Redundancy Based Test-Suite Reduction

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. The size of a test-suite has a direct impact on the costs and the effort of software testing. Especially during regression testing, when software is re-tested after some modifications, the size of the test-suite is important. Common test-suite reduction techniques select subsets of test-suites that achieve given test requirements. Unfortunately, not only the test-suite size but also the fault detection ability is reduced as a consequence. This paper proposes a novel approach where test-cases created with model-checker based techniques are transformed such that redundancy within the test-suite is avoided, and the overall size is reduced. As test-cases are not simply discarded, the impact on the fault sensitivity is minimal.

1 Introduction

Software testing is a process that consumes a large part of the effort and resources involved in software development. Especially during regression testing, when software is re-tested after some modifications, the size of the test-suite has a large impact on the total costs. Therefore, the idea of test-suite reduction (also referred to as test-suite minimization) is to find a minimal subset of the test-suite that is sufficient to achieve the given test requirements.

Various heuristics have been proposed to approximate a minimal subset of the test-suite. These techniques can reduce the number of test-cases in a test-suite significantly. However, experiments have revealed that the quality of the test-suite suffers from this minimization. Even though the test requirements with regard to which the minimization was made are still fulfilled by the minimized test-suite, it has been shown that the overall ability to detect faults is reduced. In many scenarios, especially in the case of safety related software, such a degradation is unacceptable.

This paper introduces a novel approach to test-suite reduction. This approach tries to identify those parts of the test-cases that are truly redundant. Redundancy in this context means that there are no faults that can be detected with

* 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.

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 291–305, 2007.

the redundant part of a test-case, and not without. Instead of discarding test-cases out of a test-suite, the test-cases are transformed such that the redundancy is avoided. That way, the test-suite is minimized with regard to the number of test-cases and the total number of states, while neither test coverage nor fault detection ability suffer from the degradation experienced in previous approaches.

The approach uses the state information that is included in functional tests created with model-checker based test-case generation approaches. The model-checker is also used within an optimized version of the approach. An empirical evaluation shows that the approach is feasible.

This paper is organized as follows: Section 2 first introduces the problem of test-suite reduction and points out drawbacks of current solutions. Then, the necessary preliminaries for the remainder of the paper are discussed. Section 3 presents a new definition of redundancy in the context of test-cases, and shows how test-suites can be optimized in order to reduce redundancy. Section 4 describes experiments and results with regard to this optimization. Finally, Section 5 concludes the paper with a discussion of the results and an outlook.

2 Preliminaries

In this section, test-suite reduction and previous solutions are presented. Then, the necessary preliminaries for our approach are introduced.

2.1 Test-Suite Reduction

During regression testing the software is re-tested after some modifications. The costs of running a complete test-suite against the software repeatedly can be quite high. In general, not all test-cases of a test-suite are necessary to fulfill some given test requirements. Therefore, the aim of test-suite reduction is to find a subset of the test-cases that still fulfills the test requirements. The original test-suite reduction problem is defined by Harrold et al. [1] as follows:

Given: A test-suite TS, a set of requirements $r_1, r_2, , , r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of TS, $T_1, T_2, ..., T_n$, one associated with each of the r_is such that any one of the test-cases t_j belonging to T_i can be used to test r_i.

Problem: Find a representative set of test-cases from TS that satisfies all r_is.

The requirements r_i can represent any test-case requirements, e.g., test coverage. A representative set of test-cases must contain at least one test-case from each subset T_i. The problem of finding the optimal (minimal) subset is NP-hard. Therefore, several heuristics have been presented [1,2,3].

Test-suite reduction results in a new test-suite, where only the relevant subset remains and the other test-cases are discarded. Intuitively, removing any test-case might reduce the overall ability of the test-suite to detect faults. In fact, several experiments [4,5,6] have shown that this is indeed the case, although

there are other claims [7]. Note that the reduction of fault sensitivity would also occur when using an optimal instead of a heuristic solution.

In this paper we introduce a new approach to test-suite minimization which does not have a negative influence on the fault detection ability. However, first we need to introduce some basic concepts and definitions.

2.2 Model-Checker Based Testing

In this paper we consider test-cases generated with model-checker based methods. A model-checker is a tool originally intended for formal verification. In general, a model-checker takes as input a finite-state model of a system and a temporal logic property and efficiently verifies the complete state space of the model in order to determine whether the property is fulfilled or not. If the property is not fulfilled then a counter-example is returned, which is a sequence of states beginning in the initial state and leading to the violating state. There are several different approaches that exploit this counter-example mechanism for automated test-case generation [8,9,10,11,12,13,14]. Model-checkers use Kripke structures as model formalism:

Definition 1. *Kripke Structure: A Kripke structure K is a tuple $K = (S, s_0, T, L)$, where S is the set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times S$ is the 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.*

A model-checker verifies whether a model M satisfies a property P. If M violates P, denoted as $M \nvDash P$, then the model-checker returns a trace that illustrates the property violation. The trace is a finite prefix of an execution sequence of the model (path):

Definition 2. *Path: A path $p := \{s_0, s_1, ...\}$ of Kripke structure K is a finite or infinite sequence such that $\forall i > 0 : (s_i, s_{i+1}) \in T$ for K.*

Informally, the states of a Kripke structure and its traces consist of value assignments to its input, output and internal variables. Input variables are those that are provided by the environment to the model, output variables are returned to the environment by the model, and internal variables are not visible outside of the model. A trace can be used as a test-case by providing the input variables to the system under test (SUT), and then comparing whether the outputs produced by the SUT match those of the trace. Therefore, a trace can be seen as a test-case:

Definition 3. *Test-Case: A test-case t is a finite prefix of a path p of Kripke structure K.*

The number of transitions a test-case consists of is referred to as its *length*. E.g., test-case $t := \{s_0, s_1, ..., s_i\}$ has a length of $length(t) = i$. 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*. The aim of test-suite reduction is to optimize test-suites with respect to their size and total length:

Definition 4. *Test-Suite: A test-suite TS is a finite set of n test-cases. The* size *of TS is n. The overall* length *of a test-suite TS is the sum of the lengths of its test-cases t_i: $length(TS) = \sum_{i=1}^{n} length(t_i)$.*

Coverage criteria are used to measure test-suite quality. In the model-based scenario we assumed, we are mainly interested in model-based coverage criteria. A coverage criterion describes a set of structural items or aspects that a test-suite should cover. The test coverage is the percentage of items that are actually covered, i.e., reached during test-case execution. A model-based coverage criterion can be expressed as a set of properties (*trap properties* [8]) where a test-case covers an item if the according property is violated.

Definition 5. *Test Coverage: The coverage C of a test-suite TS with regard to a coverage criterion represented by a set of properties $\mathcal{P}$ is defined as the ratio of covered properties to the number of properties in total:*

$$C = \frac{1}{|\mathcal{P}|} \cdot |\{x | x \in \mathcal{P} \wedge covered(x, TS)\}|$$

The predicate $covered(a, TS)$ is true if there exists a test-case $t \in TS$ such that t covers a, i.e., $t \not\models a$.

The fault detection ability describes the potential of a test-suite at detecting faults. The higher this ability, the more faults can be detected. In practice, the mutant score [15] is used as an estimate for the fault detection ability. A mutant results from a single syntactic modification of a model or program. The mutant score of a test-suite is the ratio of mutants that can be distinguished from the original to mutants in total. A mutant is detected if the execution leads to different results than expected.

Definition 6. *Test-case execution: A test-case $t = \{s_0, s_1, ...\}$ for model K is executed by taking the input variables of each state s_i, providing them to the SUT with a suitable test framework. These values and the produced output values represent an execution trace $tr = \{s'_0, s'_1, ...\}$. A fault is detected, iff $\exists (s'_i, s'_{i+1}) \in tr : (s'_i, s'_{i+1}) \notin T$ for K. (s_i, s_{i+1}) is referred to as a* step.

3 Test-Suite Redundancy

Previously, redundancy was used to describe test-cases that are not needed in order to achieve a certain coverage criterion. As the removal of such test-cases leads to a reduced fault detection ability, they are not really redundant in a generic way. In contrast, we say a test-case contains redundancy if part of the test-case does not contribute to the fault detection ability. This section aims to identify such redundancy, and describes possibilities to reduce it.

3.1 Identifying Redundancy

Intuitively, identical test-cases are redundant. For any two test-cases t_1, t_2 such that $t_1 = t_2$, any fault that can be detected by t_1 is also identified by t_2 and vice versa, assuming the test-case execution framework assures identical preconditions for both tests. Similarly, the achieved coverage for any coverage criterion is identical for both t_1 and t_2. Clearly, a test-suite does not need both t_1 and t_2.

The same consideration applies to two test-cases t_1 and t_2, where t_1 is a prefix of t_2. t_1 is subsumed by t_2, therefore any fault that can be detected by t_1 is also detected by t_2 (but not vice versa). In this case, t_1 is redundant and is not needed in any test-suite that contains t_2. In model-based testing it is common practice to discard subsumed and identical test-cases at test-case generation time [12].

This leads to the kind of redundancy which we are interested in: Model-checker based test-case generation techniques often lead to such test-suites where all test-cases begin with the same initial state. From this state on different paths are taken, but many of these paths are equal up to a certain state. Any fault that occurs within such a sub-path can be detected by any of the test-cases that begins with this sub-path. Within these test-cases, the sub-path is *redundant*.

This kind of redundancy can be illustrated by representing a set of test-cases as a tree. The initial state that all test-cases share is the root-node of this tree. A sub-path is redundant if it occurs in more than one test-case. In the tree representation, any node below the root node that has more than one child node contains redundancy. If there are different initial states, then there is one tree for each initial state.

Definition 7. *Test-Suite Execution Tree: Test test-cases $t_i = \{s_0, s_1, ...s_l\}$ of a test-suite TS can be represented as a tree, where the root node equals the initial state common to all test-cases: $root(TS) = s_0$. For each successive, distinct state s_j a child node is added to the previous node s_i:*

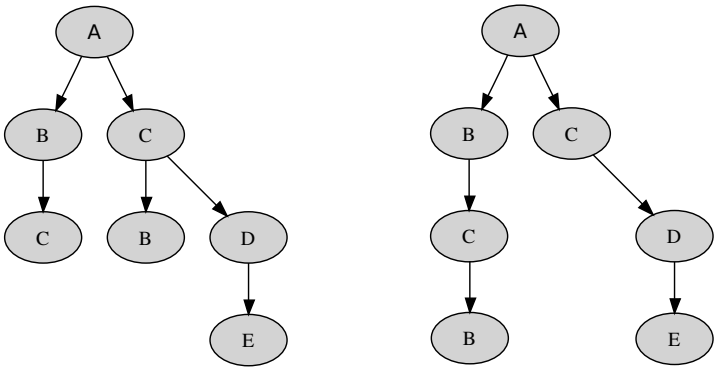
$$s_j : (s_i, s_j) \in t_i \rightarrow s_j \in children(s_i)$$

The depth of the tree equals the length of the longest test-case in TS. $children(x)$ denotes the set of child nodes of node x. Consider a test-suite consisting of three test-cases (letters represent distinct states): "`A-B-C`", "`A-C-B`", "`A-C-D-E`". The execution tree representation of these test-cases can be seen in Figure 1(a). The rightmost `C`-state has two children, therefore the sub-path `A-C` is contained in two test-cases; it is redundant. The execution tree can be used to measure redundancy:

Definition 8. *Test-Suite Redundancy: The redundancy R of a test-suite TS is defined with the help of the execution tree:*

$$R(TS) = \frac{1}{n-1} \cdot \sum_{x \in children(root(TS))} \mathcal{R}(x) \tag{1}$$

The redundancy of the tree is the ratio of the sum of the redundancy values $\mathcal{R}$ for the children of the root-node and the number of arcs in the tree ($n-1$, with



(a) 17% redundancy (b) No redundancy

Fig. 1. Simple test-suite with redundancy represented as execution tree

n nodes). The redundancy value $\mathcal{R}$ is defined recursively as follows:

$$\mathcal{R}(x) = \begin{cases} (|children(x) - 1|) + \sum_{c \in children(x)} \mathcal{R}(c) & if \quad children(x) \neq \{\} \\ 0 & if \quad children(x) = \{\} \end{cases} \tag{2}$$

The example test-suite depicted as tree in Figure 1(a) has a total of 7 nodes, where one node besides the root node has more than one child. Therefore, the redundancy of this tree equals $R = \frac{1}{7-1} \cdot \sum_{x \in children(root(TS))} \mathcal{R}(x) = \frac{1}{6} \cdot (0 + (1 + 0)) = \frac{1}{6} = 17\%$.

A test-suite contains no redundancy if for each initial state there are no test-cases with common prefixes, e.g., if there is only one test-case per initial-state.

3.2 Removing Redundancy

Having identified redundancy, the question now is how to reduce it. This section introduces an approach to solve this problem. It has already been stated that the removal of test-cases from a test-suite has a negative impact on the fault detection ability, therefore this is not an option. Instead, the proposed solution is to transform the test-cases such that the redundant parts can be omitted.

For each test-case t_i of test suite TS a common prefix among the test-cases is determined. If such a prefix is found, then the test-case is redundant for the length of the prefix and only interesting after the prefix. If there is another test-case t_j that ends with the same state as the prefix does, then the remainder of the test-case t_i can be appended to t_j, and t_i can safely be discarded. This algorithm is shown in Listing 1. It is of interest to find the longest possible prefixes, therefore the search for prefixes starts with the length of the test-case under examination and then iteratively reduces the length down to 1. This also guarantees that duplicate or subsumed test-cases are eliminated.

The function *find_test* searches for a test-case that ends with the same state as the currently considered prefix, its worst time complexity therefore is $O(|TS|)$. The complexity of *has_prefix* is $O(n)$ as it depends on the prefix length. Appending and deleting test-cases take constant time. These operations are nested in a loop over $|TS|$, which in turn is called for all possible prefix lengths. Finally, this is done for each test-case in TS. Therefore, the worst-case complexity of this algorithm is $O(|TS|^2 \cdot n \cdot (|TS| + n))$; with realistic test-suite sizes it is still applicable. The algorithm terminates for every finite test-suite. In the listing, $t[n]$ denotes the nth state of test-case t, and $t[-1]$ the last state of t.

```
for each t in TS do
  for n := length(t) downto 1 do
    for each t2 in TS do
      if has_prefix(t2, t, n) and t2 != t then
        t3 := find_test(TS, t[n])
        if t3 != None then
          append_postfix(t3, t, n)
          delete(TS, t)
          break
        end if
      end if
    end for
  end for
end for
```

Listing 1. Test-suite transformation

The algorithm has to make non-deterministic choices when selecting a test-case as a source for the prefix, when selecting a test-case to look for the common prefix and when searching for a test-case to append to. These choices have an influence on how fast a test-suite is processed. In addition, the number of test-cases remaining in the final reduced test-suite also depends on these choices. The success of the reduction depends on whether there are suitable test-cases where parts of other test-cases can be appended. A test-case that is necessary for removal of a long common prefix might be used to append another test-case with a shorter common prefix earlier. In that case, the long prefix could not be removed unless there was another suitable test-case. Determination of the optimal order would have to take all permutations of the test-suite order into consideration and is therefore not feasible. In practice, the algorithm is implemented such that test-cases are selected sequentially in the order in which they are stored in the test-suite.

Figure 1(b) illustrates the result of this optimization applied to the Figure 1(a). The test-case `A-C-B` has the common prefix `A-C`, and there is a test-case ending in `C`, therefore the postfix `B` of `A-C-B` is appended to `A-B-C`, resulting in `A-B-C-B`.

This algorithm optimizes the total costs of a test-suite with respect to two factors: It reduces the total number of test-cases (test-suite size), and it reduces the overall number of states contained in the test-suite (test-suite length). In the resulting test-suite individual test-cases can be longer than in the original test-suite. We assume that the costs of executing two test-cases of length n are higher than that of executing one test-case of length $2 \cdot n$ because of setup and pull-down overhead. Therefore, it is preferable to have fewer but longer test-cases instead of many small ones. This assumption is for example also made in [16], where the test-case generation aims to create fewer but longer test-cases.

While the computational complexity of the algorithm is high, the success depends on the actual test-suite. A test-suite might contain significant redundancy but have few test-cases that are suitable for appending, in which case not much optimization can be achieved. In addition, the order in which test-cases are selected has an influence on the results.

As we assumed a model-checker based test-case generation approach, we can make use of the model-checker for optimization purposes. If appending is not possible, then the model-checker can be used to create a 'glue'-sequence to append the postfix to an arbitrary test-case. Of course the model-checker is not strictly necessary to perform this part; there are other possibilities to find a path in the model. However, the model-checker is a convenient tool for this task, especially if it is already used for test-case generation in the first place. Listing 2 lists the extended algorithm. The function *choose_nondeterministic(TS)* chooses one test-case out of the test-suite TS non-deterministically. This choice has an influence on the length of the resulting glue-sequence. An optimal algorithm would have to consider the lengths of all such possible glue-sequences, and therefore calculate all of them. A distance heuristic is conceivable, which estimates the distance between the final state of a test-case and the state the glue sequence should lead to. For reasons of simplicity, the prototype implementation used for experiments in this paper makes a random choice.

The function *create_sequence* calls the model-checker in order to create a suitable glue sequence. A sequence from state a to state b can be created by verifying a property that claims that such a path does not exist. If such a sequence exists, the counter-example consists of a sequence from the initial state to a, and then a path from a to b. For example, when using computation tree logic (CTL) [17], this query can be stated as: `AG a -> !(EF b)`.

The presented algorithms reduce both the number of test-cases and the total test-suite length, while previous methods selected subsets of the test-suite. Therefore, the effects on the quality of the resulting test-suite are different.

Each step of a test-case adhering to Definition 3 fully describes the system state. A model-checker trace consists of the values of all input and output variables as well as internal variables. A fault is detected if the actual outputs of the implementation differ from those of the test-case. Therefore, any fault that

```
for each t in TS do
  for n := length(t) downto 1 do
    for each t2 in TS do
      if has_prefix(t2, t, n) and t2 != t then
        t3 := find_test(TS, t[n])
        if t3 != None then
          append_postfix(t3, t, p)
          delete(TS, t)
          break
        end if
      else
        t3 := choose_nondeterministic(TS)
        if t3 != t then
          s = create_sequence(t3[-1], t[n])
          append(t3, s)
          append_postfix(t3, t, n)
          delete(TS, t)
          break
        end if
      end if
    end for
  end for
end for
```

Listing 2. Test-suite transformation with glue sequences

occurs deterministically at a certain state can be detected with a step of a test-case, no matter when this step is executed. As the test-suite reduction guarantees that only redundant steps as parts of prefixes are removed, any fault that can be detected by a test-suite TS, can also be detected by the test-suite resulting from reduction of TS. It is conceivable that there are faults that do not deterministically occur at certain system states. For example, a fault might only occur after a certain sequence has been executed, or if a state is executed a certain number of times. However, we have not found such a fault in our experiments. Furthermore, it is equally possible that the transformation leads to such test-cases that can detect previously missed non-deterministic faults.

Definition 5 allows arbitrary properties for measuring test coverage. Whether the test-suite reduction has an impact on the test coverage depends on the actual properties. If the coverage depends on the order of not directly adjacent steps in the test-case, then splitting a prefix from a test-case and appending the remainder to another test-case can reduce the coverage. For example, transition pair coverage [18] requires all pairs of transitions to be covered. A transition pair can be split during the transformation. However, the appending can also lead to transition pairs previously uncovered. In practice, many coverage properties do not consider the execution order, e.g. transition or full-predicate coverage [18], or coverage criteria based on the model-checker source file [9].

4 Empirical Evaluation

This section presents the results of an empirical evaluation of the concepts described in the previous sections. The evaluation aims to determine how much reduction can be achieved with the presented algorithms, and how they perform in comparison to other approaches. Furthermore, the effects on coverage and mutant score are analyzed.

4.1 Experiment Setup

The experiment uses three examples, each consisting of a model and specification written in the language of the model-checker NuSMV [19]. For each model, 23 different test-suites are created with different methods (various coverage criteria for coverage based methods, different mutation operators for mutation based approaches, property based methods). The details of these methods are omitted for space reasons and because they are not necessary to interpret the results. In addition, a set of mutant models is created for each model. The use of a model-checker allows the detection of equivalent mutants, therefore only non-equivalent mutants are used for the evaluation of a mutant score. Car Control (CA) is a simplified model of a car control. The Safety Injection System (SIS) example was introduced in [20] and has since been used frequently for studying automated test-case generation. Cruise Control (CC) is based on [21]. A set of faulty implementations for this example was written by Jeff Offutt. The presented algorithms are implemented with Python, and the symbolic model-checker NuSMV is used.

4.2 Lossy Minimization with Model-Checkers

For comparison purposes, a traditional minimization approach is applied to the model-checker scenario, similarly to Heimdahl and Devaraj [6]. Model-based coverage criteria can be expressed as *trap properties* [8] (Section 2.2). The test-cases are converted to models and then the model-checker is challenged with the resulting models and the trap properties. For each trap property that results in a counter-example it is known that the test-case covers the according item.

A minimized subset of the test-suite achieving a criterion can be determined by calculating the covered properties for each test-case, and then iteratively selecting the test-case that covers the most yet uncovered properties. We choose *transition coverage* as first example coverage criterion. Black [13] proposed a test-case generation approach based on mutation of the reflected transition relation. The mutated, reflected properties can be used similarly to trap properties for test-case generation, to determine a kind of mutant score and also for minimization. In order to distinguish this from the mutant score determined by execution of the test-case against mutant models we dub the former *reflection coverage*.

4.3 Results

Tables 1, 2 and 3 list the average values of the minimization of the 23 test-suites for the three example models. "Redundancy" denotes the algorithm in

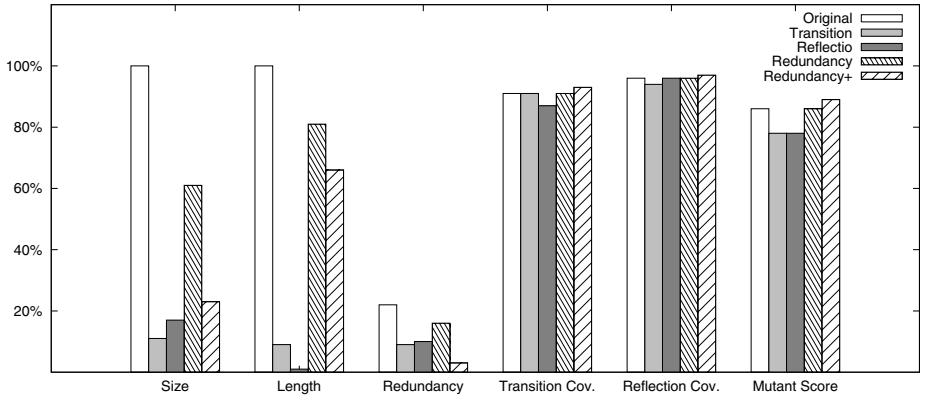


Fig. 2. Comparison of reduction methods, average percentage over all three example models and 23 test-suites each

Listing 1, and "Redundancy+" the extended version of Listing 2. In all cases the coverage-based reduction techniques result in smaller test-suites than the direct redundancy based approach. The extended redundancy based approach comes close to the coverage based approaches with respect to test-suite size. The test-suite length is reduced proportionally to the test-suite size for coverage based techniques, while as expected the redundancy based length savings are not as significant. Again, the extended algorithm achieves better results, showing that the potential saving in redundancy is bigger than what is added by the glue sequences. In general, even though the reduction in the total length is smaller with the redundancy approaches than with the coverage approaches, it is still significant and shows that the approach is feasible.

The test coverage of coverage minimized test-suites is not changed for the criterion that is used for minimization, while a degradation with the other criterion is observable. In contrast, the redundancy based approach has no impact on the coverage of either criterion. The extended redundancy approach even leads to a minor increase of the coverage, due to the glue sequences. As for the mutant score, the coverage based approaches lead to a degradation of up to 16%, while the redundancy approach has no impact on the mutant score, and the extended redundancy approach again results in a slight increase. Figure 2 sums up the results of the experiments for all models and test-suites. As these experiments use only models and mutants of the models, this raises the question whether the results are different with regard to actual implementations. Therefore, the Cruise Control test-suites are run against the set of faulty implementations. Table 4 lists the results. They are in accordance with those achieved with model mutants, which indicates the validity also for implementations.

Figure 3 illustrates the effects of the order in which test-cases are selected at several points in the algorithm as box-plots. The box-plots illustrate minimum, maximum, median and standard deviation for the achieved reduction with the 23 test-suites per example, each randomly sorted 5 times. Figure 3(a) shows the

Table 1. Results in average for Cruise Control example

Method	Size	Length	Redundancy	Transition Coverage	Mutant Score (Reflection)	Mutant Score
Original	36,55	213,77	44,55%	89,16%	95,93%	87,31%
Transition	6,23	35,6	32,00%	89,16%	94,46%	79,70%
Reflection	6,59	37,36	30,30%	78,67%	95,93%	73,42%
Redundancy	27,91	186,09	36,99%	89,16%	95,93%	87,31%
Redundancy+	8,95	152,73	4,82%	89,86%	96,44%	89,13%

Table 2. Results in average for SIS example

Method	Size	Length	Redundancy	Transition Coverage	Mutant Score (Reflection)	Mutant Score
Original	21,87	644,04	10,45%	89,28%	95,15%	78,29%
Transition	3,26	84,17	2,51%	89,28%	93,42%	68,72%
Reflection	4,91	126,3	4,53%	87,63%	95,15%	72,70%
Redundancy	14,48	440,39	6,15%	89,28%	95,15%	78,29%
Redundancy+	5,52	268,78	0,46%	91,23%	96,19%	81,42%

Table 3. Results in average for Car Control example

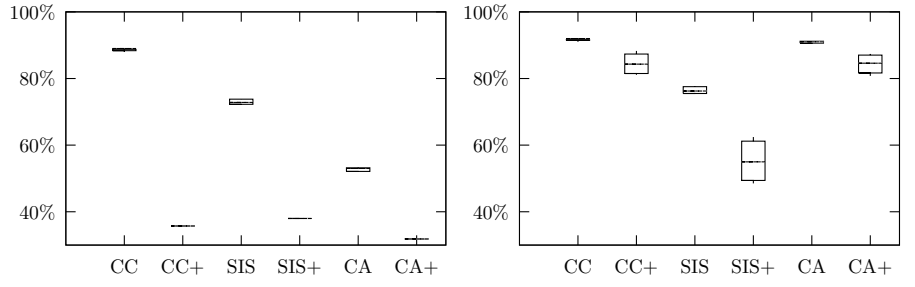
Method	Size	Length	Redundancy	Transition Coverage	Mutant Score (Reflection)	Mutant Score
Original	54,09	1351,91	10,44%	95,78%	96,07%	92,84%
Transition	3,36	71,32	3,05%	96,78%	93,76%	85,09%
Reflection	7,68	152,27	5,30%	95,54%	96,07%	87,62%
Redundancy	25,68	1182,82	4,69%	95,78%	96,07%	92,84%
Redundancy+	11,36	1058,05	1,17%	99,03%	97,53%	95,16%

effects on the test-suite sizes. As the use of glue sequences makes it possible to append to any test-case, the order has no effect on the resulting test-suite size in our experiments, therefore there is no deviation. There is insignificant variation when not using glue sequences, and also only minor variation in the test-suite length (Figure 3(b)). In contrast, the choice of a test-case to append to using a glue sequence has a visible influence on the resulting test-suite length. This suggests the use of a distance heuristic instead of the random choice.

Both presented algorithms have high worst-case complexity. However, many factors contribute to the performance: the test-suite size, the lengths of the

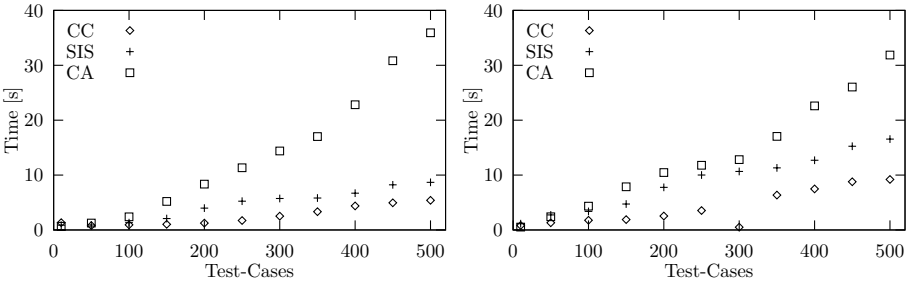
Table 4. Mutant scores for cruise-control implementation mutants

Original	Transition	Reflection	Redundancy	Redundancy+
75,8%	39,1%	37,2%	75,8%	76,5%



(a) Test-Suite Size Reduction (b) Test-Suite Length Reduction

Fig. 3. Effects of the test-case order, as percentage value of original sizes and lengths. Minimization using glue sequences is denoted by a '+' after the example name.



(a) Direct approach (b) Using glue sequences

Fig. 4. Minimization time vs. test-suite size

test-cases, the contained redundancy, the suitability of test-cases for the transformation, the order in which test-cases are selected, the effort of calculating glue sequences, etc. Figure 4 depicts the performance of the minimization for the three example models for different test-suite sizes executed on a PC with Intel Core Duo T2400 processor and 1GB RAM. Notably, the computation time for the car controller example increases more than for the other examples. This example has a bigger state space, therefore appending is not easily possible. Figure 4(b) shows that there is less difference in the increase in computation time when using glue sequences. The additional computational effort introduced by the generation of the glue sequences is very small, compared to its effect. However, performance measurement is difficult, as the redundancy is not constant along the test-suites used for measurement. In order to examine the scalability of the approach, minimization was also tested on a complex example with a significantly bigger test-suite. The example is a windscreen wiper controller provided by Magna Steyr. For a set of 8000 test-cases, basic minimization takes 35m22s. This example also shows the effects of the model complexity, as the calculation of glue sequences is costly for this model: Minimization with glue sequences takes 2h1m56s. Obviously, the performance is specific to each application and test-suite, but it seems to be acceptable in general.

5 Conclusion

In this paper we have introduced an approach to minimize the size of a test-suite with regard to the number of test-cases and the total length of all test-cases. The approach detects redundancy within the test-suite and transforms test-cases in order to avoid the redundancy. In contrast to previous approaches the quality of the resulting test-suites does not suffer with regard to test coverage or fault detection ability from this minimization under certain conditions. In fact, experiments showed that the resulting test-suites can even be slightly improved. The experiments also showed that the reduction is significant, although not as large as with approaches that heuristically discard test-cases.

One drawback of this approach is the run-time complexity of the algorithm. However, even without further optimizations the approach is applicable to realistic test-suites without problems. The transformation relies on information that might not be available in all test-suites. Complete state information is necessary, as is provided by model-checker based test-case generation approaches. There are several possibilities to continue work on this approach:

- It would be desirable to optimize the basic algorithm with regard to its worst case execution time.
- The non-deterministic choice might not always lead to the best results. Heuristics for choosing test-cases could lead to better reduction.
- The algorithms presented in this paper sequentially analyze the test-cases in a test-suite. Therefore, a single run might not immediately eliminate all the redundancy. It is conceivable to iteratively call the algorithm until the redundancy is removed completely. This is likely to lead to test-suites of very small size, where each test-case is very long.
- In this paper, a scenario of model-checker based testing was assumed. It would be interesting to evaluate the applicability to other settings.
- The presented definition of redundancy only considers common prefixes. However, common path segments might also exist within test-cases. Consideration of this kind of redundancy might lead to further optimizations.

References

1. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. **2**(3) (1993) 270–285
2. Gregg Rothermel, Mary Jean Harrold, J.v.R.C.H.: Empirical studies of test-suite reduction. Software Testing, Verification and Reliability **12**(4) (2002) 219–249
3. Zhong, H., Zhang, L., Mei, H.: An experimental comparison of four test suite reduction techniques. In: ICSE '06: Proceeding of the 28th international conference on Software engineering, New York, NY, USA, ACM Press (2006) 636–640
4. Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. **29**(3) (2003) 195–209
5. Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: ICSM '98: Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (1998) 34

6. Heimdahl, M.P.E., Devaraj, G.: Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In: ASE, IEEE Computer Society (2004) 176–185
7. Wong, W.E., Horgan, J.R., London, S., Mathur, A.P.: Effect of test set minimization on fault detection effectiveness. In: ICSE '95: Proceedings of the 17th international conference on Software engineering, ACM Press (1995) 41–50
8. Gargantini, A., Heitmeyer, C.: 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., Springer (1999) 146–162
9. Rayadurgam, S., Heimdahl, M.P.E.: 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), IEEE Computer Society (2001) 83–91
10. Hamon, G., de Moura, L., Rushby, J.: Automated Test Generation with SAL. Technical report, Computer Science Laboratory, SRI International (2005)
11. Callahan, J.R., Easterbrook, S.M., Montgomery, T.L.: Generating Test Oracles Via Model Checking. Technical report, NASA/WVU Software Research Lab (1998)
12. Ammann, P., Black, P.E., Majurski, W.: Using Model Checking to Generate Tests from Specifications. In: ICFEM. (1998)
13. Black, P.E.: Modeling and Marshaling: Making Tests From Model Checker Counterexamples. In: Proc. of the 19th Digital Avionics Systems Conference. (2000)
14. Okun, V., Black, P.E., Yesha, Y.: Testing with Model Checker: Insuring Fault Visibility. In Mastorakis, N.E., Ekel, P., eds.: Proceedings of 2002 WSEAS International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems. (2003) 1351–1356
15. Ammann, P., Black, P.E.: A Specification-Based Coverage Metric to Evaluate Test Sets. In: HASE, IEEE Computer Society (1999) 239–248
16. Hamon, G., de Moura, L., Rushby, J.: Generating Efficient Test Sets with a Model Checker. In: Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04). (2004) 261–270
17. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop, London, UK, Springer-Verlag (1982) 52–71
18. Offutt, A.J., Xiong, Y., Liu, S.: Criteria for generating specification-based tests. In: ICECCS, IEEE Computer Society (1999)
19. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A New Symbolic Model Verifier. In: CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (1999) 495–499
20. Bharadwaj, R., Heitmeyer, C.L.: Model Checking Complete Requirements Specifications Using Abstraction. Automated Software Engineering **6**(1) (1999) 37–68
21. 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)