LNCS 7212
ARCoSS

Juan de Lara
Andrea Zisman (Eds.)

Fundamental Approaches to Software Engineering

15th International Conference, FASE 2012
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2012
Tallinn, Estonia, March/April 2012. Proceedings

Springer

Table of Contents

Intermodelling and Model Transformations

Modelling and Adaptation

Product Lines and Feature-Oriented Programming

Development Process

Verification and Synthesis

Testing and Maintenance

Slicing and Refactoring

Foreword

ETAPS 2012 is the fifteenth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised six sister conferences (CC, ESOP, FASE, FOSSACS, POST, TACAS), 21 satellite workshops (ACCAT, AIPA, BX, BYTECODE, CMCS, DICE, FESCA, FICS, FIT, GRAPHITE, GT-VMT, HAS, IWIGP, LDTA, LINEARITY, MBT, MSFP, PLACES, QAPL, VSSE and WRLA), and eight invited lectures (excluding those specific to the satellite events).

The six main conferences received this year 606 submissions (including 21 tool demonstration papers), 159 of which were accepted (6 tool demos), giving an overall acceptance rate just above 26%. Congratulations therefore to all the authors who made it to the final programme! I hope that most of the other authors will still have found a way to participate in this exciting event, and that you will all continue to submit to ETAPS and contribute to making it the best conference on software science and engineering.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis, security and improvement. The languages, methodologies and tools that support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a confederation in which each event retains its own identity, with a separate Programme Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronised parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for 'unifying' talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

This year, ETAPS welcomes a new main conference, *Principles of Security and Trust*, as a candidate to become a permanent member conference of ETAPS. POST is the first addition to our main programme since 1998, when the original five conferences met in Lisbon for the first ETAPS event. It combines the practically important subject matter of security and trust with strong technical connections to traditional ETAPS areas.

A step towards the consolidation of ETAPS and its institutional activities has been undertaken by the Steering Committee with the establishment of *ETAPS e.V.*, a non-profit association under German law. ETAPS e.V. was founded on April 1st, 2011 in Saarbrücken, and we are currently in the process of defining its structure, scope and strategy.

ETAPS 2012 was organised by the *Institute of Cybernetics at Tallinn University of Technology*, in cooperation with

- ▷ European Association for Theoretical Computer Science (EATCS)
- ▷ European Association for Programming Languages and Systems (EAPLS)
- ▷ European Association of Software Science and Technology (EASST)

and with support from the following sponsors, which we gratefully thank:

Institute of Cybernetics at TUT; Tallinn University of Technology (TUT); Estonian Centre of Excellence in Computer Science (EXCS) funded by the European Regional Development Fund (ERDF); Estonian Convention Bureau; and Microsoft Research.

The organising team comprised:

General Chair:	*Tarmo Uustalu*
Satellite Events:	*Keiko Nakata*
Organising Committee:	*James Chapman, Juhan Ernits, Tiina Laasma, Monika Perkmann* and their colleagues in the *Logic and Semantics* group and *administration* of the *Institute of Cybernetics*

The ETAPS portal at http://www.etaps.org is maintained by *RWTH Aachen University*.

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Vladimiro Sassone (Southampton, Chair), Roberto Amadio (Paris 7), Gilles Barthe (IMDEA-Software), David Basin (Zürich), Lars Birkedal (Copenhagen), Michael O'Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Vittorio Cortellessa (L'Aquila), Koen De Bosschere (Gent), Pierpaolo Degano (Pisa), Matthias Felleisen (Boston), Bernd Finkbeiner (Saarbrücken), Cormac Flanagan (Santa Cruz), Philippa Gardner (Imperial College London), Andrew D. Gordon (MSR Cambridge and Edinburgh), Daniele Gorla (Rome), Joshua Guttman (Worcester USA), Holger Hermanns (Saarbrücken), Mike Hinchey (Lero, the Irish Software Engineering Research Centre), Ranjit Jhala (San Diego), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Barbara König (Duisburg), Juan de Lara (Madrid), Gerald Lüttgen (Bamberg), Tiziana Margaria (Potsdam), Fabio Martinelli (Pisa), John Mitchell (Stanford), Catuscia Palamidessi (INRIA Paris), Frank Pfenning (Pittsburgh), Nir Piterman (Leicester), Don Sannella (Edinburgh), Helmut Seidl (TU Munich),

Scott Smolka (Stony Brook), Gabriele Taentzer (Marburg), Tarmo Uustalu (Tallinn), Dániel Varró (Budapest), Andrea Zisman (London), and Lenore Zuck (Chicago).

I would like to express my sincere gratitude to all of these people and organisations, the Programme Committee Chairs and PC members of the ETAPS conferences, the organisers of the satellite events, the speakers themselves, the many reviewers, all the participants, and Springer-Verlag for agreeing to publish the ETAPS proceedings in the ARCoSS subline.

Finally, I would like to thank the Organising Chair of ETAPS 2012, Tarmo Uustalu, and his Organising Committee, for arranging to have ETAPS in the most beautiful surroundings of Tallinn.

January 2012

Vladimiro Sassone
ETAPS SC Chair

Lecture Notes in Computer Science 7212

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Advanced Research in Computing and Software Science
Subline of Lectures Notes in Computer Science

Juan de Lara Andrea Zisman (Eds.)

Fundamental Approaches to Software Engineering

15th International Conference, FASE 2012
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012
Tallinn, Estonia, March 24 – April 1, 2012
Proceedings

Springer

Volume Editors

Juan de Lara
Universidad Autónoma de Madrid
School of Computer Science
Campus Cantoblanco
28049 Madrid, Spain
E-mail: juan.delara@uam.es

Andrea Zisman
City University
School of Informatics
Northampton Square
London EC1V 0HB, UK
E-mail: a.zisman@soi.city.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-28871-5 e-ISBN 978-3-642-28872-2
DOI 10.1007/978-3-642-28872-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012932857

CR Subject Classification (1998): D.2.4, D.2, F.3, D.3, C.2, H.4, C.2.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Reduction of Test Suites Using Mutation

Macario Polo Usaola[1], Pedro Reales Mateo[1], and Beatriz Pérez Lamancha[2]

[1] Department of Information Systems and Technologies, University of Castilla-La Mancha,
Paseo de la Universidad 4, 13071-Ciudad Real, Spain
{macario.polo,pedro.reales}@uclm.es
[2] Software Testing Centre (CES), University of Republic
Lauro Müller 1989, Montevideo, Uruguay
bperez@fing.edu.uy

Abstract. This article proposes an algorithm for reducing the size of test suites, using the mutation score as the criterion for selecting the test cases while preserving the quality of the suite. Its utility is also checked with a set of experiments, using benchmark programs and industrial software.

Keywords: Test suites, mutation, test suite reduction, criteria subsumption.

1 Introduction

Mutation is a testing technique, originally proposed in 1978 by DeMillo et al. [1], which relies on the discovery of the artificial faults which are seeded in the system under test (SUT). These faults are injected in the SUT by means of a set of mutation operators, whose purpose is to imitate the faults that a common programmer may commit. Thus, each mutant is a copy of the program under test, but with a small change in its code, which is interpreted as a fault.

Mutants are usually generated by automated tools that apply a set of mutation operators to the sentences of the original program, thus producing a high number of mutants because, in general, each mutant contains only one fault. The fault in a mutant is discovered when the execution of a given test case produces a different output in the original program and in the mutant. When the fault is discovered, it is said that the mutant has been "killed"; otherwise, the mutant is "alive".

In order to obtain a good set of mutants, it is important that the seeded faults be "good", which depends on the quality of the mutation operators applied. This area has been closely studied, with the proposal of operators for different kinds of languages and environments, as for example in [2]. Faults introduced in the mutants must imitate common errors by programmers since, by means of the "coupling effect", a test suite that detects all simple faults in a program is so sensitive that it also detects more complex faults [3].

Figure 1 shows the source code of an original program (the SUT) and of some mutants: three of them proceed from the substitution of an arithmetic operator, whereas in the fourth a unary operator (++) has been added at the end of the sentence. The bottom of the figure presents the results obtained from executing some test cases

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 425–438, 2012.

on the different program versions. The test case corresponding to the test data *(1, 1)* produces different outputs in the original program (whose output is correct) and in Mutant 1: thus, this test case has found the fault introduced in the mutant, leaving the mutant killed. On the other hand, since all test cases offer the same output in the original program and in Mutant 4, it is said that Mutant 4 is alive. Moreover, this mutant will never be killed by any test case, since variable *b* is incremented *after* returning the result. Mutants like this one are called "functionally-equivalent mutants" and may be considered as noise when results are analyzed: they have a syntactic change (actually not a fault) with respect to the original source code that cannot be found.

Original	Mutant 1	Mutant 2	Mutant 3	Mutant 4

int sum(int a,int b)
{
return a + b;
}
	int sum(int a,int b)
	{
	return a - b;
	}
		int sum(int a,int b)
		{
		return a * b;
		}
			int sum(int a,int b)
			{
			return a / b;
			}
				int sum(int a,int b)
				{
				return a + b++;
				}

	Test data (a,b)			
	(1, 1)	(0, 0)	(-1, 0)	(-1, -1)
Orig.	2	0	-1	-2
Mut.1	0	0	-1	0
Mut.2	1	0	0	1
Mut.3	1	Error	Error	1
Mut.4	2	0	-1	-2

Fig. 1. Code of some mutants and their results with some test data

The test suite quality is measured in terms of the Mutation Score [4] (Figure 2), a number between 0 and 1 which takes into account the number of mutants killed, the number of mutants generated and the number of functionally-equivalent mutants. A test suite is mutation-adequate when it discovers all the faults injected in the mutants.

$MS(P,T) = \frac{K}{M - E}$,	being: *P*: program under test; *T*: test suite; *K*: # of killed mutants; *M:* # of generated mutants; *E*: # of equivalent mutants

Fig. 2. Mutation score

Since that paper by DeMillo in 1978, many works have researched and developed tools to improve the different steps of mutation testing: mutant generation, test case execution and result analysis.

Regarding **mutant generation**, most works try to decrease the number of mutants generated, with different studies existing for selecting the most meaningful operators [5, 6], as well as techniques for generating the mutants more quickly [7]. Regarding **test execution,** several authors have proposed the use weak mutation [8, 9], prioritization of the functions of the program under test [10] or the use of n-order mutants [11]. An n-order mutant has *n* faults instead of 1. Polo et al. [11] have shown that the combination of 1-order mutants to produce a suite of 2-order mutants significantly decreases the number of functionally equivalent mutants, whereas the risk of leaving faults undiscovered remains low. This has a positive influence on the **result analysis** step, whose main difficulty resides in the discovery of the functionally

equivalent mutants, which is required to calculate the Mutation Score (Figure 2). Manual detection is very costly, although Offutt and Pan have demonstrated that is possible to automatically detect almost 50% of functionally equivalent mutants if the program under test is annotated with constraints [12].

Since many equivalent mutants are optimizations or de-optimizations of the original program (for example, Mutant 4 in Figure 1 de-optimizes the original program), Offutt and Craft have also investigated how compiler optimization techniques may help in the detection of equivalent mutants [3].

In general, mutation testing has evolved over the years and, today, it is very frequently used to validate the quality of different testing techniques [13]. Some recent works related to mutation propose specific operators for specific programming languages, such as Kim et al. [14], who propose mutation operators for Java and Barbosa et al. [2], with operators for C.

These works, developed so many years after the proposal of mutation, evidence the maturity of this testing technique. With the adequate operators, the mutation score can be considered as a powerful coverage criterion [15].

This article proposes one algorithm (although another one, less efficient, is also described) for reducing the size of a test suite, based on the mutation score: given a test suite T, the goal is to obtain a new test suite T', which obtains the same mutation score as T, being $|T'| \leq T$. Furthermore, the article discusses how the subsumption of criteria may be preserved when the reduction algorithm is executed.

The article is organized as follows: the two parts of Section 2 briefly describe strategies for test case generation (where the problem of redundant test cases is presented) and some works solving the problem of test suite reduction. Section 3 then presents the algorithm for test suite reduction based on mutation, completing its description with an example taken from the literature. The validity of the algorithm is analyzed in Section 4, both with some benchmark programs, widely used in testing literature, and with a set of industrial programs. Finally, we draw our conclusions.

2 Related Work

The fact of having big test suites increases the cost of their writing, validation and maintenance, taking into account the continuous evolution of software and the corresponding regression testing [16]. Due to this, several researchers have proposed different techniques to reduce the size of a test suite, while the coverage reached is preserved. The problem of reducing a test suite to the minimum possible cardinal is known as the "optimal test-suite reduction problem" and has been stated by Jones and Harrold [17] as in Figure 3.

Given:	Test Suite T, a set of test-case requirements $r_1, r_2, ..., r_n$, that must be satisfied to provide the desired test coverage of the program.
Problem:	Find $T' \subset T$ such that T' satisfies all r_i and $(\forall T'' \subset T, T''$ satisfies all $r \Rightarrow |T'| \leq |T''|)$

Fig. 3. The optimal test suite reduction problem

Applied to the *Triangle-type* example, and starting from the results obtained by the All *combinations* strategy, the problem consists of finding a minimal subset of test

cases that obtains the same coverage as the original test suite: i.e., to obtain a set of *n* test cases that reach the same coverage as the original suite, being $n \leq 216$ and *n* being the minimal. The optimal test-suite reduction problem is NP-hard [18] and, thus, its solution has been approached by means of algorithms which provide near-optimal solutions, usually with greedy strategies. The following subsections review some relevant works.

The HGS Algorithm. Harrold et al. [19] give a greedy algorithm (usually referred to as *HGS*) for reducing the suite of test cases into another, fulfilling the test requirements reached by the original suite. The main steps in this algorithm are:

1) Initially, all the test requirements are unmarked.
2) Add to *T'* those test cases that only exercise a test requirement. Mark the requirements covered by the selected test cases.
3) Order the unmarked requirements according to the cardinality of the set of test cases exercising one requirement. If several requirements are tied (since the sets of test cases exercising them have the same cardinality), select the test case that would mark the highest number of unmarked requirements tied for this cardinality. If multiple such test cases are tied, break the tie in favor of the test case that would mark the highest number of requirements with testing sets of successively higher cardinalities; if the highest cardinality is reached and some test cases are still tied, arbitrarily select a test case from among those tied. Mark the requirements exercised by the selected test. Remove test cases that become redundant as they no longer cover any of the unmarked requirements.
4) Repeat the above step until all testing requirements are marked.

Gupta Improvements. With different collaborators, Gupta has proposed several improvements to this algorithm:

- With Jeffrey [20], Gupta adds "selective redundancy" to the algorithm. "Selective redundancy" makes it possible to select test cases that, for any given test requirement, provide the same coverage as another previously selected test case, but that adds the coverage of a new, different test requirement. Thus, maybe *T'* reaches the All-branches criterion but not def-uses; therefore, a new test case t can be added to *T'* if it increases the coverage of the def-uses requirement: now, *T'* will not increase the All-branches criterion, but it will do so with def-uses.
- With Tallam [21], test case selection is based on Concept Analysis techniques. According to the authors, this version achieves same size or smaller size reduced test suites than prior heuristics as well as a similar time performance.

Heimdahl and George Algorithm. Heimdahl and George [22] also propose a greedy algorithm for reducing the test suite. Basically, they take a random test case, execute it and check the coverage reached. If this one is greater than the highest coverage, then they add it to the result. The algorithm is repeated five times to obtain five different reduced sets of test cases. Since chance is an essential component of this algorithm, the good quality of the results is not guaranteed.

McMaster and Memon Algorithm. McMaster and Memon [23] present another greedy algorithm. The parameter taken into account to include test cases in the reduced suite is based on the "unique call stacks" that test cases produce in the program under test. As can be seen, the criterion for selecting test cases (the number of unique call stacks) is not a "usual test requirement".

In **summary**, since the optimal test-suite reduction problem is NP-hard, all the approaches discussed propose a greedy algorithm to find a good solution with a polynomial-time algorithm and, as the discussed algorithms show, test requirement for test case selection can be anything: coverage of sentences, blocks, paths… or, as it is proposed in this paper, number of mutants killed.

According to [24, 25], the degree of automation of testing tasks in the software industry is very low. Often, testing is performed in an artisanal way, and the efforts carried out in the last years to obtain test automation mostly consist of the application of unit testing frameworks, such as JUnit or NUnit. As a matter of fact, the work by Ng et al. [26] shows the best results on test automation: 79.5% of surveyed organizations automate test execution and 75% regression testing. However, only 38 of the 65 organizations (58.5%) use test metrics, with defect count being the most popular (31 organizations). Although the work does not present any data about the testing tools used, these results suggest that most organizations are probably automating their testing processes with X-Unit environments. In order to improve these testing practices, software organizations require cost and time-effective techniques to automate and to improve their testing process. Thus, the introduction of software testing research results in industry is a must.

3 Test Suite Reduction Using Mutation

This section mainly describes a greedy algorithm to reduce the size of a test suite based on the Mutation Score. This algorithm is inspired in the mutation cost reduction algorithms briefly described in [27]. The number of mutants killed by each test case is used as the criterion for the inclusion of a test case in the reduced set of selected test cases.

Figure 4 shows *reduceTestSuite*, the main function of the algorithm. As inputs, it receives the complete set of test cases, the class under test and the complete set of mutants. In line 2, it executes all test cases against the class under test and against the mutants, saving the results in *testCaseResults*.

Then, the algorithm is prepared for selecting, in several iterations, the test cases that kill more mutants, what is done in the loop of lines 5-14.

The first time the algorithm enters this loop and arrives at line 7, the value of n (which is used to stop the iterations) is $|mutants|$: in this special case, the algorithm looks for a test case that kills all the mutants. If it finds it, the algorithm adds the test case to *requiredTC,* updates the value of n to 0 and ends; otherwise, it decreases n in line 16 and goes back into the loop.

Let us suppose that *n* is initially 100 (that is, there are 100 mutants of the class under test), and let us suppose that the algorithm does not find test cases that kill *n* mutants until *n=30*. With this value, the function *getTestCasesThatKillN* (called in line 7) returns as many test cases as test cases kill *n* different mutants: i.e., if there are two test cases (tc_1 and tc_2) that kill the same 30 mutants, *getTestCasesThatKillN* returns only one test case (for example, tc_1). If the intersection of the mutants killed by tc_1 and tc_2 is not empty, then the algorithm returns a set composed of tc_1 and tc_2.

```
1. reduceTestSuite(completeTC : SetOfTestCases, cut :
    CUT, mutants : SetOfMutants) : SetOfTestCases
2. testCaseResults = execute(completeTC, cut, mutants)
3. requiredTC = ∅
4. n=|mutants|
5. while (n>0)
6.  mutantsNowKilled = ∅
7.  testCasesThatKillN =
    getTestCasesThatKillN(completeTC, n, mutants,
    mutantsNowKilled, testCaseResults)
8.  if |testCasesThatKillN>0| then
9.   requiredTC = requiredTC ∪ testCasesThatKillN
10.    n = |mutants|-|mutantsNowKilled|
11.  else
12.    n = n - 1
13.  end if
14. end_while
15. return requiredTC
16.end
```

Fig. 4. Main function of the algorithm, which returns the reduced suite

When test cases killing the current *n* mutants are found, they are added to the *requiredTC* variable (line 9) and *n* is updated to the current number of remaining mutants.

In the actual implementation of the algorithm, the execution of the complete set of test cases against the CUT and the mutants is made in a separate function (*execute*, called in line 2), which returns a collection of *TestCaseResult* objects, which are composed of the name of a test case and the list of the mutants they kill.

The function in charge of collecting the set of test cases that kill *n* mutants is called in the 7th line in Figure 4 and is detailed in Figure 5. It goes through the elements in *testCaseResults* and takes those test cases whose list of has *n* elements. Each time it finds a suitable test case, the function removes the mutants it kills from the set of mutants: in this way, the function guarantees that no two test cases killing the same set of mutants will be included in the result.

```
1. getTestCasesThatKillN(completeTC:SetOfTestCases,
    n:int, mutants:SetOfMutants, mutantsNowKilled :
    SetOfMutants, testCaseResults: SetOfTestCaseResults)
    : SetOfTestCaseResults
2.  testCasesThatKillN = ∅
3.  for i=1 to |testCaseResults|
4.   testCaseResult = testCaseResults[i]
5.  if |testCaseResult.killedMutants| == n and
    testCaseResult.killedMutants ⊆ mutants then
6.   testCasesThatKillN = testCasesThatKillN ∪
    testCaseResult.testCaseName
7.     mutantsNowKilled = mutantsNowKilled ∪
    testCaseResult.killed.Mutants
8.   mutants = mutants – mutantsNowKilled
9.  end_if
10. next
11. return testCasesThatKillN
12.end
```

Fig. 5. Function to obtain the test cases that kill *n* mutants

3.1 Example

Let us suppose the killing matrix in Table 1, corresponding to a supposed program with eight mutants (in the rows) and a test suite with seven test cases. Each column may be understood as an instance of *TestCaseResult*: in fact, there are seven instances of this type, each composed of the test case name and the list of mutants it kills: the first test case result is composed of the *tc1* test case and the mutant *m2*; the second, by *tc2* and *m4, m5* and *m6*; the last one is composed of *tc7* and an empty set of mutants, since it kills none.

Table 1. First killing matrix for a supposed program

	tc1	tc2	tc3	tc4	tc5	tc6	tc7
m1			X		X		
m2	X		X		X		
m3			X				
m4		X				X	
m5		X					
m6		X		X			
m7				X			
m8				X			

Initially, *requiredTC* is the empty set and *n=7.* In the first iteration of the 5th line loop in Figure 4, the set *mutantsNowKilled* is ∅ because there are no test cases killing seven mutants. *n* is decreased to 6, 5, 4 and 3. In this iteration, the function *getTestCasesThatKillN* is called. This function (Figure 5) iterates from *i=1* to 7. When *i=1*, it rejects tc_1 because it does not kill three mutants (the current value of *n*). When *i=2:*

- Adds *{tc2}* to the *testCasesThatKillN* set.
- Adds *{m4, m5, m6}* to the *mutantsNowKilled* set.
- Leaves *mutants* with *{m1, m2, m3, m7,m8}*.

Henceforth, the killed mutants *{m4, m5, m6}* will not be considered in the following iterations. Then, the killing matrix can be now seen such as in Table 2.

Table 2. Second killing matrix for a supposed program

	tc1	tc2	tc3	tc4	tc5	tc6	tc7
m1			X		X		
m2			X		X		
m3			X	X			
m7				X			
m8				X			

Still inside *getTestCasesThatKillN,* the *i* variable is increased to 3 and the *TestCaseResult* corresponding to *tc3* is processed. Now:

- *testCasesThatKillN = {tc2, tc3}*
- *mutantsNowKilled = {m1, m2, m3, m4, m5, m6}*
- *mutants = {m7, m8}*

Mutants *{m1, m2, m3}* will not be considered in next iterations, thus leaving the killing matrix as in **Table 3**.

Table 3. Third killing matrix for a supposed program

	tc1	tc2	tc3	tc4	tc5	tc6	tc7
m7				X			
m8				X			

getTestCasesThatKillN continues increasing *i* to 7, the function exits and the algorithm returns to line 8 of *reduceTestSuite*. Here, *{tc2, tc3}* are added to *requiredTC* and *n* is decreased to 2, which is the initial number of mutants (8) minus the number of mutants now killed (6). Now, the function executes its last iteration calling *getTestCasesThatKillN* with n=*2*, and selects the *tc4* test case and adds it to *requiredTC*. The final value of this set is: *{tc2, tc3, tc4}*.

3.2 "A Motivational Example"

In their paper [20], Jeffrey and Gupta show, using the same title that leads this section, a small program to exemplify their algorithm with selective redundancy, which has been translated into the Java program shown in Figure 7. For this class, MuJava generates 48 traditional (15 of them are functionally-equivalent) and 6 class mutants for this program.

Using the values {-1.0, 0.0, +1.0} for the four parameters of function *f*, and generating test cases with the *All combinations* algorithm, the *testooj* tool [16] generates a test file with 3×3×3×3=81 test cases, that manage to kill 100% of the non-equivalent mutants. Figure 6 shows a piece of the killing matrix for this example, ordered according to the number of mutants killed by each test case (last column).

Of the three test cases killing 28 mutants, the algorithm picks *testTS_0_78* and adds it to the reduced suite, removing *testTS_0_76* and *testTS_0_77* since they kill the same mutants. Moreover, these are removed from the set of mutants. Then, it continues picking *testTS_054* and proceeds in the same way. The reduced suite is finally composed of 7 test cases, which is 8.6% of the original size: *{testTS_0_78, testTS_0_54, testTS_0_39, testTS_0_35, testTS_0_34, testTS_0_33, testTS_0_27}*

As expected, if the reduced suite is executed against the class and its mutants, it is seen that it also reaches the highest mutation score.

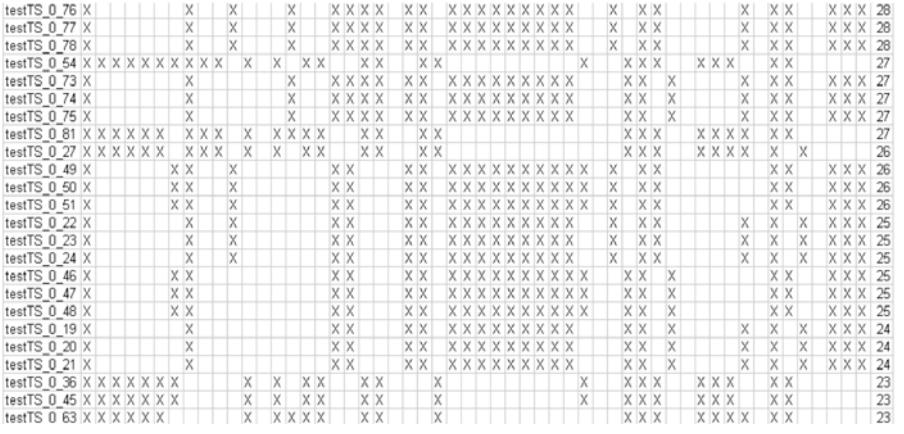

Fig. 6. A piece of the killing matrix for the original test suite, in the Jeffrey and Gupta example

4 Experiments

In order to check the results of the technique, two experiments were carried out:

The first experiment uses a set of programs which have been used in many research papers on software testing (for example, references [8, 11, 28-30]) and, therefore, can be considered as benchmarks. These programs are *Bisect, Bub, Find, Fourballs, Mid* and *TriTyp*. They are implemented as isolated Java classes. For this experiment, all the *traditional* mutation operators of the MuJava tool [31] were used to generate mutants (thus, those operators which are specific for the object-oriented characteristics of the Java language were excluded). Then, the *All combinations* strategy was used to generate test cases to kill these mutants. *All combinations* generates test cases by combining all test values of all the parameters [32].

For the second experiment, several classes were selected from a set of industrial systems. A required characteristic of the systems for this experiment is the availability of both its source code and test cases. In this case, the goal of the research is to evaluate the goodness of the technique in "actual" software. The test cases used for this experiment were those available with the corresponding project, and did not add new test cases to the test suites. All these projects are composed of several classes with different types of relationships among them. As with the first experiment, only the *traditional* operators were applied.

```
public class JGExample {
   float returnValue;
   public float f(float a, float b, float c, float d) {
        float x=0, y=0;
        if (a>0) x=2;
        else  x=5;
        if (b>0) y=1+x;
        if (c>0)
            if (d>0) returnValue=x;
            else returnValue=10;
        else returnValue=(1/(y-6));
        return returnValue;
   }

   public String toString() {
        return "" + returnValue;
   }
}
```

Fig. 7. The "motivational example" from Jeffrey and Gupta

4.1 Experiment 1: Benchmark Programs

Table 4 gives some quantitative information about the benchmark programs: number of lines of code (LOC), number of non-equivalent mutants generated by the MuJava tool (i.e., the equivalent mutants were manually removed), size of the original test suite (automatically generated using the *All combinations* strategy with the *testooj* tool) and size of the reduced suite. Of course, both the original and the reduced suite kill 100% of non-equivalent mutants and, thus, all of them are mutation-adequate.

Table 4. Results for the benchmark programs

Program	LOC	# of mutants	|Test suite|	|Reduced test suite|
Bisect	31	44	25	2 (8%)
Bub	54	70	256	1 (0,04%)
Find	79	179	135	1 (0.07%)
Fourballs	47	168	96	5 (5,2%)
Mid	59	138	125	5 (4%)
TriTyp	61	239	216	17 (7,8%)

4.2 Experiment 2: Industrial Programs

For this experiment, a set of publicly available programs was downloaded (Table 5 shows some quantitative data, measured with the Eclipse Metrics plugin). One important requirement for selecting these programs was the availability of test cases, since now the goal was to check whether among these test cases, written by the developers of the programs, there also exists any redundancy and that they can, therefore, be reduced.

In this way, the three following systems were selected and downloaded:

1) *jtopas,* a system for analyzing and parsing *HTML* pages with *CSS* code embedded. This system is included in the Software-artifact Infrastructure Repository (SIR), a website with a set of publicly available software, left by Do and others to facilitate benchmarking in testing experimentation [33].
2) *jester,* a testing tool for Java. This was developed by Ivan Moore and can be downloaded from *http://jester.sourceforge.net.*
3) *jfreechart,* a free chart library for the Java platform. It was designed for use in applications, applets, servlets and JSP.

Table 5. Quantitative data from the selected projects

Project	# of packages	# of classes	WMC (total)	LOC
jtopas	4	20	747	3,067
jester	7	67	588	3,225
jfreechart	69	877	20,584	124,664

Table 6 shows some quantitative data from the selected classes, all of them having a corresponding testing class. Table 7 shows:

1) The number of mutants generated for the class by the MuJava tool. Note that, in these projects, equivalent mutants have not been removed.
2) The number of available test cases for that class in the corresponding project.
3) The percentage of mutants killed by the original test suite.
4) The number of test cases in the reduced suite, once the original suite has been executed and the reduction algorithm has been applied.

Thus, for example, MuJava generates 94 mutants for *PluginTokenizer* from the *jtopas* project, where there are 14 test cases available on its website. These test cases kill 31% of the mutants. The last column shows the results of applying the test suite reduction algorithm, with the result that a single test case is sufficient for reaching the same coverage as the original 14 test cases.

Table 6. Quantitative data from the selected classes

Project	Program	LOC	WMC	Methods
jtopas	PluginTokenizer	157	27	16
jester	IgnoreList	26	8	3
jfreechart	CompositeTitle	63	14	8
	SimpleHistogramBin	117	32	11
	RendererUtilities	137	29	3
	XYPlot	2,470	591	194

Table 7. Test execution results

Program	Mutants	|Test suite|	Mutants killed	|Reduced test suite|
PluginTokenizer	94	14	31%	1 (7%)
IgnoreList	27	6	85%	2 (33%)
CompositeTitle	9	4	77%	1 (25%)
SimpleHistogramBin	293	5	66%	4 (80%)
RendererUtilities	801	6	81%	6 (100%)
XYPlot	3,012	21	36%	14 (66%)

5 Conclusions and Future Work

This article has presented a greedy algorithm to reduce the size of a test suite with no loss of quality, meaning that the new suite preserves the same coverage that the original suite reaches in the system under test. The testing criterion used to select the test cases is based on the percentage of mutants that each test case kills. The algorithm has been formally verified.

Moreover, it has been applied to both benchmark programs commonly used in testing research papers, as in industrial software, evidencing the utility of the algorithm in almost all cases.

In conclusion, the authors consider that mutation testing has reached sufficient maturity to be applied in the actual testing of real software. The research has now arrived at such a point that the knowledge produced over all these years is ready to be transferred to industrial testing tools.

As a future work, we plan to compare the presented reduction algorithm with the other presented in the literature, in order to determine differences of effectiveness and efficiency between them.

References

1. DeMillo, R., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. IEEE Computer 11(4), 34–41 (1978)
2. Barbosa, E.F., Maldonado, J.C., Vincenzi, A.M.R.: Toward the determination of sufficient mutant operators for C. Software Testing, Verification and Reliability 11(2), 113–136 (2001)
3. Offutt, A.J., Craft, W.: Using compiler optimization techniques to detect equivalent mutants. Software Testing, Verification and Reliability 7, 165–192 (1996)
4. Hamlet, R.: Testing programs with the help of a compiler. IEEE Transactions on Software Engineering 3(4), 279–290 (1977)
5. Mresa, E.S., Bottaci, L.: Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. Software Testing, Verification and Reliability 9, 205–232 (1999)
6. Wong, W.E., Mathur, A.P.: Reducing the Cost of Mutation Testing: An Empirical Study. Journal of Systems and Software 31(3), 185–196 (1995)

7. Untch, R., Offutt, A., Harrold, M.: Mutation analysis using program schemata. In: International Symposium on Software Testing, and Analysis, Cambridge, Massachusetts, June 28-30, pp. 139–148 (1993)
8. Offutt, A.J., Lee, S.D.: An Empirical Evaluation of Weak Mutation. IEEE Transactions on Software Engineering 20(5), 337–344 (1994)
9. Reales, P., Polo, M., Offutt, J.: Mutation at System and Functional Levels. In: Third International Conference on Software Testing, Verification, and Validation Workshops, Paris, France, pp. 110–119 (April 2010)
10. Hirayama, M., Yamamoto, T., Okayasu, J., Mizuno, O., Kikuno, T.: Elimination of Crucial Faults by a New Selective Testing Method. In: International Symposium on Empirical Software Engineering (ISESE 2002), Nara, Japan, October 3-4, pp. 183–191 (2002)
11. Polo, M., Piattini, M., García-Rodríguez, I.: Decreasing the cost of mutation testing with 2-order mutants. Software Testing, Verification and Reliability 19(2), 111–131 (2008)
12. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification and Reliability 7(3), 165–192 (1997)
13. Baudry, B., Fleurey, F., Jézéquel, J.-M., Traon, Y.L.: Automatic test case optimization: a bacteriologic algorithm. IEEE Software 22(2), 76–82 (2005)
14. Kim, S.W., Clark, J.A., McDermid, J.A.: Investigating the effectiveness of object-oriented testing strategies using the mutation method. Software Testing, Verification and Reliability 11, 207–225 (2001)
15. Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press (2008)
16. Polo, M., Piattini, M., Tendero, S.: Integrating techniques and tools for testing automation. Software Testing, Verification and Reliability 17(1), 3–39 (2007)
17. Jones, J.A., Harrold, M.J.: Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering 29(3), 195–209 (2003)
18. Garey, M.R., Johnson, D.S.: Computers and Intractability. W.H. Freeman, New York (1979)
19. Harrold, M., Gupta, R., Soffa, M.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2(3), 270–285 (1993)
20. Jeffrey, D., Gupta, N.: Test suite reduction with selective redundancy. In: International Conference on Software Maintenance, Budapest, Hungary, pp. 549–558 (2005)
21. Tallam, S., Gupta, N.: A concept analysis inspired greedy algorithm for test suite minimization. In: 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 35–42 (2005)
22. Heimdahl, M., George, D.: Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In: 19th IEEE International Conference on Automated Software Engineering, pp. 176–185 (2004)
23. McMaster, S., Memon, A.: Call Stack Coverage for Test Suite Reduction. In: 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, pp. 539–548 (2005)
24. Runeson, P., Andersson, C., Höst, M.: Test processes in software product evolution -a qualitative survey on the state of practice. Journal of Software Maintenance and Evolution: Research and Practice 15(1), 41–59 (2003)
25. Geras, A.M., Smith, M.R., Miller, J.: A survey of software testing practices in Alberta. Canadian Journal of Electrical and Computer Engineering 29(3), 183–191 (2004)
26. Ng, S.P., Murnane, T., Reed, K., Grant, D., Chen, T.Y.: A Preliminary Survey on Software Testing Practices in Australia, Melbourne, Australia, pp. 116–125 (2004)

27. Polo, M., Reales, P.: Mutation Testing Cost Redution Techniques: A Survey. IEEE Software 27(3), 80–86 (2010)
28. Offutt, A.J., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology 5(2), 99–118 (1996)
29. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-Data Generation Using Genetic Algorithms. Software Testing, Verification and Reliability 9(4), 263–282 (1999)
30. Offut, A.J., Pan, J., Zhang, T., Terwary, K.: Experiments with data flow and mutation testing. Report ISSE-TR-94-105 (1994)
31. Ma, Y.-S., Offutt, J., Kwon, Y.R.: MuJava: an automated class mutation system. Software Testing, Verification and Reliability 15(2), 97–133 (2005)
32. Grindal, M., Offutt, A.J., Andler, S.F.: Combination testing strategies: a survey. Software Testing, Verification and Reliability 15, 167–199 (2005)
33. H., D., Elbaum, S.G., Rothermel, G.: Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empirical Software Engineering: An International Journal 10(4), 405–435 (2005)