Optimising Test Execution Times in Test Suite Generation

Tilo Mücke and Michaela Huhn
{muecke, huhn}@ips.cs.tu-bs.de

Abstract: In the last decade, enormous progress has been made in generating test cases for coverage criteria automatically via model checking. However, many approaches suffer from generating large test suites with long test execution times.

In recent work, we developed a method to generate test suites from state-based design models with the shortest test execution time satisfying a given set of coverage criteria. We were able to show that valuable improvements can be achieved. But space complexity of generating the fastest test suite turned out to be too high; thus our approach was limited to small models.

Alternatively, a model checker can be called several times, each run generating a single *test case*. Thereby, the space complexity problem is solved. To build a test suite with minimised execution time, optimisations have to be applied afterwards. Here we present and compare several such algorithms to optimise test suites with respect to execution time.

1 Introduction

To reduce the costs of software testing, automation is applied more and more. Automation of the test execution is a straight forward idea and is utilised in most of todays software engineering projects. Automated test case generation, though, needs a formal description of the properties of the software, a metric for test quality and a method to generate the test cases. Many of these methods, generating tests randomly as well as driven by a model checker, suffer from generating huge test suites with long test execution times. To overcome this problem, we developed a method to generate time optimal test suites [MH04] for certain coverage criteria via model checking. Unfortunately, our method suffers from space complexity problems caused by instrumenting the model for test case generation. Thus the approach is limited to models some magnitudes smaller than those which could be handled if model checking is employed for verification. However, experiments provided us with the information that optimal test suites tend to consist of only a few but long test cases.

Here we present an alternative approach where we split the generation of optimised test suites into a generation phase and an optimisation phase. In the first phase, a test suite larger than necessary to achieve a given coverage criterion is created, e.g. by querying a model checker. In the optimisation phase a minimal set cover algorithm is employed to minimise the test suite but keeping the coverage. In the following we will discuss and compare several variants to implement both phases. Preliminary results on randomly generated models are promising.

2 Test Case Generation

We reconsider the work of [HLSC01] who generate single traces (test cases) for each partial coverage [1] contributing to a given coverage criterion and don't allow resets within a trace. The number of partial coverages ($\sharp pc$) gives the number of situations (e.g. states, transitions) in a model contributing to a coverage criterion. The model checker searches for a trace for a particular partial coverage, which is used as a test case. The time which is needed to execute the test case according to the timing information within the model [2] is called test execution time.

In general, some partial coverages will be achieved more than once by this approach (i.e. several states are covered by more than one test case) and it is reasonable to extend already generated test cases by searching from the last state of that test case for additional coverage. Thereby we obtain a larger set of test cases which achieves many partial coverages multiple times. By enlarging the test suite, test execution time is increased. However, if we apply a minimal set cover algorithm for optimisation to a larger test suite, in general, it will contain more subsets satisfying the coverage criterion, thus optimisation algorithms may find a better candidate, i.e., a test suite with shorter overall execution time.

Three functions are needed to generate the test suite:

$$generate : Models \times (Traces \cup \{\epsilon\}) \times \{0..\sharp pc\} \rightarrow Traces \cup \{\epsilon\}$$

This function generates a test case for a given model e.g. by using a model checker. A trace may be given which serves as a beginning of the new extended trace for a given partial coverage (3rd parameter). How to use a model checker to generate such traces is discussed e.g. in [RH01]. If the partial coverage cannot be achieved, ϵ is returned.

For each test case tc returned by the model checker, two additional informations are stored: the execution time for the test case $t_{exec}(tc)$ and the set of partial coverages $PC(tc)$ which is achieved by executing the test case.

$$search : Model \times \mathbb{N} \rightarrow \wp(Traces)$$

search generates a large set of test cases for a given model. As a second parameter, it takes the number of partial coverages which have to be achieved. It uses the function $generate$. Possible implementations of $search$ are discussed in section 3.

$$minsetcover : \wp(Traces) \times \mathbb{R} \rightarrow \wp(Traces)$$

minsetcover generates a subset of test cases, which satisfies the same partial coverages but has minimised execution time. To calculate the execution time for a complete test suite, the reset time t_{reset} of the system has to be added after each test case. Since $minsetcover$ is NP-complete [GJ79], a heuristic algorithm is needed. Several possibilities are discussed in section 4.

The complete test suite is calculated by $minsetcover(search(model, \sharp pc), t_{reset})$.

[1] A *partial coverage* describes one situation in the model where the coverage criterion applies and thus has to be covered by a test case. I.e., if state coverage shall be achieved, a partial coverage (test case) for each reachable state is required.

[2] i.e., we consider timed models

3 Search Strategies

A search strategy uses the *generate* function to generate a set of test cases. The naive search strategy generates a test case for each partial coverage:

function search1(model, ♯pc)
    testsuite=∅;
    for i=1 to ♯pc do
        testcase=generate(model, ϵ, i);
        if (testcase≠ϵ) then testsuite = testsuite ∪ {testcase};
    od;
    return testsuite;

3.1 Depth 2 Search

Depth 2 search shows, how to enlength test cases by adding partial coverages. A test case generated by the naive search strategy is enlengthened by a trace for each partial coverage:

function search2(model, ♯pc)
    testsuite=search1(model, ♯pc); addition=∅;
    foreach testcase ∈ testsuite do
        for i=1 to ♯pc do
            enlengthenedTestcase=generate(model, testcase, i);
            if (enlengthenedTestcase≠ϵ) then addition=addition∪{enlengthenedTestcase};
        od;
    od;
    return testsuite ∪ addition;

3.2 Heuristic Search

The heuristic search function enlengthens only the best test cases for a partial coverage which has been achieved rarely:

function searchH(model, ♯pc)
    testsuite=search1(model, ♯pc);
    for i=1 to comp · ♯pc do
        testcase=getBestTestcase(testsuite);
        pc=getWorstPartialCoverage(testsuite);
        enlengthenedTestcase=generate(model, testcase, pc);
        if (enlengthenedTestcase≠ ϵ) then testsuite=testsuite ∪ {enlengthenedTestcase};
    od;
    return testsuite;

The function $getBestTestcase$ returns the test case with the maximal value for $|PC(tc)|/(t_{exec}(tc) + t_{reset})$ which has not been enlengthened in all possible ways. The function $getWorstPartialCoverage$ returns the partial coverage which has been achieved least often in the test suite and in particular not in the chosen testcase. $comp$ is a parameter to control the heuristic search. In our experiments $comp = 10$ serves as a value for the amount of enlengthened test cases in proportion to the original ones.

4 Minimal Set Cover Algorithms

A minimal set cover algorithm generates a small[3] subset from a set of sets, such that the union of sets in the small subset equals the union of the sets in the original set. Minimal set cover algorithms can be used to reduce the size of test suites by searching for a subset of test cases which satisfy the same partial coverages but have a shorter execution time.

4.1 Simple Greedy Algorithm

To apply a greedy algorithm on a minimal set cover problem, the usual approach [OPV95] has to be complemented, i.e., we start with the full test suite and try to eliminate test cases but keep the same coverage. The function $entry$ is used to determine the order, in which the test cases are eliminated. For the most simple type of a greedy algorithm, we use the $entry$ function with $entry(testsuite, \ldots, i, \ldots) = i$*th test case in test suite*.

function greedy(testsuite,t_{reset}*)*
    reducedTestsuite=testsuite;
    for i=1 to |testsuite| do
        testcase=entry(testsuite, reducedTestsuite, i,t_{reset}*);*
        if (|testsuite.PC|==|(reducedTestsuite$\setminus$*testcase).PC|)*
            then reducedTestsuite=reducedTestsuite$\setminus\{$*testcase*$\}$*;*
    od;

4.2 Bidirectional Greedy Algorithm

The bidirectional greedy algorithm (compare [OPV95]) uses the same $entry$ function but is applied to the test suite twice running in both directions.

function greedyBi(testsuite,t_{reset}*)*
    testsuite=greedy(testsuite);
    reducedTestsuite=testsuite;
    for i=|testsuite| to 1 do
        testcase=entry(testsuite, reducedTestsuite, i, t_{reset}*);*
        if (|testsuite.PC|==|(reducedTestsuite$\setminus$*testcase).PC|)*
            then reducedTestsuite=reducedTestsuite$\setminus\{$*testcase*$\}$*;*
    od;

4.3 Sorted Greedy Algorithm

For further improvement of the test suite, the entry function can be modified such that the test cases are sorted by their quality, weakest test case returned first. Therefore, the $getBestTestcase$ function from the heuristic search can be reused.

[3] As stated in section 2, calculating a minimal set is NP-complete, thus heuristics are used in practical applications.

execution time (test case amount)	Naive Search		Depth-2-Search		Heuristic Search	
Without Minimisation	104.2	(100.0)	9817.2	(9000.0)	1716.4	(1000.0)
Simple Greedy Algorithm	68.5	(65.3)	68.5	(65.3)	68.5	(65.3)
BiDirectional Greedy Algorithm	51.5	(48.9)	51.5	(48.9)	51.5	(48.9)
Sorted Greedy Algorithm	52.6	(49.9)	51.6	(48.9)	5.7	(4.9)
Force Directed Algorithm	51.5	(48.9)	27.1	(25.1)	5.5	(5.1)

Table 1: Empirical Results of Test Execution Time Optimisation (Average of 10 Tests)

4.4 Force Directed Algorithm

This algorithm has been derived from the force directed scheduling algorithm [PK89]. Unlike the other greedy algorithms, the order in which the test cases can be deleted is not fixed, but it depends on which test cases are already deleted.

For the force directed algorithm, we use a new function

$$timesCovered : \{1..\sharp pc\} \times Testsuites \to \mathbb{N}$$

which calculates by how many test cases of a test suite a partial coverage is satisfied.

The entry function is now implemented to return the test case with the lowest quality according to

$$quality(tc) = \frac{\sum_{i=1}^{\sharp pc} \begin{cases} \frac{1}{timesCovered(i, reducedTestsuite)-1} & : \quad i \in PC(tc) \\ 0 & : \quad else \end{cases}}{t_{exec}(tc) + t_{reset}}$$

The quality of a test case is described as fraction of the assets of the test case as the numerator and the time needed to execute the test case and reset as a denominator. The assets are accumulated from the assets achieved in dependence of the partial coverages. If a partial coverage is not achieved, its asset is zero. If it is achieved, the asset depends on how many other test cases do also satisfy this partial coverage. If it is not achieved by any other criterion, its asset is set infinity, which maximises its quality. The more frequent a partial coverage is achieved, the smaller is the asset for satisfying it.

5 Experiments

In order to compare the search- and minsetcover-algorithms, they have been applied to 10 randomly generated example statecharts with 100 states and 1000 transitions each. Each transition is labeled with an after event, whose time parameter is distributed exponentially. No guards or actions are added. Test suites satisfying state coverage are generated.

Our results show, that the ability of optimising test suites with a short execution time is depending on a *combination* of search- and minsetcover-algorithms. Using a good search, but a poor minsetcover-algorithm, we achieved no reduction compared to the minsetcover-algorithm without a search. This is caused by the fact that the initial testcases of the depth-2 and heuristic search are the test cases which are generated during naive search. Thus, all greedy algorithms which are not changing the order in which the test cases are considered

do not benefit from the use of search-algorithms. With the best minsetcover-algorithm, we were able to reduce the execution time to approx. 50%, only. Using a combination of a good search and a good minsetcover-algorithm, we achieved a reduction to approx. 5%.

An application of the optimal generation method for this kind of example has only been possible with up to 23 states on 4GB of memory running approx. 30 minutes. Calculating each of the examples above has been possible with the use of 8MB of memory and a time consumption of 0.5 till 45 seconds, depending on the search- and minsetcover-algorithms.

6 Conclusion

We investigated the potential of optimising automatically generated test suites for given coverage criteria. We considered a combination of search algorithms that enlarge a test suite by high potential candidates to achieve a better coverage and minimal set cover algorithms to select a minimised test suite with short execution time. Experiments on randomly generated models are promising.

As in [HLSC01], the model checker is only used for generating single test cases and only reachability is an issue. Thus in contrast to other approaches, instrumentation of the model will not blow up the model in size.

First experiments on optimising test suites generated from real world models show that for transition coverage test suites with nearly the same test execution time as our previous approach [MH04] can be achieved.

References

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[HLSC01] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic Test Generation from Statecharts Using Model Checking. In *Workshop on Formal Approaches to Testing of Software (FATES)*, pages 15–30, 2001.

[Knu76] Donald E. Knuth. Big Omicron and big Omega and big Theta. *SIGACT News*, 8(2):18–24, 1976.

[MH04] Tilo Mücke and Michaela Huhn. Generation of Optimized Testsuites for UML Statecharts with Time. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2004.

[OPV95] Jeff Offutt, Jie Pan, and Jeff Voas. Procedures for Reducing the Size of Coverage-based Test Sets. In *Proceedings of the Twelfth International Conference on Testing Computer Software*, pages 111–123, 1995.

[PK89] P.G. Paulin and J.P. Knight. Force-directed Scheduling for the Behavioural Synthesis of ASICs. *IEEE Trans. on Computer-Aided Design*, 8(6):661–679, 1989.

[RH01] S. Rayadurgan and M. Heimdahl. Coverage Based Test-Case Generation using Model Checkers. In *Intl. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–93, 2001.