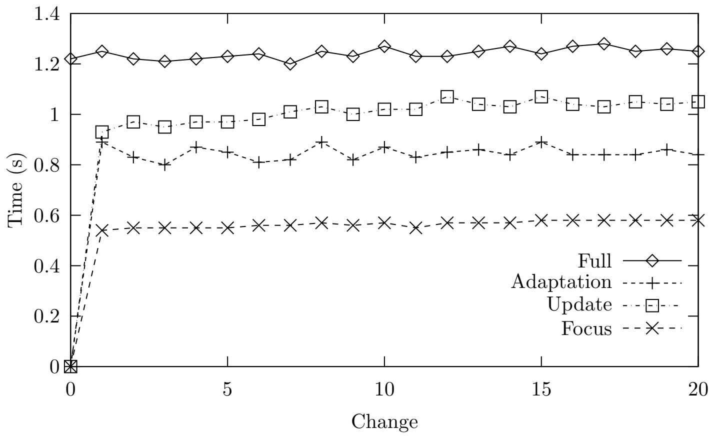
In the first step, mutants were created from the supposedly correct model. The following mutation operators were used (see [5] for details): STA (replace atomic propositions with true/false), SNO (negate atomic propositions), MCO (remove atomic propositions), LRO, RRO, ARO (logical, relational and arithmetical operator replacement, respectively). The resulting mutants were analyzed in order to eliminate equivalent mutants. This is done with a variant of the state machine duplication approach [11], where the model-checker is queried whether there exists a state where the output values of a model and its mutant differ. An equivalent mutant is detected if no counter example is returned. The set of mutants was further reduced by checking each mutant against a set of basic properties that require some elementary behavior, e.g., reachability of some important states.

Out of the resulting set of inequivalent mutants one mutant is chosen randomly, and used as new model. With this mutant, the procedure is repeated until a sequence of 20 visible model changes is achieved. The experiment was run on 20 such sequences and the results were averaged.

For each of the sequences of model versions the following is performed: Beginning with the model containing all 20 changes, test-suites are created using the methods transition coverage criterion (one test-case for each transition condition of the NuSMV model), mutation of reflected transition relation [4] and state machine duplication [11]. These three methods were chosen as they should be representative for most types of conceivable approaches. Then, the next version of the model is chosen, and the test-suites of the previous model are analyzed for obsolete test-cases, and new and updated test-suites are created. Then the mutation scores of all of these test-suites are calculated. The mutation score is the ratio of identified mutants to mutants in total. It is calculated by symbolically executing the test-case models against the mutant models. This procedure is repeated for each of the model versions up to the original model.



(a) Mutation score



(b) Creation time

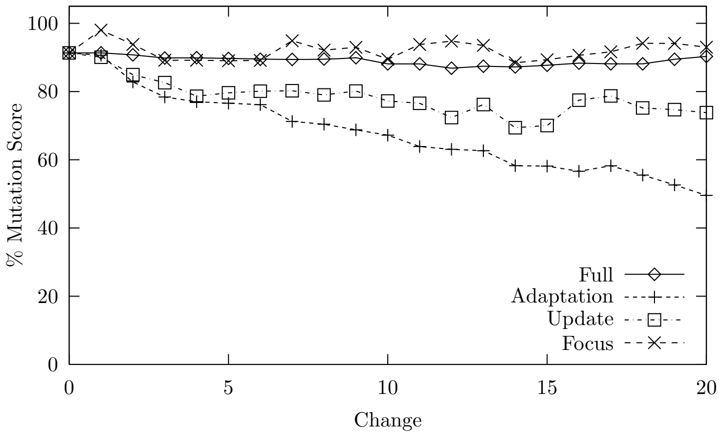
Fig. 1. Transition coverage test-case generation.

Table 1
Average number of unique new test-cases.

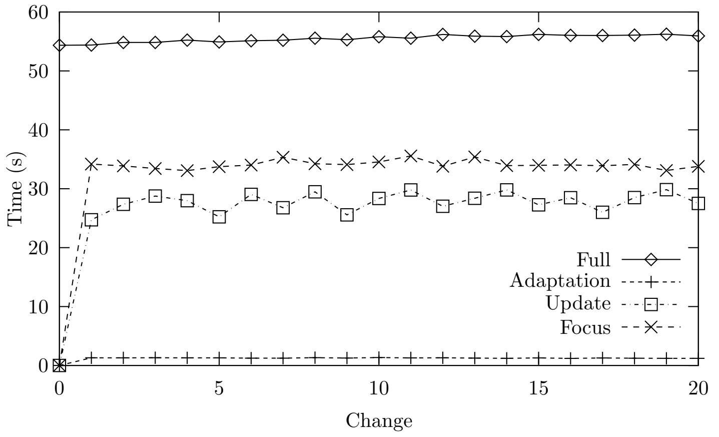
Test-Suite Type	Full	Adaptation	Update	Focus
Transition	5.75	1	1.4	6.6
Reflection	19.35	2.5	7.05	24.4
SM Duplication	33.35	2.85	6.45	29.4

3.2 Results

Figures 1-3(a) show the mutation scores of the different methods along the course of the different model version. There is a degradation of the mutation score for the *adaptation* and *update* methods. The degradation increases with each model change, therefore it could be advisable to create new test-suites after a certain number of changes when using such a method. In contrast, the change *focus* method achieves a mutation score that is sometimes even higher than that of a completely new test-



(a) Mutation score

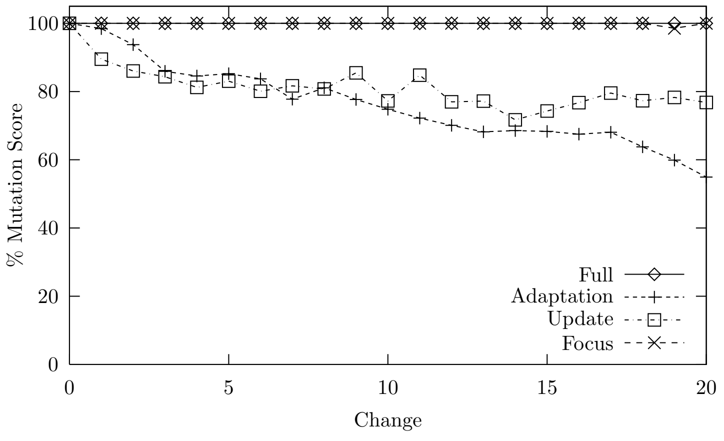


(b) Creation time

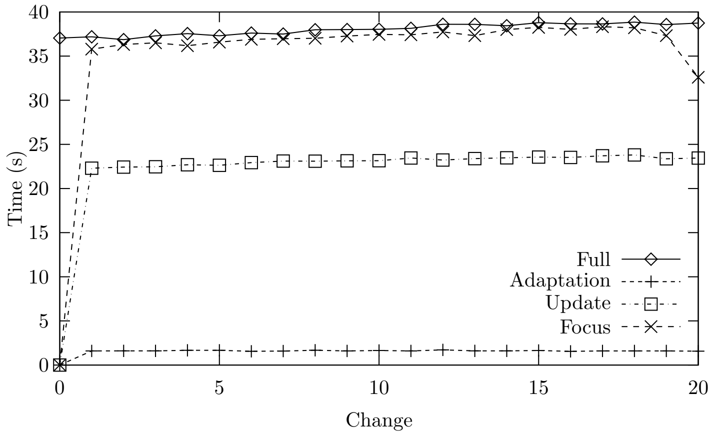
Fig. 2. Mutation of the reflected transition relation.

suite. This is because the test-suites created with the *focus* method are bigger than new test-suites for the transition coverage criterion. *Adaptation* generally achieves the lowest mutation scores. However, the mutation score is only slightly smaller than for the *update* method, so a significant performance gain could justify this degradation.

Figures 1-3(b) show the computation times for creating and for updating test-suites. All methods are faster than a complete test-case generation process. Test-suite *adaptation* performs significantly faster than all other methods in most cases. The performance of the adaptation is determined by the model complexity and number of invalid test-cases, and is therefore similar for all test-suites in the experiment. In contrast, the performance of the *update* and *focus* methods depends on the test-case generation approach they are based upon. For simple methods like transition coverage *focus* is very efficient, while there is less performance gain as test-case generation complexity increases. For the most complex approach used in the experiment (based on state machine duplication) the performance gain in



(a) Mutation score



(b) Creation time

Fig. 3. Test-case generation via state machine duplication.

comparison to creating a new test-suite is minimal.

Finally, Table 1 compares the numbers of new test-cases created in average after each model change. This chart reveals why the change *focus* method achieves such high mutation scores: the number of test-cases generated is significantly higher than for any other approach. Interestingly it is even higher than the number of test-cases generated for new test-suites with the transition and reflection approaches, although the test-case generation is still faster in average.

4 Conclusion

In this paper, we have shown how to decide whether a test-case is still valid after the model it was created from is changed. That way, it is possible to reuse some of the test-cases after a model change and reduce the test-suite generation effort. Different methods to create test-cases specific to the model change were presented. We used the model-checker NuSMV for our experiments and as an example model syntax. However, there is no reason why the approach should not be applicable to other model-checkers. Experiments have shown that the presented methods can be used to update test-suites after a model change, although there is a trade-off between performance improvement and quality loss.

The main problem of model-checker based approaches in general is the performance. If the model is too complex, then test-case generation will take very long or might even be impossible. Therefore, it is important to find ways of optimizing the approach. The potential savings when recreating test-suites after a model change are significant. Even for the small model used in our evaluation a large performance gain is observable when only selectively creating test-cases for the model changes. Although a model is usually more abstract than the program it represents, the model size can still be significant. For instance, automatic conversion (e.g., Matlab Stateflow to SMV) can result in complex models.

The methods presented to create new test-cases with minimal computational effort achieve good results. We cannot conclude that one method is superior, because the preferable method depends on the concrete scenario. If the main goal is to minimize the costs of retesting, then adaptation of old test-cases is effective as long as there are not too many and significant changes. If it is more important to maximize likelihood of detecting faults with relation to the change, then the presented method to create test-cases focusing on a change is preferable. For example, in safety related scenarios a decrease of the test-suite quality is unacceptable. Finally, the update method that creates test-cases only for changed parts seems like a good compromise; it reduces the costs while the quality decrease is not too drastic. Test-cases created with any of the presented methods can be used as regression test-suites, following the ideas of Xu et al. [13].

There are some approaches that explicitly use specification properties for test-case generation [3, 11, 2]. This paper did not explicitly cover the aspects of test-suite update with regard to specification properties. However, the idea of test-suite focus directly applies to such approaches, as well as the presented test-suite update techniques. The cruise control example is only a small model, and the changes involved in our experiments were generated automatically. This is sufficient to show the feasibility of the approach. However, actual performance measurements on more complex models and realistic changes would be desirable.

References

[1] Ammann, P. and P. E. Black, *A Specification-Based Coverage Metric to Evaluate Test Sets.*, in: *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering* (1999), pp. 239–248.

[2] Ammann, P., W. Ding and D. Xu, *Using a Model Checker to Test Safety Properties*, in: *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS 2001)* (2001), pp. 212–221.

[3] Ammann, P. E., 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)* (1998), pp. 46–54.

[4] Black, P. E., *Modeling and Marshaling: Making Tests From Model Checker Counterexamples*, in: *Proc. of the 19th Digital Avionics Systems Conference*, 2000, pp. 1.B.3–1–1.B.3–6 vol.1.

[5] Black, P. E., V. Okun and Y. Yesha, *Mutation Operators for Specifications*, in: *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering* (2000), pp. 81–88.

[6] Cimatti, A., E. M. Clarke, F. Giunchiglia and M. Roveri, *NUSMV: A New Symbolic Model Verifier*, in: *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification* (1999), pp. 495–499.

[7] Clarke, E. M. and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, in: *Logic of Programs, Workshop* (1982), pp. 52–71.

[8] Fraser, G. and F. Wotawa, *Property Relevant Software Testing with Model-Checkers*, ACM SIGSOFT Software Engineering Notes **31** (2006), pp. 1–10.

[9] Gargantini, A. and C. Heitmeyer, *Using Model Checking to Generate Tests From Requirements Specifications*, in: *ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lecture Notes in Computer Science **1687** (1999), pp. 146–162.

[10] Kirby, J., *Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System*, Technical Report TR-87-07, Wang Institute of Graduate Studies (1987).

[11] Okun, V., P. E. Black and Y. Yesha, *Testing with Model Checker: Insuring Fault Visibility*, in: N. E. Mastorakis and P. Ekel, editors, *Proceedings of 2002 WSEAS International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems*, 2003, pp. 1351–1356.

[12] Pnueli, A., *The temporal logic of programs*, in: *FOCS'77: 18th Annual Symposium on Foundations of Computer Science* (1977), pp. 46–57.

[13] Xu, L., M. Dias and D. Richardson, *Generating regression tests via model checking*, in: *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)* (2004), pp. 336–341.