

Creating Test-Cases Incrementally with Model-Checkers

Gordon Fraser and Franz Wotawa*
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz, Austria
{fraser,wotawa}@ist.tugraz.at

Abstract: Test-case generation with model-checkers is a promising field of research in software testing. Model-checker based approaches offer many advantages: They are fully automated, they are flexible due to different concrete techniques, and under certain conditions they are also efficient. There are still many issues that need to be resolved in order to achieve widespread acceptance in the industry. Because model-checkers were not originally designed with test-case generation in mind, a large percentage of the test-cases produced are duplicates. Many of the remaining test-cases share identical prefixes that do not contribute to the overall fault sensitivity of a test-suite. Some test criteria also result in large test-suites of rather short test-cases. In this paper, we address these problems and suggest to create test-cases incrementally instead of separately for each test requirement. For this, heuristics based on an estimated distance between a state and a temporal logic formula are presented, which allows to choose which test-case to extend with regard to which test requirement.

1 Introduction

Testing with model-checkers has been presented as one possible way to automate test-case generation. The idea is promising: Test-cases are created in a fully automated fashion, a range of different proposed techniques allows variation of the amount of test-cases produced and the objective of testing. Under certain conditions such as a limited model size, testing with model-checkers is very efficient.

Model-checkers, however, were not originally intended for test-case generation but for verification purposes. The demands on a verification tool are different than on a testing tool. As a consequence, the efficiency of the test-case generation and the efficiency of resulting test-suites are not as high as they could be. When creating test-cases with a model-checker, many produced counterexamples are identical. Test-cases usually contain a significant amount of redundancy; that is, they share identical prefixes that do not contribute to the overall fault sensitivity [FW07b].

*The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

Another drawback is that the size of a resulting test-suite and the length of the test-cases are difficult to influence. For example, some test criteria result in large test-suites of very short test-cases that are ineffective at detecting faults [HRV⁺03].

In this paper, we propose to create test-cases incrementally instead of using a distinct test-case for each test requirement. This allows to influence the test-suite size and test-case length, and reduces redundancy. The main contribution of this paper is that the choice of which test-case to extend is made heuristically using a distance measurement between states and temporal logic formulas.

2 Testing with Model-Checkers

A model-checker is a tool for formal verification. It takes as input an automaton based model of an application and a property specified in temporal logics, and then effectively explores the entire state space of the model. If a property violation is detected, then a counterexample is produced, which is usually given as a linear sequence beginning with an initial state of the system and illustrating the property violation.

For testing purposes, counterexamples are interpreted as test-cases. The state of a model is represented as the set of propositions that hold; consequently, each state of a counterexample also consists of a set of propositions. A suitable test-case execution framework can extract from this the test data and also the expected results (i.e., test oracle).

There are different approaches of how to systematically derive counterexamples. One way is to formulate test requirements as temporal logic properties in a negated way (*trap properties* [GH99]), such that model-checking results in a counterexample for each test requirement (e.g., [GH99,RH01]). For example, a trap property might claim that a certain state or transition is never reached. A resulting counterexample shows how the state or transition described by the trap property is reached. An alternative approach is to mutate the model such that it is inconsistent with a given specification (e.g., [ABM98]). In this paper, we focus on trap property based approaches, although extension to mutation based approaches is conceivable by representing mutants as characteristic properties [FW07a].

In general, the model-checker is called sequentially for each trap property. This approach has several problems:

- A trap property might already be covered by another test-case at the time it is model-checked. As performance of model-checkers is problematic, checking such properties should be avoided if time is important.
- Different trap properties can result in identical test-cases, which do contribute to the overall fault sensitivity of the test-suite.
- Different trap properties can result in similar test-cases, which increases redundancy [FW07b]. This also has a negative effect on the overall fault sensitivity.

3 Generating Test-Cases Incrementally

An improved method to test-case generation was proposed by Hamon et al. [HdMR04]. In their approach, each counterexample is created as an extension to the previous counterexample, until a certain maximum length is reached. Using this approach, duplicate test-cases are mostly avoided (although theoretically still possible). The drawback of this approach is that test-suites can become unnecessarily large: For one, already covered trap properties are not detected. Second, the choice of the next trap property to use for extension has an influence on the overall length of the test-cases.

In a recent paper [FW07c], we suggested the use of temporal logic rewriting rules to detect trap properties that are already covered by other counterexamples before calling the model-checker. Rewriting techniques from runtime verification are used to detect, whether a given counterexample shows a property violation for other trap properties. We also showed how to apply this technique to mutation based approaches in [FW07a]. This approach also allows to extend counterexamples instead of creating distinct counterexamples for each test requirement. An issue that was not resolved in these works is that the order in which trap properties are evaluated and counterexamples are extended has an influence on the overall size of a test-suite.

The size of a test-suite is important when it comes to test-case execution. The more test-cases have to be executed to achieve a certain test objective, and the longer these test-cases are, the higher the costs. This is especially important with regard to regression testing.

A partial solution was presented in [FW07c], where the results of the formula rewriting are used to identify opportunities for counterexample extension. In general, rewriting is used to detect whether a property is violated by a trace, or in the context of testing, covered by a test-case. This is the case if a formula can be rewritten to a contradiction. If the rewriting changes the formula but does not result in a contradiction, then this indicates that the counterexample affects the formula and there exists a (possibly short) extension to the counterexample to cover the trap property. In many cases, the rewriting will not change a formula, and therefore not deliver any suggestions for possible extensions. Consequently, a different or additional method is necessary to decide whether to extend a counterexample, and which trap property to use for the extension.

We propose to heuristically decide which trap property would result in the shortest extension to a given counterexample. The heuristic needs an estimate of the distance between a state of a model and a temporal logic formula. Given such an estimate, a test-case can be extended with the shortest possible extension for the remaining trap properties, thus minimizing the overall test-suite size.

This problem is related to directed model-checking [ELL01], where heuristic search is used to avoid the expansion of the complete state space when looking for a counterexample. Heuristic search is easiest possible when checking safety properties, that is, properties that describe behavior that has to be fulfilled at all states. Unfortunately, most trap properties used to generate test-cases are not simple safety properties. Simple heuristic search does not directly apply to such properties; heuristic search can only be applied by considering the details of the model-checking algorithm.

Properties are specified using temporal logics. In this paper, we use future time Linear Temporal Logic (LTL) [Pnu77]. An LTL formula consists of atomic propositions, Boolean operators and temporal operators. The operator " \circ " refers to the *next* state. E.g., " $\circ a$ " expresses that a has to be true in the next state. " \mathcal{U} " is the *until* operator, where " $a \mathcal{U} b$ " means that a has to hold from the current state up to a state where b is true. " \square " is the *always* operator, stating that a condition has to hold at all states of a trace, and " \diamond " is the *eventually* operator that requires a condition to eventually hold at some time in the future.

An LTL formula describes a sequence in time. Most trap properties are of the type $\square(x \rightarrow \circ \neg y)$ or $\square(x \rightarrow \diamond \neg y)$, therefore we need to estimate the distance to the first state of such a sequence, where x holds. A possible approach to heuristically estimate the distance for a trap property is by using a formula-based estimate, similar to the heuristic used in directed model-checking [ELL01], but also considering temporal operators. Let s denote the currently considered state; $h(f, s)$ is a heuristic function that estimates the distance from state s to a state that fulfills property f , and $\bar{h}(f, s)$ is an estimate for the distance from s to a state where f is violated. The definition of h is given recursively below, and the definition of \bar{h} can be made analogously. OP denotes any temporal operator, a represents an atomic proposition, and f and g denote temporal logic formulas.

$$\begin{aligned}
h(f \rightarrow \text{OP } g, s) &= h(f, s) & h(a, s) &= \begin{cases} 0 & \text{if } a \text{ is true in } s \\ 1 & \text{if } a \text{ is false in } s \end{cases} \\
h(\square f, s) &= h(f, s) & h(\neg f, s) &= \bar{h}(f, s) \\
h(\circ f, s) &= 0 & h(f \wedge g, s) &= h(f, s) + h(g, s) \\
h(\diamond f, s) &= 0 & h(f \vee g, s) &= \min\{h(f, s), h(g, s)\} \\
h(f_1 \mathcal{U} f_2, s) &= h(f_1, s)
\end{aligned}$$

As an alternative, we can extract state information from the properties to apply other, possibly more flexible heuristics. In model-checkers, models are defined over variables $\{x_0, x_1, \dots, x_n\}$, therefore all propositions refer to these variables. In this paper, we assume that a system is described only by Boolean variables. It is conceivable to extend this approach to include enumerated and numerical data types, either by adapting the heuristics to these data types, or by applying the heuristics to the Boolean representation used internally in the model-checker. First, we extract all atomic propositions that have to hold in the initial state of a counterexample sequence, and all those propositions that have to hold in all states. For the following definition, we assume that a formula is given in negation normal form, i.e., negations occur only directly in front of atomic propositions. The extraction of a set of propositions ϕ is defined recursively on the structure of the formula, where a represents an atomic proposition, and f and g temporal logic formulas:

$$\begin{aligned}
\phi(f \rightarrow \text{OP } g) &= \phi(f) & \phi(a) &= \{a\} \\
\phi(\square f) &= \phi(f) & \phi(\neg a) &= \{\neg a\} \\
\phi(\circ f) &= \{\} & \phi(f \wedge g) &= \phi(f) \cup \phi(g) \\
\phi(\diamond f) &= \{\} & \phi(f \vee g) &= \phi(f) \cup \phi(g) \\
\phi(f_1 \mathcal{U} f_2) &= \phi(f_1)
\end{aligned}$$

From the set of propositions $\phi(f)$ we can extract a Boolean vector $\vec{b} = \{b_0, b_1, \dots, b_n\}$

with value assignments for variables $\{x_0, x_1, \dots, x_n\}$. The values in \vec{b} make all propositions in $\phi(f)$ true; that is, the extraction of \vec{b} is a simple application of constraint solving. Assume further a vector $\vec{s} = \{s_0, s_1, \dots, s_n\}$ with value assignments for variables $\{x_0, x_1, \dots, x_n\}$ according to a given state s , such as the final state of a test-case. Now we can calculate an estimate for the distance between \vec{b} and \vec{s} , for example using the Hamming distance measurement, which estimates the distance as the number of bit flips between the two vectors. Obviously, if other datatypes are used, then different heuristics are necessary.

Table 1 shows a simple example, where the state is defined by Boolean variables x and y , and a simple trap property $f = \square(x \wedge y \rightarrow \bigcirc \neg y)$. For this formula, $\phi(f) = \{x, y\}$. The estimates are given for the formula rewriting and the Hamming distance, and are identical for this trivial example.

Table 1: Distance estimates for s defined by variables x and y , and $f = \square(x \wedge y \rightarrow \bigcirc \neg y)$.

x	y	$h(f, s)$	Hamming
0	0	2	2
0	1	1	1
1	0	1	1
1	1	0	0

This distance measurement can be integrated into the test-case generation to heuristically decide which test-case to extend, or which trap property to choose. Each time a counterexample is created, the distances between the final state of the counterexample to all remaining trap properties are calculated. The trap property with the smallest distance estimate is chosen as the next trap property. The initial state of the model has to be set to the last state of the previous counterexample, and then the chosen trap property is checked. The resulting new counterexample is an extension to the previous counterexample. Extension can be limited with a predefined maximum length.

The applicability of the presented distance estimates depends on the structure of the trap properties. A property of the form $\diamond f$, where everything is within the scope of a future time operator would yield no propositions for an estimation method, and would therefore require a modified approach. Luckily, most trap properties are of the type $\square(f \rightarrow \bigcirc \neg g)$.

4 Conclusions

In this paper, we have proposed to gradually extend test-cases instead of creating one distinct test-case for each counterexample that results from a test requirement. This idea has been introduced in [HdMR04] and [FW07c], but these solutions can be further optimized. The choice of which test-case to extend with a new counterexample has an influence on the overall size of the resulting test-suite. We have presented a method that allows to heuristically estimate the distance between a state and a temporal logic formula.

We have suggested methods to heuristically estimate the distance to trap properties, either by applying the heuristic directly to LTL formulas, or by deriving Boolean vectors from

formulas. As an example heuristic, we calculate a Hamming distance between the Boolean vectors representing a state and the initial state of a formula. Extension to systems with variables of deliberate types should be straight forward. Furthermore, it is conceivable to apply different heuristics directly on the propositions that represent a state.

References

- [ABM98] Paul E. Ammann, Paul E. Black, and William Majurski. Using Model Checking to Generate Tests from Specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, 1998.
- [ELL01] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 57–79, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
- [FW07a] Gordon Fraser and Franz Wotawa. Mutant Minimization for Model-Checker Based Test-Case Generation. In *Proceedings of the Third Workshop on Mutation Analysis (Mutation 2007)*, 2007. To appear.
- [FW07b] Gordon Fraser and Franz Wotawa. Redundancy Based Test-Suite Reduction. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2007.
- [FW07c] Gordon Fraser and Franz Wotawa. Using LTL Rewriting to Improve the Performance of Model-Checker Based Test-Case Generation. In *Proceedings of the Third Workshop on Advances in Model Based Testing (A-MOST 2007)*, 2007. To appear.
- [GH99] Angelo Gargantini and Constance 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*, volume 1687, pages 146–162, Toulouse, France, September 1999. Springer.
- [HdMR04] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating Efficient Test Sets with a Model Checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 261–270, 2004.
- [HRV⁺03] Mats P.E. Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-Generating Test Sequences using Model Checkers: A Case Study. In *Third International International Workshop on Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 42–59. Springer, October 2003.
- [Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA*, pages 46–57. IEEE, 1977.
- [RH01] Sanjai Rayadurgam and Mats P. E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91, Washington, DC, April 2001. IEEE Computer Society.