Validating Second-Order Mutation at System Level

Pedro Reales Mateo, Macario Polo Usaola, and José Luis Fernández Alemán

Abstract—Mutation has been recognized to be an effective software testing technique. It is based on the insertion of artificial faults in the system under test (SUT) by means of a set of mutation operators. Different operators can mutate each program statement in several ways, which may produce a huge number of mutants. This leads to very high costs for test case execution and result analysis. Several works have approached techniques for cost reduction in mutation testing, like n-order mutation where each mutant contains n artificial faults instead of one. There are two approaches to n-order mutation: increasing the effectiveness of mutation by searching for good n-order mutants, and decreasing the costs of mutation testing by reducing the mutants set through the combination of the first-order mutants into n-order mutants. This paper is focused on the second approach. However, this second use entails a risk: the possibility of leaving undiscovered faults in the SUT, which may distort the perception of the test suite quality. This paper describes an empirical study of different combination strategies to compose second-order mutants at system level as well as a cost-risk analysis of n-order mutation at system level.

Index Terms—Empirical evaluation, high-order mutation, mutation testing

1 Introduction

Testing tasks are essential for software developers in order to assure quality of their products and systems. The goal of testing is to identify errors in the systems to fix them. Simplifying, testers design tests that exercise functionalities of the system under test (SUT) and check whether the obtained results are equal to the expected ones. Therefore, a test suite (a set of tests) is better when is able to find more errors.

However, testing tasks are usually expensive in time and resources. Thus, effective testing techniques and methodologies are required in order to reduce costs.

In the literature, several authors have proposed so many techniques over the years to improve testing tasks, to reduce their costs, or to improve their effectiveness: white-box techniques use coverage criteria to know the areas of the code which are executed by test cases, many criteria exist to this end. On the other side, black-box techniques are based on the observation of the outputs that the system gives to the inputs, with no regard for what pieces of code are executed.

There are several levels for doing testing: unit (which is focused on finding faults on small parts of the system, such as classes or methods), integration testing (checks the interaction between units), or system testing (which tests the behavior of the system with its environment).

- *P. Reales Mateo and M. Polo Usaola are with the University of Castilla-La Mancha, Paseo de la Universidad, 4, Ciudad Real 13071, Spain. E-mail: {pedro.reales, macario.polo}@uclm.es.*
- *J.L. Fernández Alemán is with the Facultad de Informática, University of Murcia, Campus de Espinardo, Murcia 30100, Spain. E-mail: aleman@um.es.*

Manuscript received 18 Feb. 2011; revised 9 May 2012; accepted 18 May 2012; published online 5 June 2012.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-02-0047.
Digital Object Identifier no. 10.1109/TSE.2012.39.

This paper is focused on functional testing at the system level, which is focused on finding errors in the system functionalities. The paper is about a concrete testing technique, mutation testing. With mutation testing, small syntactic changes are introduced in copies of the system under test in order to create tests that are able to find those syntactic changes. Recently, Jia and Harman have published a complete and comprehensive survey about mutation testing [1].

The paper analyzes their application at system level and the goodness of *n-order* mutation (a variation of mutation testing) in multiclass systems.

The paper is organized as follows: Since the technique is applied, illustrated, and validated for multiclass systems, Section 2 describes some background related to mutation at system level. Section 3 provides an overview of other works analyzing higher order mutation (HOM) testing. Sections 4 and 5, respectively, describe the research methodology and the experimental results obtained. Finally, we draw our conclusions and present possible lines of future work.

2 Background on Mutation Testing

Mutation is a technique for software testing based on discovering the faults artificially introduced in copies of the system under test, which are called "mutants" [2]. Mutants result from the application of "mutation operators" to the original program, which introduce syntactic changes in its code that, in most cases, can be interpreted as faults (there are also "functionally equivalent mutants," whose behavior cannot be distinguished from that of the original system). Suppose the piece of code shown in the *Original* row in Table 1 is the code of a small program to be tested: The application of the *Arithmetic Operator Replacement* mutation operator produces mutants 1, 2, and 3, whereas the *Unary Operator Insertion* operator produces mutant 4.

TABLE 1
A Fragment of a Supposed System under Test and Four of Its Mutants

Version	Code

Original	int sum(int a, int b) {
	return a + b;
	}

Mutant 1	int sum(int a, int b) {
	return a - b;
	}

Mutant 2	int sum(int a, int b) {
	return a * b;
	}

Mutant 3	int sum(int a, int b) {
	return a / b;
	}

Mutant 4	int sum(int a, int b) {
	return a + b++;
	}

In the case of mutants 1, 2, and 3, the change introduced by the operator is actually a fault with respect to the original implementation of the program: In fact, a test case made up of the pair (1, 1) discovers the three faults since the original program returns 2, and these mutants, respectively, return 0, 1, and 1. The change introduced in mutant 4, however, executes a post-increment of b after having returned the sum of a and b, which prevents observation of any different behavior. Thus, the change can be considered a code de-optimization instead of a fault, and leads to this mutant being marked as equivalent.

Taking into account that the number of equivalent mutants may be quite high (Polo et al. report 18.66 percent in some benchmark programs [3]), that their detection is usually performed by hand (although there are several studies related to automated detection [4], [5]), and that, in some cases, a mean time of 15 minutes can be required to uncover their equivalence [6], equivalent mutants can be considered as "noise," which hampers the stage of result analysis. Furthermore, a test suite is "mutation-adequate" when its "Mutation Score" (Fig. 1. Mutation score) is 1, and reaching this value can be considered as the stopping condition for the mutation testing process: Thus, the reduction in the number of equivalent mutants is important to alleviate the costs of mutation testing.

The mutation process in Fig. 2 was proposed by Amman and Offutt [2], and requires automated tools to be successfully carried out: Bold boxes represent automatic steps in the process. Although Step 3 (*Remove equivalent mutants*) is theoretical and computationally an undecidable problem (to take a decision means executing the program with the full range of values), in practice some techniques exist to detect them: By annotating the code with constraints, Offutt and Pan detect almost 50 percent [4]; Schuler and Zeller use a statistical method to determine the probability of a mutant being equivalent [5]. Ideally, the process requires the automated generation of test cases to kill mutants. Then, the ineffective tests are eliminated (a test $t \in T$ is considered ineffective when $MS(P,T) = MS(P,T-\{t\})$, i.e., when it does not kill any additional mutants). The costs of this process involve computational costs (mutant creation and execution, and calculation of the mutation score) and human costs (tool configuration and result analysis, which involves the detection of equivalent mutants).

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

Fig. 1. Mutation score.

2.1 High-Order Mutation

Several techniques have been developed to reduce the costs of mutation and to improve its effectiveness (a very recent and complete survey is shown in [7]). One of them is a recently proposed and explored technique, *high-order* mutation [3], [8], which consists of the insertion of n faults in the mutant instead of 1. With $M = \{m_1, m_2, \ldots, m_p\}$ being a set of first-order mutants, a generation of second-order mutants can be built combining m_1 and m_2 into $m_{1,2}$ (which holds the faults of m_1 and m_2), m_3 and m_4 into $m_{3,4}$, etc. Two approaches of high-order mutation have recently been studied.

The first approach, studied by Polo et al. [3] and Papadakis et al. [9], tries to reduce the costs of mutation testing by reducing the number of mutants through the combination of first-order mutants into second-order (or higher) mutants. This way, the number of mutants is reduced to half and, since the mean number of first-order equivalent mutants is around 18-20 percent, the presence of second-order equivalent mutants is also significantly decreased (the probability of randomly combining two functional equivalent mutants is approximately $1/5 * 1/5 = 1/25 = 4\%$).

The goal of the second approach, studied by Harman et al. [10], is the improvement of the effectiveness of mutation testing through the creation of higher order mutants that simulate better errors than first-order mutants. Thanks to this

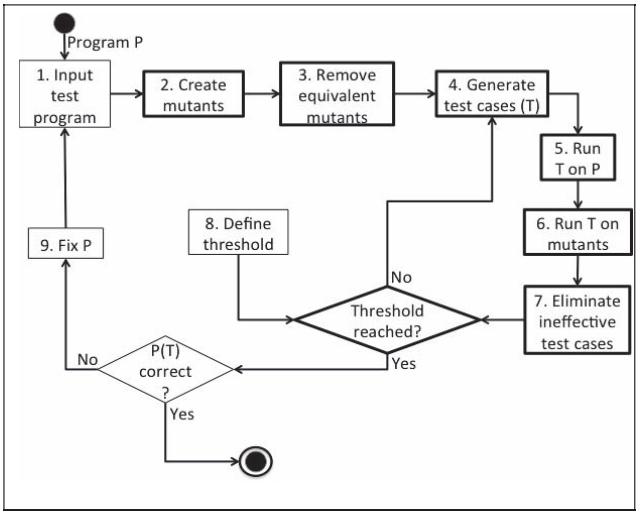


Fig. 2. Mutation process.

SUT ={A, B, C, D}
M_A={A_1, A_2, … A_P}
M_B={B_1, B_2, … B_Q},
M_C={C_1, C_2, … C_R}
M_D={D_1, D_2, … D_S}

Fig. 3. Class mutants of the supposed system.

kind of high-order mutation, it is possible to design better tests that will allow finding more real errors.

This paper is focused on the first approach: using high-order mutation in order to reduce costs of mutation analysis. Since several authors have analyzed the proneness of different combination algorithms to generate equivalent mutants [3], [11], this paper evaluates and discusses several strategies to combine first-order mutants into second-order mutants. The underlying idea of the *high-order* mutation approach studied in this paper is to reduce the costs of mutation testing through smaller mutant sets; however, this practice involves a risk that must be adequately measured: A second-order mutant m may be killed by a test case t, but it may happen that t has only discovered one of the two faults inserted in m. Thus, in a very bad case, a Mutation Score of 1 in a second-order mutant suite could even correspond to 0.5 in the original first-order suite if only half of the faults are discovered.

Historically, mutation has been applied at the unit level, that is, faults are inserted in a concrete unit (i.e., a class). At the unit level, a tester is focused on design test cases to test concrete units; thus the mutation score is calculated taking into account only the mutants of that unit. However, at the system level, a tester is focused on designing tests to check other features, like interactions between components, with other systems or users and complete functionalities. Therefore, if we apply mutation at the unit level, there are features that can be difficult to test, such as concrete interactions among units, configurations, or interactions between systems.

This study analyzes n-order mutation in multiclass systems, where faults can be inserted in different classes of a system thanks to our supporting tool, Bacterio, which has a specific module for generating mutant versions of the whole application, and not only of a single class (like other tools).

3 Mutation at Multiclass and System Levels

Suppose S is a system made up of four classes $\{A, B, C, D\}$. Applying mutation at unit level, the tester will generate mutants for one of these classes (for example, A) and will focus on writing a test suite T_A to reach a prefixed value of the mutation score on A: Supposing this prefixed value is 1, the tester will stop when

$$MS(A, T_A) = 1.$$

When this happens, s/he continues with the remaining classes generating mutants for B, C, D and writing test cases to obtain an MS of 1 with test suites T_B, T_C, and T_D.

Assuming that the artificial faults are good enough, a process like this produces enough tested units. However, since the tester's attention is focused on units, testing the behavior of other system characteristics, such as the sequence of interactions among units or whole functionalities, is beyond the scope of the process.

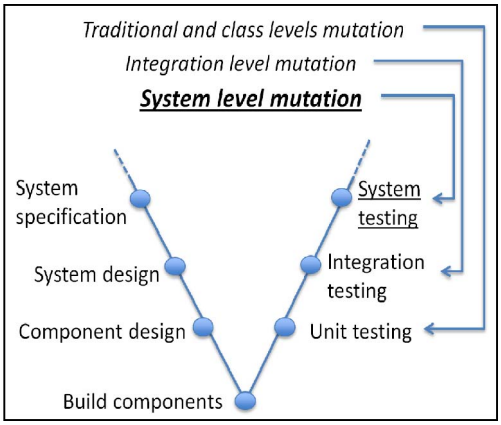


Fig. 4. Mutation at the system level.

Suppose the mutation tool produces the set of P, Q, R, and S mutants for the four classes in the system shown in Fig. 3.

We define a *system mutant* or a *mutant version* as a copy of the SUT which contains at least one faulty class or module.

To kill a mutant version, the tester must now focus on getting the execution of a given functionality to lead the mutant version into a state different to the original version. That is, the executed scenario must reach the mutated statement, which may be relatively far from the interface directly exercised by the test suite.

This type of mutation, proposed by Reales et al. [12], has an important benefit and perfectly complements other kinds of mutation such as traditional mutation and integration mutation. While in traditional mutation the tester focuses on finding errors in classes or modules, in system level mutation the tester is focused on finding errors in complete functionalities or in the complete system, which is usually made up of multiple elements which interact with each other. This new point of view helps the tester to find other kinds of errors that can go unnoticed (Fig. 4).

Since mutation at the system level complements other kinds of mutation, it is important to perform experiments to see if the technique used with traditional mutation is also effective, which is the focus of the study presented in this paper.

3.1 First-Order System-Level Mutation

In the example, a first-order mutated version of the system consists of three original classes and one mutant class, for example: $M_1 = \{\mathbf{A}_1, B, C, D\}$, $M_2 = \{A, \mathbf{B}_1, C, D\}$, etc. In this way, there are up to $P + Q + R + S$ mutant versions of the system.

Now, a test case may not directly interact with a simple class mutant (i.e., there is no specific test suite T_A for testing A or T_B for testing B), but it interacts with the whole application, perhaps dealing with B mutated instances from messages sent from the test suite to A (Fig. 5). Thus, the observability of test case results is different, and the tester's perspective in designing test cases also differs: The goal, now, is to observe state differences between the behavior of the complete original system and the complete mutant version.

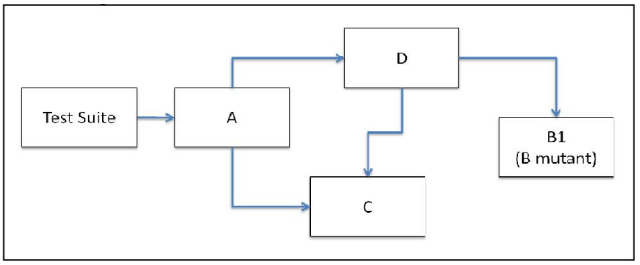


Fig. 5. Indirect use of a mutant from the test suite.

3.2 Second-Order System Level Mutation

At system level, a second-order mutant version is a system version containing two faults. There are two different types of mutant versions: interclass and intraclass.

Interclass mutants. A second-order mutant can be generated with the introduction of two mutant classes in a mutant version. Continuing with the example, a possible mutant version could include a mutant from A and a mutant from B, another could contain a mutant from A and a mutant from C, etc. (Fig. 6).

Since the faults are inserted in two different classes, we call these versions "interclass mutants."

Intraclass mutants. Another way to produce second-order mutants is with the inclusion of a second-order class mutant in each mutant version: Supposing $A_{1,2}$ is a mutant proceeding from the combination of the faults introduced in A_1 and A_2; a second-order mutant version of the system could be made up of $A_{1,2}$, B, C, and D (Fig. 7).

The main difference of high-order mutation at system level with respect to unit level is that the faults introduced in an interclass high-order mutant can be very far apart (i.e., in different classes relatively far one from the other, from the path length execution point of view). This introduces a higher risk of missing faults because this distance (and, maybe, the complete independence of execution paths separating the two faults) increases the probability of finding only one fault. At unit level, the separation between the two faults is much more reduced, making it easier to kill these mutants.

3.3 Equivalent Versions

If the change introduced in the mutated class is not a fault (i.e., it is an equivalent mutant), then the mutated version will also be functionally equivalent to the original system and it will not be possible to kill it.

Several authors have proposed different techniques to identify equivalent mutants, all of them being manual or semi-automatic. From an algorithmic point of view, the detection of equivalent mutants is an undecidable problem [13].

In 1979, Baldwin and Sayward [14] proposed the use of compiler optimization techniques to detect equivalent mutants since an equivalent mutant can be consider a optimization or deoptimization of the original program (the code optimizer substitutes, for example, $a = b * 1$ by $a = b$).

Another technique proposed by Offutt and Pan [15] is based in constraint solving. The tester must define constraints for the input data. Then, a constraint solving system determines whether a mutant is equivalent. In their studies, Offutt and Pan automatically found 45 percent of the equivalent mutants from 11 systems.

$M1=\{\mathbf{A_1}, \mathbf{B_1}, C, D\}$
$M2=\{\mathbf{A_1}, B, \mathbf{C_1}, D\}$

Fig. 6. Two second-order interclass mutant versions.

Program slicing was proposed by Hierons et al. [16] to assist in the detection of equivalent mutants. This technique is able to show only those parts of the code that can be affected by a previous statement. This is very useful for a tester to identify equivalent mutants.

The most recent technique was proposed by Schuler and Zeller [5] and it is based on the impact of mutants (how a mutation impacts the values of the variables). This technique is able to automatically determine the probability that a mutant is equivalent, but the tester finally has to decide which mutants are or not equivalent.

All these techniques were proposed and experimented with at unit level. Only the last one is perfectly suitable to work at system level because it is based on executions through several methods and classes. But, with small adaptations, some of the other techniques could be used at system level.

At system level, test cases check complex interactions among classes; therefore, the detection of equivalence can be more difficult and costly than at unit-level mutation.

Thus, mutation at system level is an especially suitable context for applying *high-order* mutation to reduce the number of mutants and thus the number of equivalent mutants [3], [11] or, in the context of this paper, the number of equivalent versions.

When two mutations are combined into one mutant (a second-order mutant) there are two possibilities:

1. The resulting second-order mutant is a nonequivalent mutant that can be killed. This is due to the fact that the combined effect of the two mutations will show unexpected results. This normally happens when two nonequivalent mutants are combined or when a nonequivalent mutant is combined with an equivalent mutant. It is also possible to create a second-order nonequivalent mutant by combining two equivalent mutants, but the probability is very small.
2. The resulting second-order mutant is an equivalent mutant. This is due to the fact that the combined effect of the two mutations will not show unexpected results. This normally happens when two equivalent mutations are combined. It is also possible to create a second-order equivalent mutant by combining two mutations so that the effect of the first one is cancelled by the second, although, here too, the probability is very small.

4 Working at Multiclass and System Levels

Working at system level requires appropriated technologies and tools. To perform the experiments presented in this

$M1=\{\mathbf{A_{1,2}}, B, C, D\}$

Fig. 7. A second-order intraclass mutant version.

paper, we used flexible weak mutation (FWM), which is specially designed to work at system level [12], as well as the Bacterio mutation tool, which implements flexible weak mutation and allows testers to perform mutation analyses at multiclass and system levels. Both are described in the next sections.

4.1 Flexible Weak Mutation

In order to decide whether a mutant version is killed or alive, the test case must be capable of observing the whole version state during its corresponding execution scenario.

At system level, this can be achieved by means of *Flexible Weak Mutation* [12], a recently proposed technique which almost continuously inspects the states of all the objects involved in the execution scenario; in its current implementation (in the Bacterio tool), both the code of the original and of the mutants is instrumented to leave a trace of each object state in a *log* file. An execution engine looks for differences between the traces of the SUT and of the mutant. Each object leaves its trace according to the tester's preferences:

1. invoking its $toString()$ method, if it is implemented in the self class,
2. inspecting its field values,
3. with a specific implementation of a *hashCode* function, which is calculated from the field values,
4. invoking $toString()$ if it is implemented and using field inspection otherwise,
5. invoking $toString()$ if it is implemented and using the *hashCode* function otherwise.

4.2 Bacterio: A Tool for Mutation at System Level

One of the most important elements to perform a mutation analysis is the mutation tool. A mutation tool can automate the mutant generation, mutant execution, and result analyses tasks to a greater or lesser extent.

Bacterio [12] is a recently developed Java mutation tool that implements several mutation techniques to reduce mutation costs and increase the automation of these tasks. This is the first tool that implements Flexible Weak Mutation, making it possible to perform mutation analyses at system level. Also, this tool implements Strong mutation and two variants of Weak mutation, so it can be used to perform traditional mutation analyses as well.

Bacterio also implements several cost reduction techniques: Selective mutation [17], [18], [19], Mutant Sampling [20], [21], [22], Mutant Schemata [23], and Bytecode translation [24]. Likewise, a recent extension of Bacterio includes an advanced mutant execution architecture that supports remote parallel executions. This architecture distributes the working load among different heterogeneous hosts in a network to execute test cases in parallel.

An important feature of Bacterio, for the experiment presented, is that this tool implements high-order mutation. This tool is able to create mutants with $1, 2, 3, \ldots$ or n errors automatically. To make these high-order mutants, Bacterio has a set of combination algorithms (Each Choice [25], FirstToLast [3], Random Each Choice [3], and Pairwise [26], some of which are described in Section 4.4.1) and a set of restrictions which can be selected along with the algorithms (combine mutants only between operators, do not combine errors in the same class and do not combine errors in the same method).

Due to the features presented in Bacterio, this tool has been selected to perform the mutation analyses included in the experiment presented in this work.

5 Other Studies on Higher Order Mutation

A small number of studies have conducted research on the application of different strategies to produce higher order mutations.

The first study related to higher order mutation investigated the mutant coupling effect [27]. This study determined that, in structured languages, test cases produced for first-order mutants were sufficiently capable of killing most of the higher order mutants. It would be interesting to check the validity of this result with new paradigms such as object-oriented programming.

After Offutt's work, no research about high-order mutation was performed until 2008. That year [8], and later in 2009 [1], Jia and Harman advocated the use of search-based optimization techniques as a means of efficiently exploring the space of Higher Order Mutants, guided by a fitness function that seeks to capture mutant "quality." These authors proposed a new high-order mutation technique based on the following concept: "A high-order mutant only is killed if all the introduced faults are found," and presented an experiment where the authors found high-order mutants that were more difficult to kill than the first-order mutants used to compose the higher order mutant. These mutations (combined mutations are more difficult to kill) are named *subsuming higher order mutation*.

In the experiment presented, the authors used 10 C benchmark programs, 23 C mutation operators according to the five effective operator categories proposed by Offutt et al. [18], and the tool *Milu* [28], specially designed to study higher order mutations in C. To find subsuming high-order mutations, four algorithms were used: a greedy algorithm, a genetic algorithm, a hill climbing algorithm, and a random algorithm. The relevant findings were: 1) The subsuming higher order mutations are generally numerous (between 47.0 and 89.5 percent in the programs studied), 2) there is a tendency toward larger decoupling as the order of higher order mutations increases (for all programs studied), and 3) the rate of strongly subsuming higher order mutations was low (between 0.01 and 0.4 percent in the programs studied).

In another recent work, Polo et al. [3] studied the influence of the number of high-order mutants on the costs of mutation at unit level. The authors presented three algorithms to combine first-order mutants in order to make higher order mutants: *LastToFirst*, *DifferentOperators*, and *RandomMix*. Two experiments were conducted to evaluate the influence of the combining algorithm on the number of higher order equivalent mutants.

In the experiments, nine Java programs were used: six research benchmark applications and three industrial systems. The tools used were *Mujava* [24] to create the first-order mutants using traditional mutation operators and *testooj* [29] to generate higher order mutants using the three combination algorithms and executing them to perform the experiment. The results of the experiment showed

that the number of equivalent mutants can be significantly reduced by using second-order mutation (from 18 to 5 percent). Since each mutation operator has a different proneness to produce equivalent mutants, the best reduction was found with the *DifferentOperators* combination algorithm. However, this algorithm generated a larger number of second-order mutants (60 percent) than the two others (50 percent). In the further cost and risk analysis, the authors show that high-order mutation may be effective even with six faults inserted per mutant.

Papadakis and Malevris [11] presented a study to evaluate second-order mutation at unit level. In this work, the authors conducted an experiment to evaluate the effectiveness and the efficiency of second-order mutation. The experiment involved eight industrial programs written in the C programming language: seven programs from the Siemens suite and the Space program [30] (all of these are available with faults and a large number of test cases). To perform the experiment, Delamaro et al. used 44 mutation operators for C and the *Proteum* mutation tool [31]. Due to the high number of obtained mutants, only 10 percent of the generated mutants were selected for the experiment.

In the experiment, six different sets of mutants were made using the mutant sampling technique [20] (the sets contained 10, 20, 30, 40, 50, and 60 percent of the selected mutants). Then, the mutants from each set were combined to make six sets of second-order mutants. Seven combining strategies were applied obtaining a total of 42 sets of second-order mutants: two combination strategies, *RandMix* and *DiffOp*, adopted as proposed in [3], and five new strategies, *First2Last, SameNode, SameUnit*, *SU_F2Last*, and *SU_DiffOp*.

The findings of this study revealed that first-order mutation is more effective at detecting faults than second-order mutation. However, second-order mutation is more efficient than first-order mutation. Second-order mutation was less costly during the test design and execution phases since the number of produced mutants and required test cases decreased. Regarding equivalent mutants, their number was drastically reduced, depending on the combination strategy used to make second-order mutants (between 80 and 90 percent). As a result of the experiment, the authors concluded that techniques that demand more resources might detect more faults than techniques that demand fewer resources.

Kintis et al. introduce a set of four second-order strategies [32]: Relaxed Dominator strategy (RDomF), Strict Dominator strategy (SDomF), and two Hybrid Strategies, which try to jointly combine second- and first-order mutation approaches. These strategies were chosen from previous research [11]. The proposed strategies make use of the dominance relation between program nodes in order to construct mutant pairs with a high chance of coupling [33]. Three experiments were performed using the MuJava mutation testing tool [24] and a set of 15 Java program test units. First-order mutants were created using all the MuJava method-level operators, while second-order and weak mutation mutants were generated based on the strategies proposed. The experimental results suggested that both the second-order mutation testing strategies and weak mutation are quite effective as they result in a relatively high strong mutation coverage score (between 96 and 98 percent). Furthermore, this experimental study found evidence that weak mutation is, in general, not comparable with strong as both achieve a 97 percent collateral coverage on each other

Langdon et al. [34], [35], [36] present a new technique to make higher order mutants, called *multi-objective-order mutation testing*. This technique is based on a genetic algorithm that explores the syntactic and semantic distance of mutants with respect to a given test suite.

Syntactic distance is calculated by adding the number of changes weighted by the actual difference. Semantic distance is obtained by measuring the number of test cases that show different behaviors between the original program and a concrete mutant. A search-based approach with a genetic algorithm was used to find complex faults in the space of higher order mutants.

An experiment was conducted to evaluate the proposed genetic algorithm. Three applications were used: two real-world programs, the *schedule* and *tcas* benchmarks [37], and a simplified version of the well-known triangle program [38]. The study focused on the ROR mutation operator. The experiment confirmed that adding faults to a faulty program makes it more error-prone due to the fact that complex mutants are coupled to simple mutants (Mutation Testing Coupling Hypothesis [38]). However, in the *tcas* program, the algorithm was able to find higher order mutants that were harder to kill than any of the first-order mutants, thus partially refuting this hypothesis. The authors claim that the strategy used makes the approach scalable, thus mitigating the computational cost problem, while the identification of complex faults addresses the problem of fault realism. The study was recently completed by including the *gzip* benchmark in the experiment [34].

Kaminski et al. [39] studied subsuming higher order logic mutation as an approach to selective mutation for programs and queries. These authors explored the use of subsuming higher order logic mutants to address the selection of appropriate logic mutation operators, the reduction of the number of mutants, and the number of equivalent mutants. Both Java and SQL were employed as the source code to be tested. Their experiment showed that tests killing all the subsuming higher order mutants killed a high percentage of general mutants while reducing both the number of mutants and the number of equivalent mutants.

Kaminski et al. [40] reformulate the classic definition of the ROR operator [22], showing that, for any given clause, only three mutants are necessary, thus eliminating generation of 57 percent of the ROR mutants. The concept of higher order mutation operator is used to define Relational Operator Replacement Global (RORG), a mutation operator, with the aim of successfully integrating a syntactic criterion like ROR mutation with the semantic criterion of Multiple Condition-Decision Coverage (MCDC). Kaminski and Ammann [41] introduce a new logic criterion based on component criterion feasibility to reduce the number of test inputs needed while still guaranteeing that double faults (second-order mutants) are detected.

Just et al. [42] present a compiler-integrated approach called conditional mutation to reduce the time overhead for generating and executing the mutants. It is based on transforming the Abstract Syntax Tree (AST) to collect all mutants in conjunction with the original program within the resulting assembled code. The authors describe how to extend conditional mutation to support higher order mutation.

TABLE 2
Goal/Question Metric Template

Goal	To investigate the adequacy of n-order mutation at system level.
Question	Working at system level, do test cases designed for 2nd-order mutants have the same quality as test cases designed for 1st-order mutants?
Metric	The quality of the test suites measured in terms of mutation scores at system level.
Object of study	N-order mutation.
Purpose	To evaluate and improve the effectiveness of mutation testing.
Focus	To investigate the adequacy of 2nd-order mutation at system level.
Perspective	From the point of view of the researcher.
Context	At system level.

To the best of our knowledge, this is the first paper studying the adequacy of second-order mutation at system level by using the Java language. An empirical study by Purushothaman and Perry [43] found that 90 percent of postrelease faults are complex faults. Therefore, our study contributes to the software testing body of knowledge by providing evidence to understand the impact of second-order mutation on the effectiveness of testing mutation, consequently benefiting testers in terms of providing suggestions to construct test suites effectively.

6 Research Methodology

In this section, detailed information is provided on the procedure followed to design and conduct our experiment in order to evaluate the suitability of high-order mutation as opposed to first-order mutation.

6.1 Research Goals

The research goals were outlined using the Goal/Question Metric (GQM) framework [44]. The GQM template [45] of the experiment is shown in Table 2.

6.2 Hypothesis

In order to investigate the adequacy of n-order mutation at system level, we have to evaluate the quality of the tests developed with this technique. For the sake of simplicity, this experiment only focuses on second-order mutation. To evaluate the quality of the tests designed for a second-order analysis, we need to compare them with tests designed for a first-order analysis. Thus, the following null hypothesis is proposed:

- H_0: Tests designed for second-order analysis have a quality similar to tests designed for first-order analysis.

This contrasts with the following alternative hypothesis:

- H_1: Tests designed with second-order analysis have lower quality than tests designed for first-order analysis.

There is also another alternative hypothesis: "Tests designed with second-order analysis have higher quality than tests designed for first-order analysis." This is due to the fact that there can be some mutations which are more difficult to find by second-order mutants [8] due to masking.

However, this alternative hypothesis was directly rejected because the probability of creating those second-order mutants is very low. Moreover, high-order mutation is used in the experiments to reduce the mutants set (the first of the approaches described in Section 2.1), which reduces the probability even more. Thus, in case of any differences, the tests designed with the smaller mutants set will have a lower quality.

6.3 Variable Description

Measuring the adequacy of n-order mutation can be achieved using several metrics (e.g., mutation scores [2], fitness of mutants [8], missed errors [3], etc.). Since our study is aimed at investigating the quality of test suites designed using first-order mutation and second-order mutation, the metric selected was Mutation Score. Hence, the **dependent variable** of the presented experiment is the quality of the designed test suites measured in terms of the *Mutation Scores* (Fig. 1), which will be calculated with first-order mutants. Thus, the test suites designed with second-order mutation must be executed for first-order mutants to calculate the score, which will be compared to the score achieved by the tests designed for first-order mutants.

Our **independent variables** are related to the techniques used to perform the mutation analysis (first-order mutation and second-order mutation). Since we can use different combination algorithms, combination restrictions, and reduction algorithms to perform second-order mutation analyses, our independent variables are: 1) *Combination algorithm* used to create second-order mutants (one value is "no algorithm," which corresponds to first-order mutation), 2) *Combination restriction* (that is, interclass), which can be or not be applied, and 3) *Reduction algorithm* to obtain a mutation adequate test suite.

6.4 Instrumentation

This section describes the instruments used to perform the experiment.

6.4.1 Combination Algorithms and Restrictions

To perform this experiment, we used four **combination algorithms** to combine faults in order to obtain second-order mutants. The four algorithms start with a list of faults of first-order mutants, which will be combined. The procedure for each algorithm is presented in the following paragraphs. All of these assume the prior existence of a list of mutants.

FirstToLast:

1. Takes the first nonused mutant from the top of the list.
2. Takes the first nonused mutant from the end of the list. If there are no more nonused elements, it selects an element next to the element selected in Step 1.
3. Combines the two selected elements to create a new second-order mutant.

4. Marks the two elements as used.
5. This process is repeated until all the elements in the list are used.

Between-Operators:

1. Takes the first nonused mutant from the top of the list and determines the mutation operator used to make the fault.
2. Takes the first nonused mutant from the top of the list that has been generated with a different mutation operator than the previous selected element. If there are no unused elements with a different operator, selects the less used element with a different operator.
3. Combines the two selected elements to create a new second-order mutant.
4. Marks the two elements as used (if an element has been used before, increases the number of uses).
5. This process is repeated until all the elements in the list are used.

Random:

1. Randomly selects a nonused element from the list.
2. Randomly selects another nonused element from the list. If there are no more nonused elements, selects a used element randomly.
3. Combines the two selected elements to create a new second-order mutant.
4. Marks the two elements as used.
5. This process is repeated until all the elements in the list are used.

Each Choice:

1. Takes the first nonused mutant from the top of the list.
2. Takes another nonused mutant from the top of the list. If there are no more nonused elements, selects the element before the element selected in Step 1.
3. Combines the two selected elements to create a new second-order mutant.
4. Marks the two elements as used.
5. This process is repeated until all the elements in the list are used.

Since this experiment was performed at system level, we introduced a **combination restriction** to all the algorithms in order to disperse the faults throughout the system. This restriction means that the *combined errors must be inserted into different classes*. Under this restriction, situations where it is necessary to combine an unused with an already used fault can occur, so the algorithms were modified to support this. In the experiment, we made second-order mutants with each algorithm using the restriction and without using it, obtaining eight different sets of second-order mutants for each application.

6.4.2 Applications under Test

The experiment was performed with seven relatively large applications (Table 3).

- The first application (*Chess*) is a system implemented by software engineering students in their fifth year at the University of Castilla-La Mancha. This system implements a chess server which includes a chess engine that encodes all the chess rules, a database to store game information, a graphical user interface to manage the server, and a communication layer allowing clients to connect and play with each other.
- The next application (*Cine*) is a complete cinema management system with a facility to reserve locations. This system was developed by researchers in the Alarcos research group at the University of Castilla-La Mancha for the purposes of software engineering experimentation.
- Another application (*Monopoly*) implements the widely known board game. It was implemented by software engineering professors (fifth year computer science studies at the University of Castilla-La Mancha). The students use this system to check their skills in test writing after several classes on software testing.
- The fourth application (*ECal*) is a lightweight, web-based event information tracking application available at www.sourceforge.com.
- The fifth application (CitasMed) is a management system for medical appointments. It was implemented by a freelance software engineer and it is currently used at the internal medicine consultation service of Gutiérrez Ortega Hospital.

TABLE 3
Quantitative Application Description

Application	Number of classes	Lines of code	Number of available test cases
Chess	14	2,288	913
Cine	10	678	197
Monopoly	40	641	108
ECal	34	1,246	53
CitasMed	37	2,797	446

6.4.3 Test Suite Description

For the five applications, the test cases were implemented by third parties and written in JUnit. All of them show a pass verdict (that is, they highlight the JUnit green bar).

Although they contain redundant test cases, test suites of Cine, Monopoly, Chess, and CitasMed achieve a 100 percent of first-order mutation score, so the first-order mutants alive after the execution are equivalent mutants. The application downloaded from the Internet (*ECal*) is also distributed with test cases. Since it is publicly available, we believe that its developers considered this test suite as complete and adequate, which led to the release of this software.

To perform the experiment, we needed mutation-adequate **test suites** for first- and second-order mutation. In order to obtain them, we performed the mutation process described in Fig. 2, with some differences. First, we did not remove any equivalent mutant (Step 3 of the process) due to the fact that we did not have any tools that implement automatic techniques to identify them and we were working with more than 67,000 mutants (Table 5). Second, in Step 4, we selected all the already designed test cases instead of generating them automatically. Third, the threshold (Step 8) was determined as the mutation score achieved by the total set of existing test cases.

TABLE 4
Killing Matrix of a Supposed Program

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

In the process, in Fig. 2, Step 7 had a strong impact on the results of the experiment in this paper: A second-order mutant may be killed by two different test cases $(t1, t2)$, each finding a different error. In this situation, either $t1$ or $t2$ is considered ineffective and is removed from the test suite, losing the effectiveness of the mutation process.

In order to study second-order mutation in different situations and establish good practices to use with it, we implemented three greedy algorithms to reduce a test suite based on the mutants killed by the set.

These algorithms were inspired by the greedy test suite reduction algorithms from Gupta et al. [46], [47], [48], which obtain reduced test suites that preserve the test requirement (in our case, the mutation score) of the original suite: T being a test suite and $T_R \subseteq T$, $MS(P,T) = MS(P,T_R)$. In our case, since the systems' developers consider T as adequate and since T_R obtains the same Mutation Score as T, T_R is also adequate: Consider, for example, the killing matrix in Table 4, which shows a program with seven mutants and five test cases. The Mutation Score of this T is 1. A greedy algorithm can be applied to this matrix to obtain a reduced suite reaching the same Mutation Score. In fact, if we add $tc1$ and $tc5$ to T_R, we obtain a test suite with just two test cases, which is as adequate as T.

The three algorithms used in the experiment are as follows:

MAX algorithm:

1. Select the test case t that kills the **highest** number of mutants, remove t from the killing matrix, and add t to T_R.
2. Remove the mutants killed by t from the killing matrix.
3. Remove the test cases that do not kill any mutant from the killing matrix.
4. If the matrix contains tests, go to 1.

MIN algorithm:

1. Select the test case t that kills the **lowest** number of mutants, remove t from the killing matrix, and add t to T_R.
2. Remove the mutants killed by t from the killing matrix.
3. Remove the test cases that do not kill any mutant from the killing matrix.
4. If the matrix contains tests, go to 1.

RDM algorithm:

1. Select a test case t **randomly**, remove t from the killing matrix, and add to T_R.
2. Remove the mutants killed by t from the killing matrix.
3. Remove the test cases that do not kill any mutant from the killing matrix.
4. If the matrix contains tests, go to 1.

Due to the nature of the algorithms, it is expected that the MAX algorithm will reduce the set of test cases more than MIN since more mutants are killed with each test. And also, it is expected that the RDM algorithm will reduce the set of test cases more than MIN but less than MAX. These expectations are corroborated by the results of Tables 7, 9, and 11.

The nature of the killing matrix has a decisive influence on the reduced test suite obtained, and the matrix, in turn, depends on the mutation techniques used. For example, a matrix proceeding from the execution of a first-order mutant suite is probably different than another proceeding from second-order mutants. Furthermore, a matrix proceeding from the execution of second-order mutants created with a given combination algorithm is probably different from another proceeding from second-order mutants created with another combination algorithm.

Therefore, for each combination of *application*, *order*, *combination algorithm*, *restriction*, and *reduction algorithm,* we have to perform the mutation process as described in this section to obtain the mutation adequate test suite.

6.4.4 Tools

Due to the nature of this experiment and the applications used, we needed a set of **tools** for Java technology to perform the experiment. To manage the source code of the applications and the test cases, we selected the Eclipse platform [49].

To analyze the applications and obtain information about their metrics (LOC, number of classes, etc.) we used the Metrics plug-in for Eclipse [50]. Metrics shows descriptive information such as number of packages, classes, and methods; LOC; cyclomatic complexity, Weighted Method per Class (WMC), etc.

To perform mutation analyses, we used the Bacterio application [12] described in Section 2.5. This application implements Flexible Weak Mutation and can execute mutation analyses at system level, which is a prerequisite of the experiment. Also, this tool implements the combination algorithms to make the second-order mutants described in this section and supports high-order mutation.

Finally, to perform the statistics analysis and collect the data, we used SPSS [51]. This is a statistics application which supports the statistics necessary for our experiment.

6.5 Mutation Operators

The mutation operators used in the experiments were the operators supported by the Bacterio mutation tool: five interface mutation operators defined by Ghosh and Mathur [52] to mutate interactions between elements (SWAP, which swaps parameters; TEX, which forces the throwing of exceptions; INC and DEC, which increment and decrement numeric parameters; and NUF, which substitutes parameter objects by *null*), and five traditional mutation operators selected by Offutt in [18] (AOR: Aritmetic operator replacement; ROR, Relational operator replacement; UOI, unary operator insertion; LCR, logic connector replacement; and ABS, absolute value insertion).

$MS_{1\text{-}order}(S_i, T_{2\text{-}orderA}) \approx MS_{1\text{-}order}(S_i, T_{1\text{-}orderA}) \rightarrow 2^{nd}$-order mutation is adequate at system level.

Fig. 8. Possible adequacy of second-order mutation.

6.6 Experimental Procedure

The following paragraphs present the experimental procedure carried out for this experiment. The procedure has five steps.

1. We took the applications and generated first-order mutants.
2. These first-order mutants were combined into second-order with each combination algorithm twice: The first time we applied the combination restriction (*interclass*) and the second time we did not (*intraclass*).
3. For each application, we applied the MAX algorithm (Section 4.4.3) to the second-order killing matrix. Thus, from here we obtained new test suites as adequate as the original ones from a second-order point of view. Since there are four algorithms and two restrictions, for each application A we obtained eight reduced test suites for second-order mutants ($TS_{2\text{-order,A,MAX}}$) and one reduced test suite for first-order mutants ($TS_{1\text{-order,A,MAX}}$).
4. For each application A, we executed all the $TS_{2\text{-order,A,MAX}}$ against the first-order mutant sets. The mutation scores obtained can be compared with the scores achieved with the $TS_{1\text{-order,A,MAX}}$.
5. We performed Steps 1-4 with the MIN and the RDM algorithms (thus obtaining: $TS_{2\text{-order,A,MIN}}$, $TS_{2\text{-order,A,MIN}}$, $TS_{2\text{-order,A,RDM}}$, and $TS_{1\text{-order,A,RDM}}$). In the case of RDM, since it depends on chance, we repeated the execution three times and considered the mean of the mutation scores.
6. We compared the mutation score values and performed statistical tests in order to evaluate our hypothesis.

At this point we can consider that if the Mutation Score reached by $TS_{2\text{-order,A}}$ on first-order mutants is similar to the Mutation Score reached by $TS_{1\text{-order,A}}$ on the same mutant set, then designing test cases for second-order mutants is effective enough for testing applications (Fig. 8).

The Mutation Score that any test suite reaches on a second-order mutant set is always greater than the Mutation Score reached on the original first-order set since a second-order mutant dies when one of its two faults is found (actually, there is a very low probability of combining two nonequivalent mutants into an equivalent second-order mutant; the chances of combining two equivalent mutants into a nonequivalent second-order mutant are equally low). Thus, if a tester uses second-order mutation, her/his mutation scores should be significantly higher than with the first-order. This is due to the fact that:

1. There are fewer second-order equivalent mutants, because the probability of combining two first-order equivalent mutants is low.
2. Discovering one of the two faults inserted is usually enough to consider the mutant as killed.
3. The probability of combining two nonequivalent mutants into an equivalent second-order mutant is very low.

Since the reduction in the number of equivalent mutants can decrease down to 5 percent (with the first-order it is around 20 percent), the mutation score achieved with second-order mutation should be around 0.95, which is considered adequate if the equivalent mutants are not identified, which is the common practice.

6.7 Analysis Procedure

Since the aim of this study is to measure the effect of second-order and first-order mutation on mutation scores, we used the nonparametric Mann-Whitney test, which determines whether the means of two groups are significantly different. The signification level established in this test to reject H_0 was 0.05. If the result of the test had a signification level lower than 0.05, H_0 would be rejected.

In order to compare each combination algorithm and the interclass restrictions, we applied the Wilcoxon signed-rank test, which was designed to compare two related samples and determine if there are any significant differences between them. This is also a nonparametric test.

7 Results

This section shows the results obtained in the execution of the experiment. Table 5 shows the number of mutants obtained from each application (we handled 67,694 mutant versions in total). Note that BO = "Between Operators," EC = "Each Choice," FL = "First-Last," and Rm = "Random."

TABLE 5
Number of Mutants Generated

Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	2,040	1,127	1,101	1,081	1,062	1,213	1,020	1,020	1,020
	Chess	5,737	3,571	3,289	2,908	3,220	3,548	2,847	2,848	2,869
	Monopoly	1,250	743	641	634	656	713	625	625	625
	Ecal	711	381	358	418	356	357	356	356	356
	CitasMed	3,184	1,625	1,624	1,594	1,592	1,618	1,592	1,592	1,592
Total mutants		12,922	7,447	7,013	6,635	6,886	7,449	6,440	6,441	6,461
Total										67,694

TABLE 6
Mutation Scores with First-Order and Second-Order Mutants (These Scores Were Calculated with Different Sets of Mutants)

		Reduced test suites								
Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	81.51	97.87	97.91	97.77	96.7	98.43	96.56	96.47	96.86
	Chess	82.56	98.06	97.47	97.35	96.89	98.87	97.96	97.33	97.21
	Monopoly	80.23	97.3	96.72	96.05	96.34	96.77	96	96	97.12
	Ecal	75.1	94.75	96.64	95.21	95.5	97.47	92.97	95.22	93.82
	CitasMed	80.4	96.49	96.51	95.79	96.48	96.66	95.6	95.79	96.79
Mutation score means		79.96	96.89	97.05	96.43	96.39	97.64	95.82	96.16	96.35
Standard deviation		2.87	1.35	0.61	1.08	0.61	0.99	1.83	0.79	1.42

As seen here, the number of second-order mutants is nearly half of the mutants obtained with first-order mutation, but it is different depending on the combination algorithm used to create the second-order mutants. Equivalent mutants have not been identified. Table 6 shows the mutation scores reached by the original test suites of each application and, therefore, by the mutation adequate reduced test suites (reduced with any reduction algorithm).

The Mutation Score of the first-order mutant execution has scores around 80 percent (excepting the *ECal* application, with 75 percent), while the second-order mutation analysis has scores around 95 percent. This information is important in a real testing process: If equivalent mutants are not counted, a mutation score of 80 percent may be considered as adequate in first-order, while the score should be around 95 percent in second.

After obtaining this information, we executed the experimental procedure described in Section 4.6. The results are presented in three groups, one for each algorithm used to remove the inefficient tests: MAX, MIN, and RDM.

7.1 Results Using the MAX Algorithm

Table 7 shows the number of tests of each mutation adequate test suite using the MAX algorithm for first- and second-order mutation. MAX is the most efficient algorithm (from the point of view of the size of the reduced suite) of the three proposed reduction algorithms: For example, the original test suite of the *Chess* application was reduced from 913 test cases to 303 test cases and preserves the same mutation score for first-order mutants, which is common for the remaining applications. This is a first indicator that second-order is less rigorous than first-order mutation.

Regarding the first-order mutants, Table 8 makes it possible to compare the scores reached: 1) the test suites reduced with the first-order killing matrixes, and 2) the test suites reduced with the second-order killing matrixes. As seen in this table, the second-order reduced test suites obtain lower scores than the reduced test suites for first-order mutants (around 6 percent lower): This means that, in fact, second-order mutation has implications for the possible loss of quality of test suites. The idea is therefore to evaluate whether this loss is acceptable or not.

Note, however, the penultimate row in Table 8: It shows the means of the mutation scores achieved by each reduced test suite. As can be seen, the differences between the means of the second-order scores range from 71.81 to 74.90 percent, quite similar to the 79.96 percent obtained by first-order score means.

In order to determine the existence of a statistical difference between the scores obtained with test suites adequate for first-order mutants and test suites adequate for second-order mutants, we applied the Mann-Whitney test (Section 4.7) to compare the scores shown in Table 8.

The second column in Table 13 shows the results of the test: The signification level is 0.005, which is lower than 0.05; thus, we can reject H0. This indicates that, in this experiment,

TABLE 7
Number of Tests Included in Each Test Suite Reduced with the MAX Algorithm

		Reduced test suites								
Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	69	38	44	38	45	45	40	37	44
	Chess	303	118	134	114	135	120	131	109	325
	Monopoly	55	26	32	25	24	30	27	25	29
	Ecal	42	28	22	24	22	19	25	21	22
	CitasMed	226	105	116	104	118	117	107	104	110
Total number of test cases		695	315	348	305	344	331	330	296	530

TABLE 8
Mutation Scores Achieved by the Tests Obtained with the MAX Algorithm When Executed on First-Order Mutants

		Reduced test suites								
Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	81.51	76.43	78.78	76.58	78.34	77.26	77.31	76.09	78.04
	Chess	82.56	77.27	77.43	74.97	77.29	77.25	77.04	74.24	80.02
	Monopoly	80.23	75.59	75.92	75.12	75.12	76.64	75.76	75.2	75.92
	Ecal	75.1	70.78	67.27	67.41	66.85	66.01	67.55	61.37	67.41
	CitasMed	80.4	71.86	73.46	72.15	73.25	72.93	72.3	72.15	72.8
Mutation score means		79.96	74.39	74.57	73.25	74.19	74.02	73.99	71.81	74.90
Standard deviation		2.87	2.89	4.53	3.64	4.53	4.83	4.12	6.02	4.99

TABLE 9
Number of Tests Included in Each Test Suite Reduced with the MIN Algorithm

		Reduced test suites								
Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	101	92	87	103	91	104	88	91	98
	Chess	460	336	309	307	336	324	335	320	330
	Monopoly	76	76	52	51	50	55	52	59	52
	Ecal	63	53	49	44	50	49	53	42	51
	CitasMed	329	229	231	217	229	237	218	217	242
Total number of test cases		1,029	786	728	722	756	769	746	729	773

TABLE 10
Mutation Scores Achieved by the Tests Obtained with the MIN Algorithm When Executed on First-Order Mutants

		Reduced test suites								
Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	81.51	79.47	79.76	79.81	80.60	80.25	80.06	79.86	80.50
	Chess	82.56	80.53	79.82	79.94	80.15	80.20	80.17	79.94	80.08
	Monopoly	80.23	80.18	78.82	77.70	77.14	78.90	78.66	78.10	78.02
	Ecal	75.1	72.89	70.79	71.91	71.07	70.51	71.35	69.10	72.19
	CitasMed	80.4	76.76	76.82	75.72	76.76	76.89	76.45	75.72	77.23
Mutation score means		79.96	77.97	77.20	77.02	77.15	77.35	77.34	76.54	77.61
Standard deviation		2.87	3.20	3.79	3.34	3.75	4.06	3.67	4.50	3.33

there is empirical evidence to assert that there are statistical differences in the scores obtained with first-order and second-order mutation using the MAX algorithm in Step 7 of the mutation process; this suggests that second-order mutation loses effectiveness.

7.2 Results Using the MIN Algorithm

This section shows the results of the experiment using the MIN algorithm in Step 7 of the mutation process. Table 9 shows the number of test cases of each mutation adequate test suite for first and second-order mutants. As the MIN algorithm is less efficient than the MAX algorithm, the test suites obtained using the MIN algorithm are bigger than the test suites obtained using the MAX algorithm, although they are smaller than the original ones: For the *Chess* application, for example, the original test suite is reduced to 460 test cases (over the 303 test cases obtained with MAX, but under the original 913).

Table 10 shows the mutation scores obtained by the test suites in Table 9 with the first-order mutants. In general, we can see that the scores obtained by the adequate test suites for second-order mutants are lower than the scores obtained by the adequate test suites for first-order mutants, but these differences are very small, around 2.5 percent. This information demonstrates that using the MIN algorithm in the mutation process, second-order mutation becomes more effective than MAX.

TABLE 11
Number of Tests Included in Each Test Suite Reduced with the RDM Algorithm

		Reduced test suites								
Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	92	66	68	61	72	66	71	62	73
	Chess	395	231	236	216	244	224	231	223	327
	Monopoly	64	66	42	40	37	45	41	40	43
	Ecal	52	42	32	36	41	32	39	31	40
	CitasMed	293	200	191	202	193	196	192	171	201
Total number of test cases		890	604	569	559	580	570	571	522	679

TABLE 12
Mutation Scores Achieved by the Tests Obtained with the RDM Algorithm When Executed on First-Order Mutants

		Reduced test suites								
Algorithms	Order	1st	2nd							
	Interclass Restriction		Yes				No			
	Algorithm		BO	EC	FL	R	BO	EC	FL	R
Applications	Cine	81.51	77.66	79.96	78.39	79.47	79.72	79.86	78.54	79.67
	Chess	82.56	79.24	79.16	78.34	79.12	79.21	79.23	78.13	80.06
	Monopoly	80.23	80.18	77.70	76.66	76.58	77.78	77.46	76.82	77.78
	Ecal	75.10	71.49	70.93	71.07	71.49	70.79	70.51	67.42	72.61
	CitasMed	80.40	75.54	76.60	75.51	76.38	75.91	76.01	75.32	75.41
Mutation score means		79.96	77.04	76.76	76.12	76.53	76.65	76.67	75.07	77.30
Standard deviation		2.87	3.59	3.41	3.12	3.30	3.54	3.92	4.56	3.20

TABLE 13
Mann-Whitney U Results for the MAX, MIN, and RDM Algorithms

Statistics	Results for MAX algorithm	Results for MIN algorithm	Results for RDM algorithm
Mann-Whitney U	22	39	32
Wilcoxon W	842	859	852
Z	-2.817	-2.203	-2.456
Asymptotic Sig.	**0.005**	**0.028**	**0.014**
Exact Sig.	0.003	0.026	0.011

The penultimate row in Table 10 shows the means of the mutation scores. The differences between the means of the second-order scores range from 76.54 to 77.97 percent, which are very similar to the 79.96 percent obtained by the first-order score means.

Once more, we applied the Mann-Whitney test with the scores from Table 10, comparing the scores obtained with adequate test suites for first-order and the scores obtained with adequate test suites for second-order mutants. Table 13 (third column) shows the results of the test. Now, the significance level is 0.028, which is lower than 0.05. Therefore, in this case there is empirical evidence to reject our H0 when using the MIN algorithm in Step 7 of the mutation process; this also suggests that the second-order loses effectiveness.

7.3 Results Using the RDM Algorithm

The RDM algorithm is a random algorithm, so the experiments using this algorithm were repeated three times and the final data was calculated as the mean of the three repetitions in order to mitigate results from chance, as the experimental procedure shows (Section 4.6). Thus, all the data shown in this section (number of test cases and scores) is the mean of three values.

The data from the reduced test suites appear in Table 11. The test suite sizes are between those obtained with MAX and with MIN.

The scores achieved by the second-order test suites with the first-order mutants are shown in Table 12. These scores are close to the scores achieved with the MIN algorithm, thus indicating that using the RDM algorithm in Step 7 of the mutation process is more rigorous than using the MAX algorithm. The table shows that, in general, the scores are also lower than the scores achieved with adequate test suites for first-order mutants, but, as with the MIN algorithm, the differences are small, around 3 percent.

The results of applying the Mann-Whitney test to the scores in Table 12 appear in the fourth column in Table 13. The signification level of the tests is now 0.014, which is lower than 0.05; therefore, there is empirical evidence to reject our H0 with the RDM algorithm. This also suggests that second-order mutation loses effectiveness.

TABLE 14
Wilcoxon Test Results Comparing the Combination Algorithm and Restrictions

Statistics	Intraclass-interclass	BO-EC	BO-FL	BO-R	EC-FL	EC-R	FL-R
Z	-0.405	-0.73	-4.206	-0.638	-4.052	-0.792	-3.918
Asymptotic sig.	**0.685**	**0.465**	**0.000**	**0.524**	**0.000**	**0.428**	**0.000**

7.4 Additional Studies: Comparison of the Combinations Algorithms and Restrictions Used in the Experiment

Up to this point, we have analyzed the differences between first-order and second-order mutation and the effect of using the second-order in the scores, using several combinations of elements, with each producing similar but different results. In order to establish good practices to create second-order mutants, this section analyzes and compares the combination algorithms and the restrictions to determine the best way to create second-order mutants.

In the first part of the analysis in this section, we compared the scores obtained both when the *interclass* restriction was and was not used (respectively, columns "Yes" and "No" in Tables 8, 10, and 12). To compare the scores we applied the Wilcoxon signed-rank tests for paired samples (see Section 4.7). The established hypotheses to apply these tests were:

- H0: The two samples have a similar mean. In other words, the scores obtained with and without the restriction are similar.
- H1: The mean of the two samples differs. In other words, the scores obtained with the restriction and without it are different.

Table 14 (second column) shows the results of the test. The significant level is 0.685, which means that there is no empirical evidence to reject the H0, and thus applying the restriction has no effect on the results.

In the second analysis in this section, we compared the scores obtained with each combination algorithm (columns BO, EC, FL, and R in Table 8, 10, and 12). To compare the scores obtained with each algorithm we applied the Wilcoxon signed rank test (see Section 4.7) comparing them two by two. The established hypotheses for each pair were:

- H0: The distributions are the same with the different treatments. In other words, the scores obtained with each combination algorithm are similar.
- H1: The distributions are not the same with the different treatments. In other words, the scores obtained with each combination algorithm are different.

Table 14 (columns 3 to 8) shows the results of the Wilcoxon signed-rank test for the six pairs (BO-EC, BO-FL, BO-R, EC-FL, EC-R, and FL-R). The signification level is always higher than 0.05 except when the FL combination algorithm is compared with another one. This means that only the FL algorithm obtains significantly different results in the scores. For purposes of illustration, Table 15 shows the mean, maximum, and minimum values in the scores: Note that they are always lower with the FL algorithm.

Therefore, we can conclude that the interclass/intraclass restriction has no influence on quality. As for the combination algorithms, the worst option is using FL (*FirstToLast)* to create second-order mutants.

7.5 Risk Analysis of Higher Order Mutation

Even if data from the experiments presented in Section 5 show small differences between first-order and second-order mutation (second-order mutation loses between 2.5 and 6 percent of mutants), the statistical analysis demonstrated significant differences. Therefore, using higher mutation always involves the possibility of losing some errors introduced by the mutation operators, which may imply a lower effectiveness in mutation testing.

However, in some situations, the cost savings of second-order mutation [3] may justify the risk of leaving undiscovered faults. In this section we analyze the probabilistic risk of second-order mutation compared to its cost savings. For this, we have designed a mathematical formula (Fig. 14) based on the probability of estimating the percentage of faults that can remain undiscovered by test cases using higher order mutation.

Let us suppose that we use some mutation operators to create n mutations (or syntactic changes), and with these n mutations we create a set of k-*order* mutants kMs, each with k mutations.

- $ms = \{m_1, m_2, m_3, \ldots, m_n\}$.
- $kMs = \{M_{1\ldots k}, M_{k+1\ldots 2k}, M_{2k+1\ldots 3k}, \ldots, M_{n-(k-1)\ldots n}\}$.

TABLE 15
Descriptive Statistical Values of the Combination Algorithms

Combination alg.	Media	Typical deviation	Min	Max
1st-Order	79.96	2.87	75.1	82.56
BO	76.20	3.70	66.01	80.53
EC	76.10	3.80	67.27	80.17
FL	74.98	4.34	61.37	79.94
R	76.26	3.76	66.90	80.53

$$PK(M_{i...i+k-1}) = \frac{|\bigcup_{j=i}^{j=i+k-1} t(m_j)|}{|ts|}$$

Fig. 9. Probability of killing a mutant.

$$PK_k(M_{i...i+k-1}) = \frac{|\bigcap_{j=i}^{j=i+k-1} t(m_j)|}{|ts|}$$

Fig. 10. Probability of discovering all mutations.

$$PK_{1...k-1}(M_{i...i+k-1}) = \frac{|\bigcup_{j=i}^{j=i+k-1} t(m_j)|}{|ts|} - \frac{|\bigcap_{j=i}^{j=i+k-1} t(m_j)|}{|ts|} = \frac{|\bigcup_{j=i}^{j=i+k-1} t(m_j)| - |\bigcap_{j=i}^{j=i+k-1} t(m_j)|}{|ts|}$$

Fig. 11. Probability of killing a mutant without detecting all the mutations.

We also have a set of t-test cases that kill some of those mutants.

- $ts = \{t_1, t_2, t_3, \ldots, t_t\}$.

Let $t(m_i)$ be the set of test cases that discover the fault in m_i. The probability, $PK(M_{i...i+k})$, of killing the mutant $M_{i...i+k}$ is the probability of selecting a test case that finds any of the faults in $m_i, m_{i+1}, \ldots$ or m_{i+k}. That is:

- $PK(M_{i...i+k-1})$ = Probability of selecting $t \in \{t(m_i)\} \cup \cdots \cup \{t(m_{i+k-1})\}$.

And the probability of selecting $t \in (\mathrm{t}(m_i) \cup \cdots \cup t(m_{\mathrm{i+k-1}}))$ is the number of tests in the set $\mathrm{t}(m_i) \cup \cdots \cup t(m_{\mathrm{i+k-1}})$ divided by the total number of tests.

Fig. 9 determines the probability of killing a mutant, but does not ensure that all the mutations in the mutant will be discovered. To discover all the mutations inside the mutant, we have to select a test case that discovers all the mutations. Fig. 10 shows the probability of discovering all the mutations when a mutant is killed. Therefore, the probability of killing a mutant without detecting all the mutations is $\mathrm{PK}_k(\mathrm{M}_{i...i+k-1} - \mathrm{PK}(\mathrm{M}_{i...i+k-1})$ (Fig. 11).

We now know how to determine the probability of not finding all the mutations in a higher order mutant when it is killed. Thus, the probability of not finding errors in a complete set of k-order mutants is the mean of the probability of each mutant since these probabilities are exclusive. The formula for a complete set of k-order mutants is described in Fig. 12.

$$PK_{1...k-1}(kMs, ts) = \frac{\sum_{j=1}^{j=|kMs|} PK_{1...k-1}(kMs_j)}{|kMs|}$$

Where kMs is composed of mutants as $M_{i...i+k-1}$

Fig. 12. Probability of not detecting all the mutations of a complete set of k-order mutants.

$$PUm(kMs) = \frac{|ms| - |kMs|}{|ms|} * 100$$

Where ms is a set composed of all the mutations included in each mutant of kMs

Fig. 13. Percentage of possible lost mutations.

$$RUm(kMs, ts) = \frac{|ms| - |kMs|}{|ms|} * PK_{1...k-1}(kMs, ts)$$

Where ms is a set composed of all the mutations included in each mutant of kMs

Fig. 14. Risk analysis formula.

The next step is to determine how many mutations are lost based on this probability. The number of lost mutations is the number of total mutations minus the number of higher order mutants. This is due to the fact that at least one mutation must be discovered to kill a mutant. Also, an equivalent mutant can be created from no equivalent first-order mutants or a decoupled high-order mutant can be created (the two faults must be found to kill it). However, the probability of creating them is very low; thus, to simplify, these kinds of mutants were not taken into account. The percent of possible undiscovered mutations is represented by the formula in Fig. 13.

Therefore, the formula that determines the percent of mutations that cannot be detected using high-order mutation is the formula described in Fig. 14.

To apply the formula in Fig. 14 we need the set of k-order mutants to know the mutations contained by each k-order mutant and the first-order killing matrix to know all the $t(m_j)$. The Bacterio tool is able to determine these two elements and calculate the PUm formula for any order.

Fig. 15 shows five tables (one for each application used in this paper). Each table shows the percent of mutations that can be lost with an order from 1 to 10 for each combination algorithm (Section 4.4.1). Also, each graph shows a curve with the cost of higher order mutation calculated as 1/order.

The tables show that when the order increases, fewer mutations will be discovered and the cost savings is lower. The cut point between the cost savings and the percent of undiscovered mutation is between the third- and fifth-order. However, using third-order mutation can sometimes be very risky, as for example with the Monopoly application, where we can lose more than 20 percent of the mutations.

8 Discussion

At first glance, the effectiveness of second-order mutation at system level is quite similar to that of first-order mutation

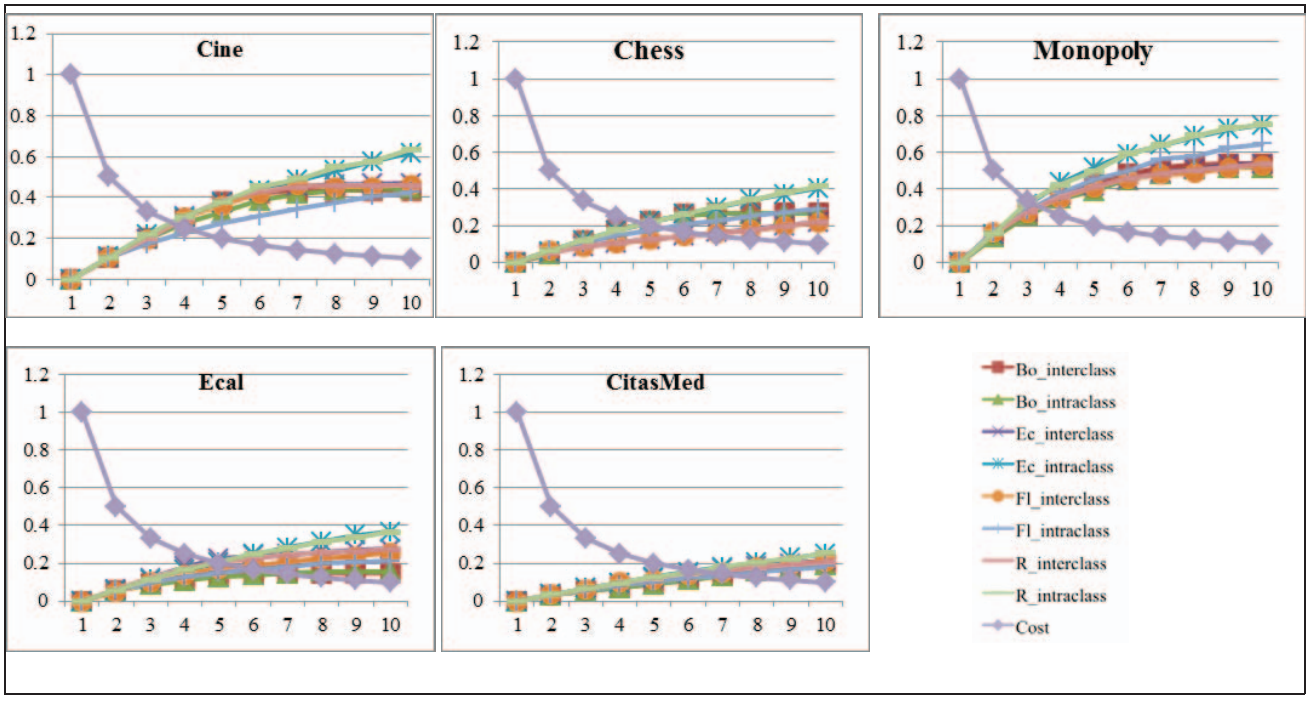


Fig. 15. Risk analysis graphs.

(second-order mutation loses 3.8 percent of mutants). However, even if the difference is small, as shown in Table 13, there is empirical evidence to accept the alternative hypothesis: Second-order mutation has lower quality than first-order mutation. Thus, second-order mutation gives slightly lower mutation scores than first-order mutation, irrespective of the reduction algorithm (MAX, MIN, and RDM) used (although some algorithms lose more effectiveness than others).

Although the results from first order and second order algorithms are not apparently very different (Table 15), there is significant evidence of the loss of effectiveness in the last ones. However, the risk analysis shows that second-order mutation may be adequate when using the algorithms presented in this paper at system level with no critical applications. This is due to the fact that the costs of mutation testing are reduced more than a half [3] (in terms of number of mutants and equivalent mutants) and the effectiveness is only slightly decreased.

The results of applying the probabilistic system to the applications of the experiments show that, up to third-order mutation, it will be worthwhile (in terms of "lost effectiveness/costs reduction") for all the applications, even fourth or fifth-order mutation being worthwhile for some of them (always taking into account that, with higher orders, a greater lost of effectiveness is conveyed).

One important condition which was taken into account during this experiment is the score achieved by the test cases with second-order mutants (around 95 percent), which must be higher than the score achieved by test cases with first-order mutants (around 80 percent). The difference between the scores of first-order and second-order mutation is due to the fact that the number of equivalent mutants decreases using second-order mutation [3], [11], and one of the two errors inserted in each version is uncovered.

This study confirms the results shown in previous studies applied at unit level [3]. Taking into account the high cost savings of second-order mutation as well as the novel test design strategy supported by mutation at system level and Flexible Weak Mutation, second-order mutation is sufficiently effective and requires fewer resources.

After conducting the experiments, we can conclude that second order mutation reduces costs significantly without losing much effectiveness. Thus, whenever system testing is performed for big, noncritical applications, we would recommend using high-order mutation, since the costs are reduced. However, for small or critical applications, we would recommend using first-order mutation since high-order mutation loses some effectiveness.

9 Threats to Validity

This section discusses several issues that could threaten the validity of the experiments and how they have been alleviated.

9.1 Threats to Construct Validity

Construct validity is the "degree to which the independent and the dependent variables are accurately measured by the measurement instruments used in the experiment" [53].

In this experiment, the dependent variable is the quality of the test suite, and it is objectively measured as the mutation score reached in a set of mutants. Independent variables are also deterministic: the number of faults in each mutant, combination algorithm, and combination restriction.

9.2 Threats to Internal Validity

Internal validity is the degree of confidence in a cause-effect relationship between factors of interest and the observed results [53].

Due to the nature of both experiments (automatic and deterministic generation of mutants, execution of test cases, and result analysis), all variables have been controlled and, therefore, threats to internal validity are minimized. However, the quality of the test cases can threaten the internal validity since we have not removed the equivalent mutants, but, given the usual percentages of equivalent mutants, we can consider that working with scores around 74-75 percent is adequate, taking into account that the developers have considered their test suites to be complete enough.

9.3 Threats to External Validity

External validity is the "degree to which the research results can be generalized to the population under study and other research settings" [53]. The greater the external validity, the more the results of an empirical study can be generalized to actual software engineering practice.

Obviously, the nature and the size of the sample influence the generalization of the results, which makes it impossible for us to affirm that our conclusions are completely conclusive.

In order to mitigate this threat, we left all the experimental information and the tool available for download at http://mutationandcombinatorialtesting.blogspot.com.

10 Conclusions and Future Work

This paper has presented a study about second-order mutation at system level. An experiment with five programs was performed in order to compare the quality of second-order and first-order mutation techniques on the basis of the mutation scores achieved.

The result of the experiment shows small differences in the effectiveness of the two techniques (the quality of the test cases designed for second-order is slightly lower in terms of the mutation score). Cost savings from second-order (lower number of mutants and equivalent mutants and fewer executions resulting in a cheaper mutation process) may, though, validate this technique for testing noncritical applications.

In order to increase the validation of the study, we plan to perform more replications of the experiment presented here using other kinds of applications. Additionally, we intend to extend the study to more combination algorithms to create second-order mutants.

Finally, we plan to repeat these experiments with higher orders such as third, fourth, etc., in order to determine up to what order the mutation process is sufficiently safe.

Acknowledgments

This work has been partially financed by the Spanish Ministry of Science and Technology, projects PEGASO, TIN2009-13718, PANGEA, TIN2009-13718-C02-02, and DIMITRI, TRA2009-0131. Pedro Reales has an FPU grant from the Spanish Ministry of Education. The authors would like to thank their colleague Mercedes Fernández Guerrero from the Department of Applied Mathematics at the University of Castilla-La Mancha for her kind support and help with the statistical analysis of the data.

References

[1] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379-1393, 2009.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge Univ. Press, 2008.

[3] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Software Testing, Verification, and Reliability*, vol. 19, no. 2, pp. 111-131, 2008.

[4] A.J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification, and Reliability*, vol. 7, no. 3, pp. 165-192, 1997.

[5] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," *Proc. Third Int'l Conf. Software Testing Verification and Validation*, Apr. 2010.

[6] B.J.M. Grün, D. Schuler, and A. Zeller, "The Impact of Equivalent Mutants," *Proc. IEEE Int'l Conf. Software Testing, Verification, and Validation Workshops*, pp. 192-199, Apr. 2009.

[7] M. Polo and P. Reales, "Mutation Testing Cost Reduction Techniques: A Survey," *IEEE Software*, vol. 27, no. 3, pp. 80-86, May/June 2010.

[8] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," *Proc. Eighth Int'l Working Conf. Source Code Analysis and Manipulation*, pp. 28-29, Sept. 2008.

[9] M. Papadakis, N. Malevris, and M. Kintis, "Mutation Testing Strategies: A Collateral Approach," *Proc. Fifth Int'l Conf. Software and Data Technologies*, pp. 325-328, June 2011.

[10] M. Harman, Y. Jia, and W. Langdon, "A Manifesto for Higher Order Mutation Testing," *Proc. Third Int'l Conf. Software Testing, Verification, and Validation Workshops*, pp. 80-89, 2010.

[11] M. Papadakis and N. Malevris, "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies," *Proc. Software Testing, Verification, and Validation Workshops*, pp. 90-99, Apr. 2010.

[12] P. Reales, M. Polo, and J. Offutt, "Mutation at System and Functional Levels," *Proc. Third Int'l Conf. Software Testing, Verification, and Validation Workshops*, pp. 110-119, Apr. 2010.

[13] T.A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, no. 1, pp. 31-45, Mar. 1982.

[14] D. Baldwin and F.G. Sayward, "Heuristics for Determining Equivalence of Program Mutations," report, 1979.

[15] A. Offutt and J. Pan, "Detecting Equivalent Mutants and the Feasible Path Problem," *Wiley's Software Testing, Verification, and Reliability*, vol. 7, no. 3, pp. 165-192, Sept. 1997.

[16] R. Hierons, M. Harman, and S. Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *Software Testing, Verification, and Reliability*, vol. 9, no. 4, pp. 233-262, Dec. 1999.

[17] E. Mresa and L. Bottaci, "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study," *Software Testing, Verification, and Reliability*, vol. 9, no. 4, pp. 205-232, Dec. 1999.

[18] A.J. Offutt, G. Rothermel, R.H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, pp. 99-118, 1996.

[19] E.F. Barbosa, J.C. Maldonado, and A.M.R. Vincenzi, "Toward the Determination of Sufficient Mutant Operators for C," *Software Testing, Verification, and Reliability*, vol. 11, no. 2, pp. 113-136, 2001.

[20] R.A. DeMillo and E.H. Spafford, "The Mothra Software Testing Environment," *Proc. 11th NASA Software Eng. Laboratory Workshop*, 1986.

[21] A.T. Acree, "On Mutation," thesis, School of Information and Computer Science, Georgia Inst. of Technology, 1980.

[22] K.N. King and A.J. Offutt, "A Fortran Language System for Mutation Based Software Testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685-718, 1991.

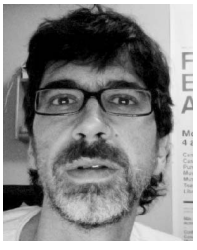
[23] R. Untch, A. Offutt, and M. Harrold, "Mutation Analysis Using Program Schemata," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 139-148, June 1993.

[24] Y.-S. Ma, J. Offutt, and Y.R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification, and Reliability*, vol. 15, no. 2, pp. 97-133, 2005.

[25] M. Grindal, A.J. Offutt, and S.F. Andler, "Combination Testing Strategies: A Survey," *Software Testing, Verification, and Reliability*, vol. 15, pp. 167-199, 2005.

[26] K.-C. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing," *IEEE Trans. Software Eng.*, vol. 28, no. 1, pp. 109-111, Jan. 2002.
[27] A.J. Offutt, "Investigations of the Software Testing Coupling Effect," *ACM Trans. Software Eng. and Methodology*, vol. 1, no. 1, pp. 15-20, 1992.
[28] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," *Proc. Third Testing: Academic and Industrial Conf. Practice and Research Techniques*, pp. 94-98, Aug. 2008.
[29] M. Polo, M. Piattini, and S. Tendero, "Integrating Techniques and Tools for Testing Automation," *Software Testing, Verification, and Reliability*, vol. 17, no. 1, pp. 3-39, 2007.
[30] H. Do, S. Elbaum, and G. Rothermel, "Infrastructure Support for Controlled Experimentation with Software Testing and Regression Testing Techniques," *Proc. Int'l Symp. Empirical Software Eng.*, pp. 60-70, Aug. 2004.
[31] M.E. Delamaro, J.C. Maldonado, and A. Vincenzi, "Proteum/IM 2.0: An Integrated Mutation Testing Environment," *Proc. First Workshop Mutation Analysis*, pp. 91-101, Oct. 2000.
[32] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating Mutation Testing Alternatives: A Collateral Experiment," *Proc. Asia Pacific Eng. Conf.*, Nov./Dec. 2010.
[33] H. Agrawal, "Dominators, Super Blocks, and Program Coverage," *Proc. 21st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pp. 25-34, Jan. 1994.
[34] W. Langdon, M. Harman, and Y. Jia, "Efficient Multi-Objective Higher Order Mutation Testing with Genetic Programming," *J. Systems and Software*, vol. 83, no. 12, pp. 2416-2430, 2010.
[35] W. Langdon, M. Harman, and Y. Jia, "Multi Objective Higher Order Mutation Testing with GP," *Proc. 11th Ann. Genetic and Evolutionary Computation Conf.*, pp. 1945-1946, 2009.
[36] W. Langdon, M. Harman, and Y. Jia, "Multi Objective Higher Order Mutation Testing with Genetic Programming," *Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques*, pp. 21-29, 2009.
[37] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.*, 1994.
[38] R. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34-41, April 1978.
[39] G. Kaminski, U. Praphamontripong, P. Ammann, and J. Offutt, "A Logic Mutation Approach to Selective Mutation for Programs and Queries," *Information and Software Technology*, vol. 53, pp. 1137-1152, 2011.
[40] G. Kaminski, P. Amman, and A.J. Offut, "Better Predicate Testing," *Proc. Sixth Int'l Workshop Automation of Software Test*, pp. 57-63, 2011.
[41] G. Kaminski and P. Amman, "Using Logic Criterion Feasibility to Reduce Test Set Size While Guaranteeing Fault Detection," *Proc. Int'l Conf. Software Testing*, Apr. 2009.
[42] R. Just, G.M. Kapfhammer, and F. Schweiggert, "Using Conditional Mutation to Increase the Efficiency of Mutation Analysis," *Proc. Sixth Int'l Workshop Automation of Software Test*, pp. 50-56, 2010, 2011.
[43] R. Purushothaman and D.E. Perry, "Toward Understanding the Rhetoric of Small Source Code Changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6 pp. 511-526, June 2005.
[44] V.R. Basili and H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Softwareenvironments," *IEEE Trans. Software Eng.*, vol. 14, no. 6, pp. 758-773, June 1988.
[45] V.R. Basili, F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Trans. Software Eng.*, vol. 25, no. 4, pp. 456-473, July/Aug. 1999.
[46] M. Harrold, R. Gupta, and M. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology*, vol. 2, no. 3, pp. 270-285, 1993.
[47] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," *Proc. Int'l Conf. Software Maintenance*, pp. 549-558, 2005.
[48] S. Tallam and N. Gupta, "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," *Proc. Sixth ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.*, pp. 35-42, 2005.
[49] "Eclipse Framework," http://www.eclipse.org/, Feb. 2011.
[50] L. Walton, "Eclipse Metrics Plugin," http://eclipse-metrics.sourceforge.net/, Feb. 2011.
[51] "SPSS Statistics," http://www.spss.com/, Feb. 2011.
[52] S. Ghosh and A.P. Mathur, "Interface Mutation," *Software Testing, Verification, and Reliability*, no. 11, pp. 227-247, 2001.
[53] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic, 2000.

Pedro Reales Mateo received the MSc degree from the University of Castilla-La Mancha, where he is currently working toward the PhD degree in computer science. His research areas include the automation of software processes, especially model-driven architecture, software product lines and testing, and mutation testing. Currently, his research work focuses on design and validation techniques to improve the mutation testing process.



Macario Polo Usaola received the PhD degree in computer science from the University of Castilla-La Mancha and the MSc degree in computer science from the University of Seville. He is an associate professor of software engineering and information systems testing in the Department of Information Systems and Technologies at the University of Castilla-La Mancha, Spain. In addition to his research on software engineering and software testing, he is also the author of several novels.

José Luis Fernández Alemán received the BSc (Hons) and PhD degrees, both in computer science, from the University of Murcia in 1994 and 2002, respectively. Currently, his main research interests include mutation testing, computer-based learning, and its application to the fields of computer science and nursing. He has contributed to many Spanish-funded research projects whose topics were related to software engineering. Currently, he is an associate professor of programming and software quality in the Department of Computer Science and Systems at the University of Murcia, Spain. He has published several papers in the areas of e-learning and software engineering. Publications include journal articles in *Software and System Modeling*, the *IEEE Transactions on Education*, *Nursing Education Today* and the *Transactions on Edutainment*.