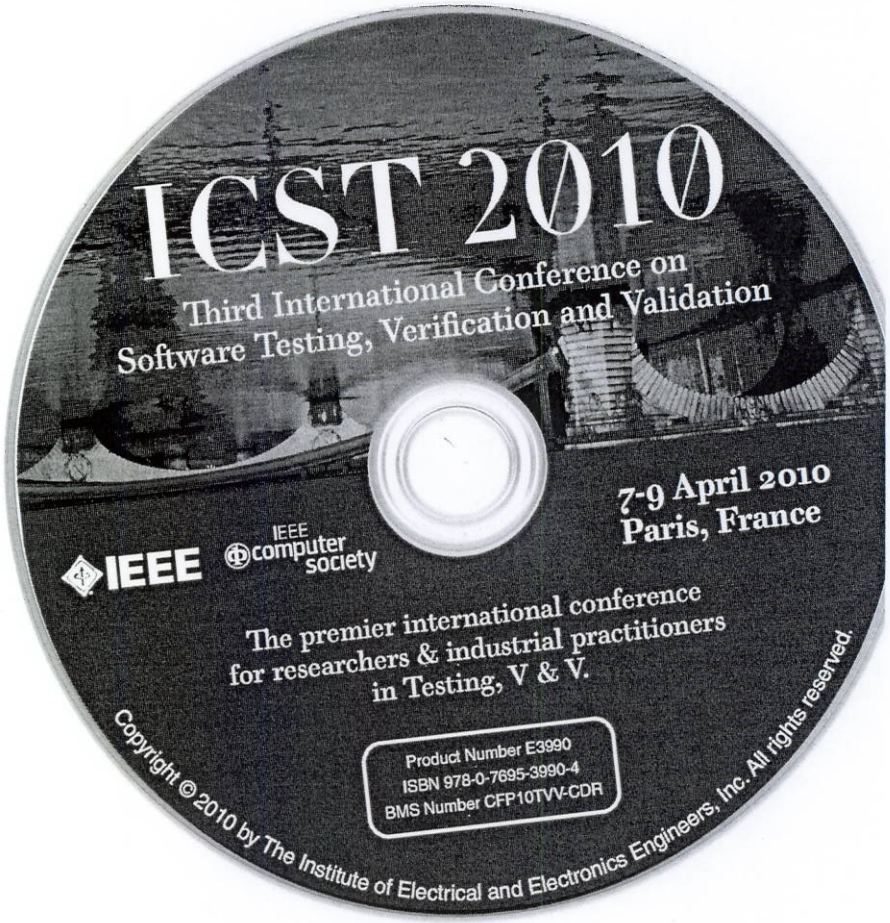
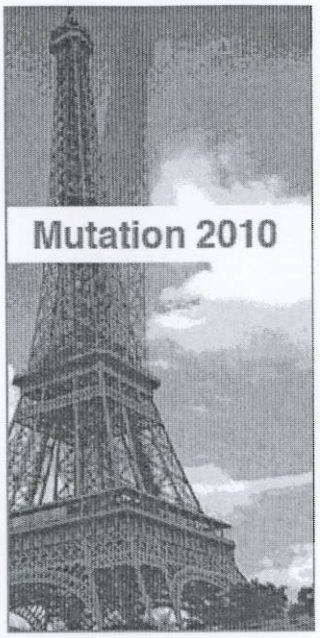

ICST 2010
Third International Conference on
Software Testing, Verification and Validation
7-9 April 2010
Paris, France
IEEE
IEEE computer society
The premier international conference
for researchers & industrial practitioners
in Testing, V & V.
Product Number E3990
ISBN 978-0-7695-3990-4
BMS Number CFP10TVV-CDR
Copyright © 2010 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.




# Mutation 2010: The 5th International Workshop on Mutation Analysis

April 6 2010, Paris, France (associated with ICST 2010)

### About Mutation 2010

Mutation is acknowledged as an important way to assess the fault-finding effectiveness of tests sets. Mutation testing has mostly been applied at the source code level, but more recently, related ideas have also been used to test artifacts described in a considerable variety of notations and at different levels of abstraction. Mutation ideas are used with requirements, formal specifications, architectural design notations, informal descriptions (e.g. use cases) and hardware. Mutation is now established as a major concept in software and systems V&V and uses of mutation are increasing. The goal of the Mutation workshop is to provide a forum for researchers and practitioners to discuss new and emerging trends in mutation analysis. We invite submissions of both full-length and short-length.

### Keynote Speaker

Mark Harman, King's College, UK - "How HOM Helps Mutation Testing".

### Topics of interest

- Mutation-based test adequacy criteria (theory or practical application).
- Mutation testing using higher order mutants.
- Test-case generation using mutants.
- Using mutation in empirical studies (e.g. studies that compare mutation with other testing techniques).
- Industrial experience with mutation.
- New mutation systems for programming languages (e.g. for languages not yet addressed, or offering improvements on existing ones) and for higher-level descriptive notations (e.g. formal specification notations and architectural design notations).
- Novel applications of mutation including mutation for QoS properties (security, performance, etc.).

### Submissions and Publication

Two types of papers can be submitted to the workshop:

- *Full papers (10 pages):* Research, case studies.
- *Short papers (6 pages):* Research in progress, tools, experience reports, problem descriptions, new ideas.

Each submitted paper must conform to the IEEE format and submission guidelines. Submissions will be evaluated according to the relevance and originality of the work and to their ability to generate discussions between the participants of the workshop. Each paper will be reviewed by three reviewers and accepted papers will be published in the IEEE Digital Library.

### SCP Special Issue

Authors of selected papers will be invited to submit extended versions of their papers to a special issue of the journal Science of Computer Programming (SCP).

### Important dates

| | |
|---|---|
| Submission of full papers: | ***January 15, 2010*** |
| Notification of acceptance: | *March 2, 2010* |
| Camera-ready: | *March 26, 2010* |
| Date of workshop: | *April 6, 2010* |

### Website

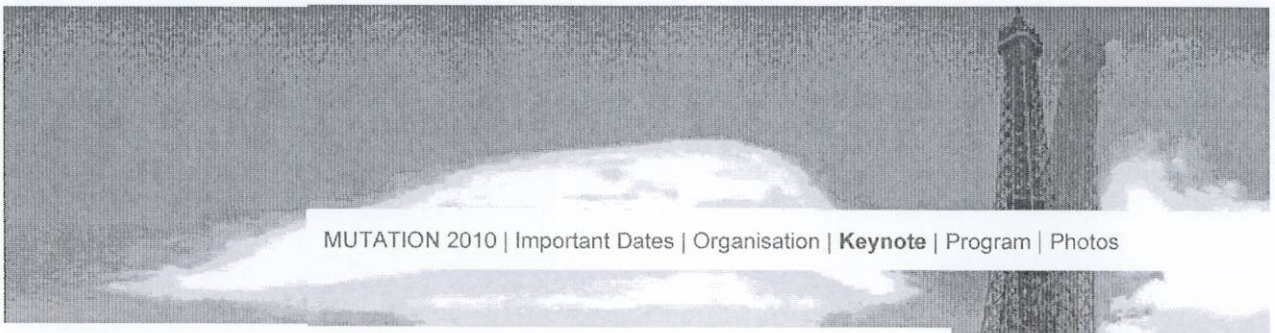
For more information see http://www.st.cs.uni-saarland.de/mutation2010

### Organizers

Lydie du Bousquet, LIG, France
Jeremy S. Bradbury, *UOIT, Canada*
Gordon Fraser, Saarland University, Germany

### Program Committee

Roger Alexander, *Washington State University, USA*
Paul Ammann, *George Mason University, USA*
Benoit Baudry, *INRIA, France*
Leonardo Bottaci, *University of Hull, UK*
Byoungju Choi, *EWHA Womans University, South Korea*
John A Clark, *University of York, UK*
James R Cordy, *Queen's University, Canada*
Rich DeMillo, *Georgia Tech, USA*
Mark Harman, *King's College, UK*
Mark Hampton
Rob Hierons, *Brunel University, UK*
Bill Howden, *University of California at San Diego, USA*
Jose Carlos Maldonado, *Universidade de Sao Paolo, Brasil*
Mercedes Merayo, *Universidad Complutense de Madrid, Spain*
Phil McMinn, *University of Sheffield, UK*
Akbar Siami Namin, *Texas Tech University, USA*
Jeff Offutt, *George Mason University, USA*
Macario Polo, *University of Castilla-La Mancha, Spain*
David Schuler, *Saarland University, Germany*
Yves Le Traon, *University of Luxembourg, Luxembourg*
Laurie Williams, *North Carolina State University, USA*
Eric Wong, *University of Texas at Dallas, USA*
Lu Zhang, *Peking University, China*





## Keynote speaker: Mark Harman (King's College, London)

### How HOM Helps Mutation Testing

Higher Order Mutation testing (HOM testing) is simply Mutation testing in which simply faults are combined to create Higher Order Mutants. This keynote will briefly cover the history of Mutation Testing, considering why it has been traditionally a first order paradigm and giving motivations for a move to the higher order paradigm. The talk will show how Search Based Software Engineering (SBSE) can be used to seek out, from the impossibly large space of potential higher order mutants, those which may possess important and valuable properties that make them good at revealing faults that may otherwise go unnoticed. This keynote is based on joint work with Yue Jia and William B. Langdon in the CREST Centre, King's College London.

Mark Harman is professor of Software Engineering in the Department of Computer Science at King's College London. He is widely known for work on source code analysis and testing and he was instrumental in the founding of the field of Search Based Software Engineering, a field that currently has active researchers in 24 countries and for which he has given 14 keynote invited talks. Professor Harman is the author of over 150 refereed publications, on the editorial board of 7 international journals and has served on 90 programme committees. He is director of the CREST centre at King's College London. More details are available from the CREST website. Together with Yue Jia (also from the CREST Centre), he is responsible for a resource repository and recent analytical survey of the field of Mutation Testing.

NEWS

Photos and presentations available [10.04.2010]

Workshop program available [30.03.2010]

Accepted papers published [05.03.2010]

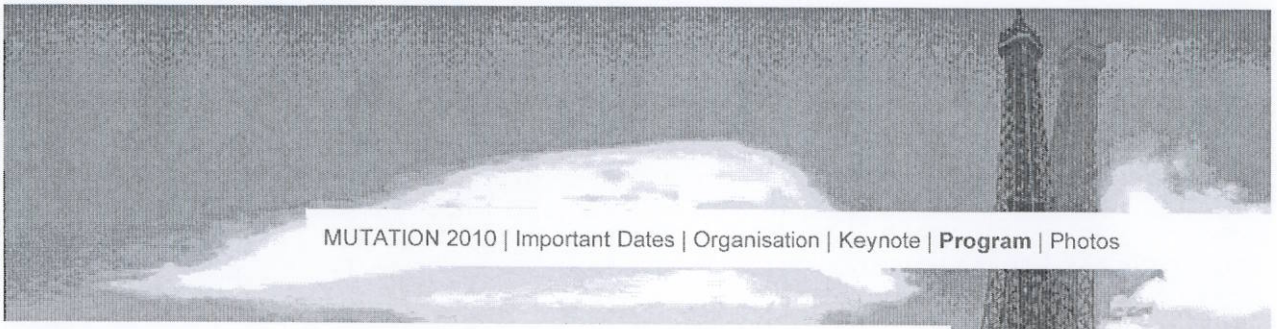
Deadline extended to 29.01.2010 [14.01.2010]

Keynote speaker announced [11.11.2009]

Special issue [9.11.2009]

PC announced [5.11.2009]

Mutation 2010 Workshop website is online [20.7.2009]





## Program

09:00 -- Opening

09:05 -- Keynote

- Mark Harman: How HOM Helps Mutation Testing

10:05 -- Paper Session 1

- Mike Papadakis and Nicos Malevris. An Empirical Evalution of the First and Second Order Mutation Testing Strategies

10:30 -- Break

11:00 -- Paper Session 2

- John Clark, Haitao Dan and Rob Hierons. Semantic Mutation Testing
- Pedro Reales Mateo, Macario Polo Usaola and Jeff Offutt. Mutation at System and Functional Levels
- Mark Trakhtenbrot. Implementation-Oriented Mutation Testing of Statechart Models
- Leonardo Bottaci. Type Sensitive Application of Mutation Operators for Dynamically Typed Programs

12:30 -- Lunch

14:30 -- Discussion Session: "Open Challenges in Mutation Testing" (includes survey results)

- Panelists:
  - Mark Hampton, Former CTO of Certess
  - Mark Harman, King's College, UK
  - Jeff Offutt, George Mason University, USA

16:00 -- Break

16:30 -- Paper Session 3

- Upsorn Praphamontripong and Jeff Offutt. Applying Mutation Testing to Web Applications
- Antonia Estero-Botaro, Francisco Palomo-Lozano and Inmaculada Medina-Bulo. Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions
- Salem Adra and Phil McMinn. Mutation Operators for Agent-Based Models
- Vilas Jagannath, Milos Gligoric, Steven Lauterburg, Darko Marinov and Gul Agha. Mutation Operators for Actor Systems

18:00 -- Closing

NEWS

Photos and presentations available [10.04.2010]

Workshop program available [30.03.2010]

Accepted papers published [05.03.2010]

Deadline extended to 29.01.2010 [14.01.2010]

Keynote speaker announced [11.11.2009]

Special issue [9.11.2009]

PC announced [5.11.2009]

Mutation 2010 Workshop website is online [20.7.2009]

# Mutation at System and Functional Levels

Pedro Reales Mateo, Macario Polo Usaola
Dept. Tecnologías y sistemas de la información
University of Castilla-La Mancha
Ciudad Real, Spain
{pedro.reales, macario.polo}@uclm.es

Jeff Offutt
Software Engineering
George Mason University
Fairfax, VA 22030, USA
offutt@gmu.edu

***Abstract*—Mutation analysis has been applied to many testing problems, including functional programs in numerous languages, specifications, network protocols, web services, and security policies. Program mutation, where mutation analysis is applied to programs, has been applied to the unit level (functions and methods), integration of pairs of functions, and individual classes. However, program mutation has not been applied to the problem of testing multiple classes or entire software programs, that is, there is no system level mutation. This paper introduces a project on the problem of multi-class and system level mutation testing. The technical differences between using mutation to test single classes and multiple classes are explored, and new system level mutation operators are defined. A new execution style for detecting killed mutants, Flexible Weak Mutation, is introduced. A support tool, Bacterio, is currently under construction.**

***Keywords- Mutation; Testing; System Testing; Flexible Weak Muation; Mutation Process***

## I. INTRODUCTION

Generally speaking, mutation analysis uses well defined rules (called *mutation operators*) that are defined on syntactic structures (such as grammars or program source) to make systematic changes to the syntax or to the objects developed from the syntax [1, 2]. When applied to programs, mutation analysis makes systematic changes to the program, then asks the tester to design inputs that cause the mutated program (*mutant*) to create output that is different from the original version of the program. We customarily think of these mutants as being faults, although it is possible that the original program was faulty and the mutant is correct, or the mutant has no affect on the functional behavior of the program (*equivalent*). Most mutation systems introduce one change at a time (changing one terminal symbol in the grammar), although it is possible to make multiple changes (called *higher-order mutants*). This paper is focused on program mutation and assumes all mutants are single-order.

Mutation has previously been applied to programs at three levels [3]. (1) The original application of mutation was to individual functions or methods (*unit level*) [4]. Unit level mutation modifies individual program functions or methods. (2) *Integration level* mutation targets communication between two functions [5-7]. (3) *Class level* mutation modifies multiple functions in a single class. Mutation analysis has been applied to many programming languages and to various other situations.

The significant innovation of this research is to take a step higher in abstraction and apply mutation to *multi-classes* and *complete programs*. The primary motivation for this innovation is to design test inputs that explicitly target faults in how classes interoperate in OO software and to target system-level faults, that is, faults that are not present at lower levels of abstraction. An additional motivation is that we hope these ideas can eventually lead to another path for industry adoption of this very effective criteria-based testing approach by allowing mutation to be used during system-level testing.

Applying mutation analysis to the multi-class and system levels requires the solution of several new technical problems. After a discussion of previous mutation analysis strategies in Section II, Section III discusses some of the novel problems that need to be solved for system level mutation. Most obviously, we are targeting new faults and need to design new mutation operators. Previous mutation systems have used three general types of operators: (1) operators that imitate faults that programmers make, such as replacing one scalar variable with another; (2) operators that force good tests, such as failing only if an expression has the value zero; and (3) operators that imitate uncommon faults, such as changes to a logic predicate that can only be killed by very powerful tests. Section IV presents a preliminary set of new mutation operators for system level testing.

A more subtle problem is that we have to re-define what we mean by "program output." In Mothra [8, 9], which implemented unit-level mutation, program output was simple: return values of the functions. Weak mutation [10-14], which has mostly been discussed in the context of unit-level mutation, considered intermediate state, focusing on the program counter and the variables that can be changed by the mutation. This is important because we must depend on weak mutation in system-level mutation to control cost. In muJava [5], which implements class-level mutation, program outputs are tester-defined strings that return the state of the class.

If we think of program output more generally, we must consider what part of the output (in strong mutation) or program state (in weak mutation) do we check? At the system level with strong mutation, outputs can be values that are explicitly printed, values stored in files and databases, and signals sent to external devices. Some heavily interactive

programs accept inputs, perform some computation, then wait for additional computations (for example GUI programs like Powerpoint or game programs). With these *input-driven* systems, the intermediate state between inputs should be considered to be output. In multi-class and system-level mutation, state is distributed among multiple objects, not always easily accessible, and much of the state is not relevant. Section V discusses a novel approach to compare state called flexible weak mutation. Finally, Section VI presents a tool for multi-class and system-level mutation, which is still under development.

## II. RELATED WORK

### A. Mutation process

Mutation testing involves three steps:

(1) Mutant generation. Mutation operators are applied to the original program to get a set of mutants. A mutation operator is a rule that introduces a syntactic change in the original program.
(2) Mutant execution. Tests are executed against the original program and all the mutants.
(3) Result analysis. Results of the executions are analyzed and the mutation score is calculated (measuring of the quality of the test suite).

The result analysis step counts the number of killed mutants, which are mutants that have different behavior from the original program for at least one test. A test case must fulfil three conditions to kill a mutant (R.I.P. model) [2]:

$$MS(P,T) = \frac{K}{(M - E)}$$

…where:

***P***: Program under test
***T***: Test suite
***M***: Number of mutants
***K***: Number of killed mutants
***E***: Number of equivalent mutants

Figure 1. Mutation score

*1) Reachability*. The mutated statement of the program must be executed by the test case.

*2) Infection*. The execution of the mutated statement must put the mutant into an erroneous state.

*3) Propagation.* The erroneus state must propagate until it reaches the external environment.

An early mutation testing process was defined by Offutt [15]. Figure 2 shows a modification to the original process [16]: the inputs to this process are the program under test and the test suite. First, the test cases are executed. If the system has faults, the program must be fixed and the tests re-executed. When the test cases do not find faults, a set of mutants is created. Then, the test cases are executed with each mutant and the mutation score is calculated after removing the equivalent mutants. Additionally, ineffective test cases can be removed from the suite. If the mutation score reaches a predefined threshold, the process finishes; otherwise, new test cases must be added and the process should start again.

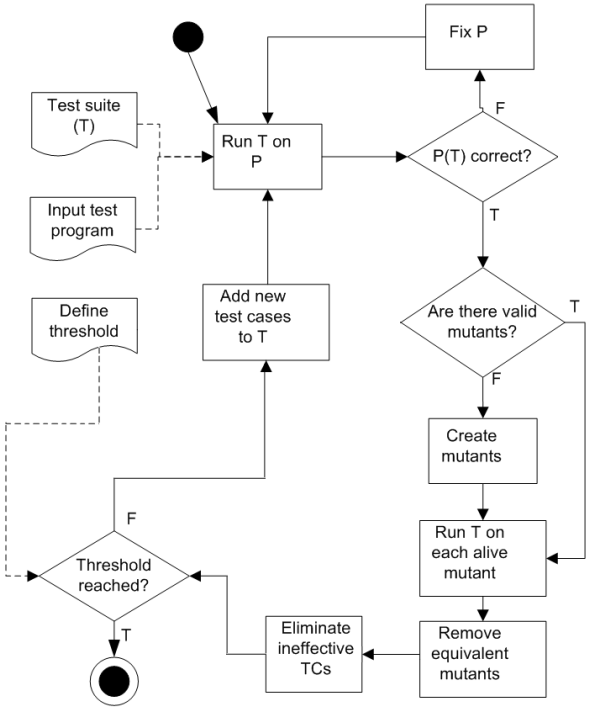



Figure 2. Muation testing process

### B. Test case execution techniques

In strong mutation [4], each test case is executed against the original program and against the mutant: if the mutant exhibits output different from the output of the original program, then it is said that the test case has *killed* the mutant. Thus, the difference is observed at the end of the execution.

To reduce execution cost, Howden [10] proposed *weak mutation*, which only requires the first two conditions to kill a mutant (reachability and infection). In weak mutation, the states of both programs are compared at a predetermined point after the execution of the mutated instruction. If the states are different at that point, the mutant is killed. Howden's paper did not specify exactly where the state comparison should be done. Several studies [11-13] found weak mutation to be less costly and almost as effective as strong mutation.

Woodward and Halewood [14] proposed *firm mutation*, which tries to obtain the advantages of weak mutation without its disadvantages. The state of the mutant is analyzed at a predetermined point between the execution of the mutated part and the end of the execution. It has the advantages of reducing the computational cost with respect to strong mutation, and an improvement in effectiveness with respect to weak mutation. Offutt and Lee's study [13] compared the checking of states at four different locations: after the mutated intermediate code instruction, after the source line statement, after the mutated basic block, and after

the basic block each time it is iterated in a loop. Thus this could be considered to be firm mutation.

### C. Mutation tools

Several mutation tools have been built. Mothra [9] is an early mutation tool and mutates Fortran functions. Mothra generates mutants, executes test cases defined by the tester or generated automatically, and performs results analysis to calculate the mutation score.

Proteum [17, 18] performs unit-level mutation analysis for C programs. A tester selects a C function, then uses Proteum to generate mutants, execute test cases and calculate the mutation score.

Mutation has also been applied in at the class level. Java tools include Jumble [19], Javalanche [20] and muJava [5]. In addition to the "traditional" mutation operators, muJava includes a set of "class-level" operators, which test object-oriented characteristics such as inheritance or polymorphism. This tool has been used in many research studies.

The first tool to bring mutation to the system level is Certitude [21]. Certitude applies mutation analysis to systems on chips for hardware verification. The mutation operators are applied to C++ code and communication protocols of TML SystemC-based designs at the integration level.

### D. Mutation at the integration level

Delamaro et al. [6] presented an approach for integration testing through mutation. The authors analyzed integration faults and designed new mutation to affect connections between units.

Proteum [18] performs mutation at the integration level. This tools implements the operators defined by Delamaro, Maldonado and Mathur [6] for C functions.

Another integration mutation tool is muJava. With muJava, a tester can test specific integration relationships in the class hierarchy of a system through the class-level mutation operators.

Ghosh and Mathur [7] proposed a technique to test components using mutation. This work describes mutation operators to modify interfaces between components.

These techniques do not target true system-level faults. The next section describes a new kind of **system-level mutation**.

## III. Mutation at the Multi-Class and System Levels

Following the ideas of Delamaro et al., this work defines a new kind of mutation analysis that can be used during system; mutation at the system level. Figure 3 illustrates differences among mutation at the different testing levels discussed in this paper.

*Mutation at multi-class and system levels* considers a whole system with its complete set of functionalities, instead of individual units and classes. This allows testers to test a complete system.

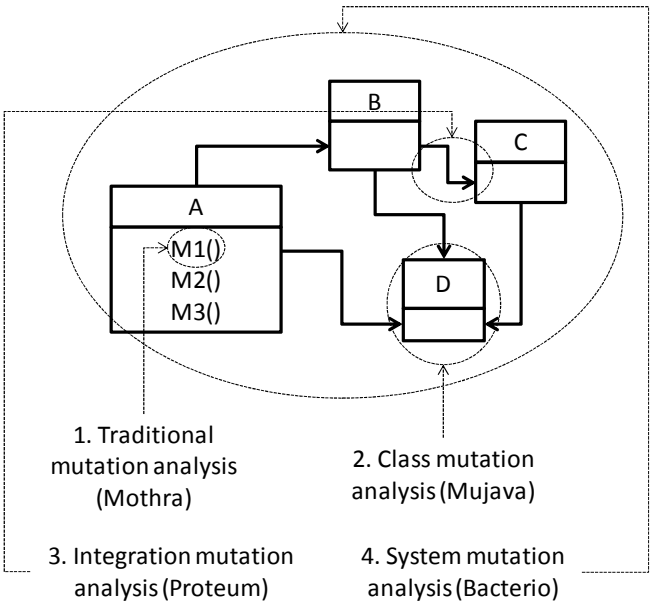



Figure 3. Kinds of mutation analysis

Consider the four class system in Figure 3. In unit level mutation (1), the mutations are introduced into units and can be found with unit test cases. Mutation analysis must be done for each unit to check the quality of the test suite. For example, the tester creates mutants for "*M1()*", then designs test cases and tests for "*M1()*" until an acceptable mutation score is reached. This process must be repeated for each method ("*M2()*", "*M3()*", etc). In class level mutation (2), complete classes must be mutated and tested. Moreover, if a method interacts with another, the tester will have to use a stub if the second method has not been tested or implemented.

At the integration testing level (3), the connections between elements in the system must be tested. In the example of Figure 3, the tester must test the connections among classes through integration test cases. Integration mutation analysis [6, 7, 22] can be used. Each connection is mutated and tested separately.

At the system level (4), ideally all units and all connections among units have been tested separately. So, the overall functionalities supported by the system must be tested. The tester has to generate mutants for the whole system and design test cases to test the system functionality. Then, the tester only has to perform mutation once for the whole system.

This new type of mutation has an important benefit: since a system is normally composed of multiple elements that interact with each other, and since traditional mutation only takes into account separated units, this new type of mutation can check interactions between all the elements of the complete application. Thus, a mutation-adequate test set will complement tests from lower testing levels (unit testing) to higher testing levels (system testing), as illustrated in the V-model in Figure 4.

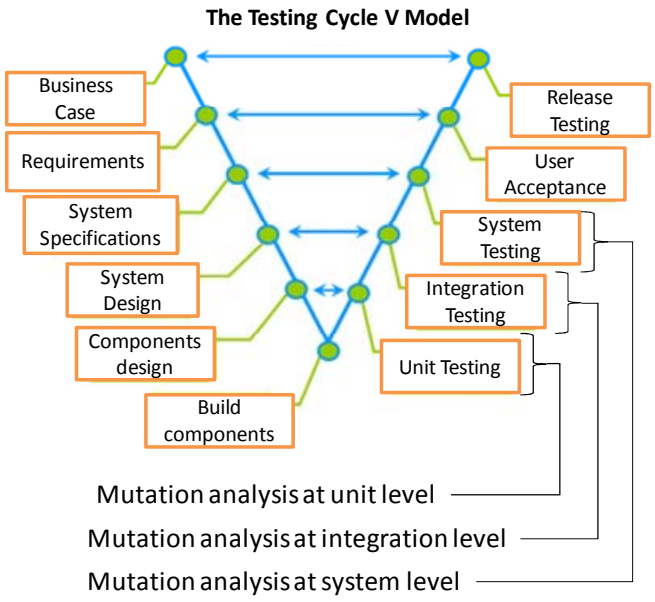



Figure 4. The Testing V-model and Mutation Analysis

Mutation at the system and functional levels also allows testers to check features that are difficult to test with other mutation approaches. For example, it is possible to check whether the sequences of relations between units execute the required functionalities. Additionally, it allows the testing of the functionalities to be supported by a complete system, as well as by its former subsystems.

## IV. Implications of Mutation at the Multi-Class and System Levels

Running a complete system requires more execution than running a single method, thus it is important to use weak mutation. Performing mutation at the system level has several differences, some of which are specific to weak mutation. (A) **What** to compare, that is, which portions of the **state**, is quite different. (B) Execution of the mutants is more complicated. (C) New mutation operators must be defined to modify system level behavior. (D) Detecting equivalent mutants is different. These differences are discussed in the following subsections.

### A. The state of a mutant

A crucial task in mutation is to compare each mutant to the original system in each execution. If we say that when the mutated state differs from the original state, the state is *anomalous*, then a mutant is marked killed when its state becomes anomalous.

In unit level and class level mutation, state comparison for weak mutation is straightforward, since it only requires comparison of the local variables in the unit under test. Under strong mutation, the problem is to identify and compare the outputs of the function. In muJava [5], for example, the problem is solved by the tester who writes each test to return a string that represents the state of the class under test.

In integration mutation [6], the state of two units is compared, instead of one (the caller and the called state). In Proteum [18], the comparisons are done with result values and outputs printed on the screen.

The notion of state in multi-class and system level mutation is more complicated. State is distributed in multiple objects, global and other static variables, and local variables. This makes it particularly difficult to do the obvious, that is, compare the entire state of the program.

As mentioned in Section I, input-driven systems can change our notion of what an output is. If a system test is **not** designed to take the program through to an exit, then the intermediate state when the software waits for the next input has to be compared under strong mutation.

The notion of *masking* is also different. An erroneous state is *masked* during execution if subsequent state becomes correct. In unit, integration, and class level mutation, masking can happen in two ways. First, a variable with an incorrect value can get a new (correct) value before the incorrect value infects another part of the state. Second, a variable with an incorrect value can go out of scope, losing its value. In multi-class and system level mutation, a third possibility is that an object with an incorrect value is destroyed (before the incorrect value can infect another part of state).

Deciding *what* to compare and *when* to compare are crucial decisions for this level of mutation. To solve these problems, we propose a new method of execution, **flexible weak mutation**, as described in Section V.

### B. Problems managing an entire system

Another consequence of system level mutation is that a mutation tool must consider the complete system, with multiple classes and elements. Thus, a mutated version of the original system must be loaded for the execution of each test case, which is more complicated than replacing a single unit.

Two possible ways to solve this problem are:

*1) Replace the whole system*. In this approach, many copies of the SUT as mutants are generated. This approach is simple to implement but has the disadvantage of requiring a lot of space. Suppose that we have a system of 5 megabytes and we generate 1000 mutants. We need 5 gigabytes to store all mutants.

*2) Replacing part of the system.* This approach can be more complex than the previus approach but solves the space problem. In this approach, only the mutated parts are replaced. A disadvantage is that the mutation system must keep track of which portions of the system are changed for each mutant, and control the execution environment environment enough to swap parts of the system in and out during execution.

Additionally, we have to decide whether to replace the source code or executable. Source code replacement is more expensive because of compilation. Unit level mutation can accelerate mutant execution with mutant schemata [23] or compiler-integration [24]. Another technique is bytecode transformation [5].

Also, managing the executions of test cases may involve more difficulties. In system mutation, mutation tools must execute tests for the complete system. These tests must simulate the interactions from actors (people or other systems), as well as the messages that the SUT sends to or receives from the external environment. Taking into account that these tests must be reproducible, a way for capturing and replaying, or for defining user interface events, is also required. Additional complications include database connections, distributed system interactions, setting up of a web server, etc.

### C. *Mutation operators*

Another consequence of mutation at the multi-class and system levels is that new mutation operators are needed.

Mutation operators are rules or functions to make syntactic changes. For example, the Arithmetic Operator Replacement (AOR) mutation operator for Java [25] replaces each arithmetic operator by other applicable operators. Method level operators have been designed for Java [5, 25], Fortran [9], Ada [26], C [27], C# [28] and SQL [29].

Most faults at the multi-class and system level are related to the interactions between units or components and the configuration of the elements that compose the system. System testing focuses on the top level functionalities, whereas integration testing focuses on connections among software components. Also, at the system level, the faults can be related to user interfaces and user interactions.

Table 1, 2 and 3 describe system level mutation operators. Space prohibits the complete definitions.

### D. *Equivalent mutants*

A mutant is equivalent if the change can never result in anomalous behavior. Detecting equivalent mutants can be expensive. This is usually done by hand, since determining whether a mutant is equivalent is undecidable, although sometimes heuristics can be used [30].

Researchers have developed partial techniques to reduce the number of equivalent mutants. For example, program slicing [31] helps human analysts find equivalent mutants. Also, Grün et al. [32] analyzed the *impact* of mutants and determined that a mutant with high impact is less likely to be equivalent.

Polo et al. [33] applied high-order mutation, a technique which combines *n* mutants (each with a single fault) into a new mutant containing all mutants. Since about 20% of first-order mutants can be equivalent, the probability of a high-order mutant being equivalent goes down dramatically. Polo et al.'s paper includes a brief discussion about how different muJava operators produce different numbers of equivalent mutants. Thus, high-order mutation could be combined with other techniques (selective mutation, for example [34]) to get fewer equivalent mutants.

These techniques can be used for multi-class and system level mutation. Program slicing would need to be adapted to the system level. Also, new experiments related to impact analysis are needed for complete systems. Regarding *n-order* mutation, each mutant version may be seeded with more than one fault.

TABLE 1. MUTATION OPERATORS RELATED TO CONFIGURATION

| | |
|---|---|
| TFLD: Text File Line Deletion | The TFLD operator removes a line of a text in a configuration file. |
| TFLI: Text File Line Insertion | The TFLI operator inserts an empty line into a configuration file. |
| TFALD: Text File All Lines Deletion | The TFALD operator removes all lines from configuration file. |
| CNVI: Component new version interchange | The CNVI operator interchanges a component of the system with a new version of the same component. |
| COVI: Component old version interchange | The COVI operator interchanges a component of the system with an old version of the same component. |
| CPVI: Configuration Parameter Values Interchange | The CPVI operator interchanges the value of two configuration parameters with compatible types. |
| CPDV: Configuration Parameter Default Value | The CPDV operator changes the value of a configuration parameter to the default value. |

TABLE 2. MUTATION OPERATORS RELATED TO SEQUENCES OF INTERACTIONS

| | |
|---|---|
| CMCR: Compatible Method Calls Replacement | The CMCR operators replace a method call by another compatible method call of the same called class. |
| MOR: Method Overloading Replacement | The MOR operator replaces the call to a method by the call to another overloaded method. |
| COI: Calls Order Interchange | The COI operator interchanges two calls made in the body of a method. |

TABLE 3. MUTATION OPERATORS RELATED TO GRAPHICAL USER INTERFACES

| | |
|---|---|
| GCPI: Graphical component Position interchange | The GCPI operator interchanges the position of two graphical components |
| GOCOC: Graphical Ordered Component Order Interchange | The GOCOC operator changes the order of the elements of a graphical ordered element (Trees, lists …). |
| GCD: Graphical Component Deletion | The GCD operator removes a graphical component of the graphical user interface. |
| DGCM: Disabled Graphical Component Modification | The DGCM operator converts an enable component into disable. |

## V. FLEXIBLE WEAK MUTATION

As discussed in section II.B, three techniques have been used to execute mutation analysis: strong, weak and firm mutation. Although these techniques have been studied and implemented in several tools, they cannot be used to execute mutation analysis at the system level, because none solve the problems described in Section IV.A.

Strong mutation requires anomalous states to propagate to the external environment through some output, or in the case of input-driven systems, to an intermediate state. Weak mutation, on the other hand, ignores propagation. Intermediate states of program execution are compared. Past uses of weak mutation have been at the unit level, where state is relatively small and localized. When mutating at the multi-class and system levels, however, state is distributed among all the objects, in static class variables, and in global variables. Thus an anomalous part of the state could be in many places.

When classes interact, a mutated part of a system can lead to an anomalous state in another part of the system. In weak mutation, after execution of the mutated statement, execution stops and the state of the mutant is checked against the state of the original system. If the anomalous state is in another, non-mutated object, traditional weak mutation will not discover it. In this case, the mutant would not be killed under weak mutation, even if an anomalous state exists in another object. These same issues appear in firm mutation.

### A. An illustrative example

To illustrate the problems in the previous paragraphs, let us consider a system with three classes and a test case that produces the sequence of calls shown in the sequence diagram of Figure 5. This illustrative example has three classes that interact with each other. A test case executes the functionality encapsulated in the method a*M1 (int x, int y)* of *ClassA*. This execution in turn executes some methods of *ClassA*, *ClassB* and *ClassC*, as well as creating and destroying an object of *ClassC*. Also, three mutants have been generated form the original system.

Table 4 shows the mutated statements in bold, and includes all the information required to understand the impact on the original system. *ClassA* has three methods: *aM1 (int x, int y)*, *aM2()* and *aM3 (int y)*; *ClassB* has two methods: *bM1()* and *bM2 (ClassA z)*; and *ClassC* has two methods: *cM1 (int x)* and *cM2 (int y)*.

Mutant 1 changes method *cM2 (int y)* of *ClassC*. This implies that after the execution of this method the field *c* of *ClassC* will have an erroneous value, since the statement *c=5y+y* has been replaced by *c=5y*y*.

Mutant 2 changes method *bM2 (ClassA z)* of *ClassB*. This change means the value of the field "*a*" of *ClassA* will be erroneous after the execution of the mutated statement. In this mutant the statement *b = 1+ z.a* has been replaced by *b = 1+ z.a++*.

Finally, the third mutant changes the method *cM1 (int x)* of *ClassC*. This change implies that the result of the method can be different after the execution of the mutated statement since the statement *return c==x* has been replaced by *return c == |x|*.

TABLE 4. ORIGINAL SYSTEM AND MUTANTS

**Original system**

```
class ClassA
{
    int a;
    …
    aM1 (int x, int y){…}
    aM2 (){…}
    aM3 (int y){…}
…}

class Class B
{
    int b;
    …
    bM1 (){…}
    bM2 (ClassA z)
    {
       …
      b = 1 + z.a
    …}
…}

class ClassC
{
     int c;
     …
    cM2 (int x)
    {
       …
       return c == x;
    }
    cM2 (int y)
    {
       …
       c = 5y + y
    …}
…}
```

**Mutant 1**

```
class ClassA{…}
class Class B{…}
class ClassC
{
      int c;
      …
      cM1 (int x){…}
      cM2 (int y){
      …
      c = 5y * y
      …}
…}
```

**Mutant 2**

```
class ClassA
{
      int a;
  …}
class ClassB
{
      int b;
      …
      bM2 (ClassA z){
        …
        b = 1 + z.a++
        …
      }
…}
class ClassC{…}
```

**Mutant 3**

```
class ClassA{…}
class ClassB{…}
class ClassC
{
      int c;
      …
      cM1 (int x){…
      return c == |x|;
      }
      cM2 (int x){…}
…}
```

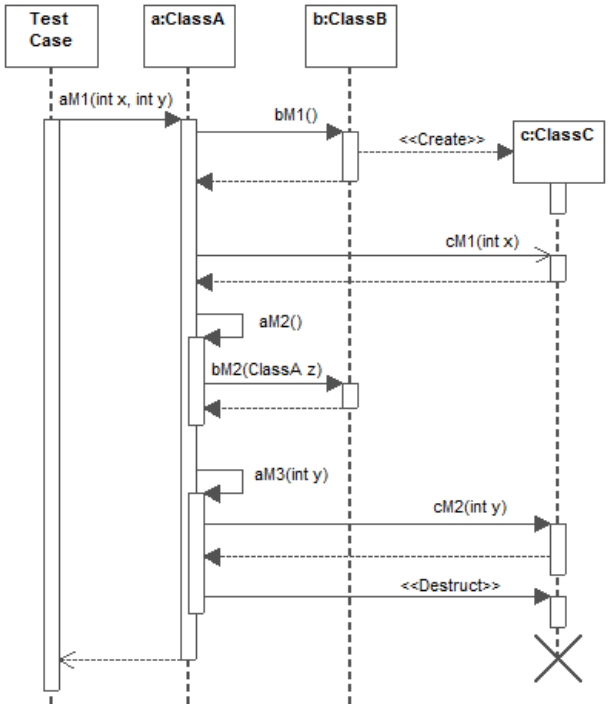



Figure 5. Illustrative example of interactions in a system

Now, suppose a tester executes mutant 1 under strong mutation. This mutant changes method *cM2 (int y)* in *ClassC*. This change causes the state of object *c* (Figure 5) to be anomalous when c*M2 (int y)* is executed.

Since strong mutation was used, the anomalous state of the *c* object will never be compared, because *c* will be destroyed. At the end of the execution there will be only two objects, *a* and *b*, and *c* does not affect the states of *a* and *b* at the end of the test. This is only important in input-driven systems.

On the other hand, suppose the tester uses weak mutation and executes mutant 2. The change of this mutant produces an anomalous state in the object *a* after the execution.

Just after the mutated statement, execution is stopped and the state of the *b* object is compared with the state of *b* in the original system. But as the anomalous state is in *a*, not *b*, the analysis will conclude that the mutant is alive.

Firm mutation has both of these problems or only the first, depending on the implementation and where the execution is stopped.

### B. *The solution: Flexible Weak Muation*

To solve the problems described in the previous section we have invented a new technique named *flexible weak mutation*. This technique can be used to determine when a mutant version is killed, solving the problems of weak and strong mutation, including the state comparison problem.

Flexible weak mutation is between weak mutation and strong mutation, but not in the same way that firm mutation is. In flexible weak mutation, the point where the execution is stopped is decided dynamically and sometimes the execution is not stopped. Flexible weak mutation works as follows:

- When the original system is executed, a trace of the execution is stored. The state of each unit that executes a method is stored at the beginning of each method, and immediately before each possible method output. At the end of the execution of the original system, there will be a trace with the sequence of states for all units.
- Then, when a mutant is executed, the state of each unit that executes a method in the mutant system is compared to the state stored in the trace of the original system, both at the beginning and at the end of the method. In other words, the trace that the mutant generates is compared to the trace of the original system during execution.
- If during the execution of a mutant an anomalous state is found, execution of the mutant is stopped and it is considered killed.
- If an anomalous state is not found, the execution is not stopped and continues until the end.

There are two important differences here.

1. The execution is only stopped when an anomalous state is found, instead of a predefined point in execution (a dynamic decision).
2. To stop execution when the anomalous state is found, the state has to be compared at different times during execution. In flexible weak mutation the state of the mutant is checked multiple times, instead only once.

If the anomalous part of the state is in a non-mutated object, checking at multiple times and places increases the chance of finding the anomaly, allowing the mutant to be killed during execution. However, execution is allowed to continue, so if a mutant does not result in an immediately obvious state anomaly, execution can continue until final output.

Flexible weak mutation also allows the state to be compared across multiple objects. In flexible weak mutation, a trace of the execution of the original system is stored, together with the state of each unit that executes a method. Later, this trace is compared at runtime with the trace left by each mutant.

### C. *An example of execution with flexible weak mutation*

Figure 6 shows an example using flexible weak mutation. This example takes only the test case of Figure 5 and the three mutants of Table 4. The test inputs used by the test case are one for parameter x and three for parameter y of the method *aM1 (int x, int y)*.

The first step to perform the analysis is to execute the original system and to store a trace of the execution with the states of the classes that execute a method. Figure 6-(A) shows this.

The sequence of execution is represented by rectangles in the top of the figure. Each rectangle is composed of the name of the method at the beginning and at the end, and a sequence of sub-rectangles representing the method calls inside. At the beginning and at the end of each method, the state of the object that executes the method is stored. These

states are represented in the figure with the name of the object in capital letters.

After the trace of the original system is stored, the mutants can be executed. Figure 6-(B) shows the execution of mutant 1. During execution, the states of the objects that execute a method are compared with the states stored in the trace at the beginning and at the end of each method.

After the execution of the mutated part, the object *c* has an anomalous state that is compared with the expected state stored in the trace. At this point, a difference between states is found, so execution is stopped and the mutant is killed. Figure 6-(C) shows the execution of mutant 2. This is an example of how flexible weak mutation solves the problem with weak mutation. The process is similar to the execution of the mutant 1. The difference is that after execution of the mutated part, the anomalous state is in object *a*, instead of object *b*, which is the mutated object. After execution of the mutated statement, the state of *b* is compared with the expected state. The states are equal so execution is not stopped, so at the end of method *aM2()*, which called method *bM2()*, the state of the object *a* is compared and the unexpected state is found. Then, the execution is stopped and mutant 2 is killed.

Figure 6-(D) illustrates the situation when the execution of mutated part does not leave any object in an anomalous state. In this example, mutant 3 is executed, but with the value 1 for parameter *x* the mutated statement (*return c == |x|;*) produces the same result as the original statement (*return c == x;*). Here, an anomalous state is not found and the execution of the mutant is not stopped, so the mutant is not killed.

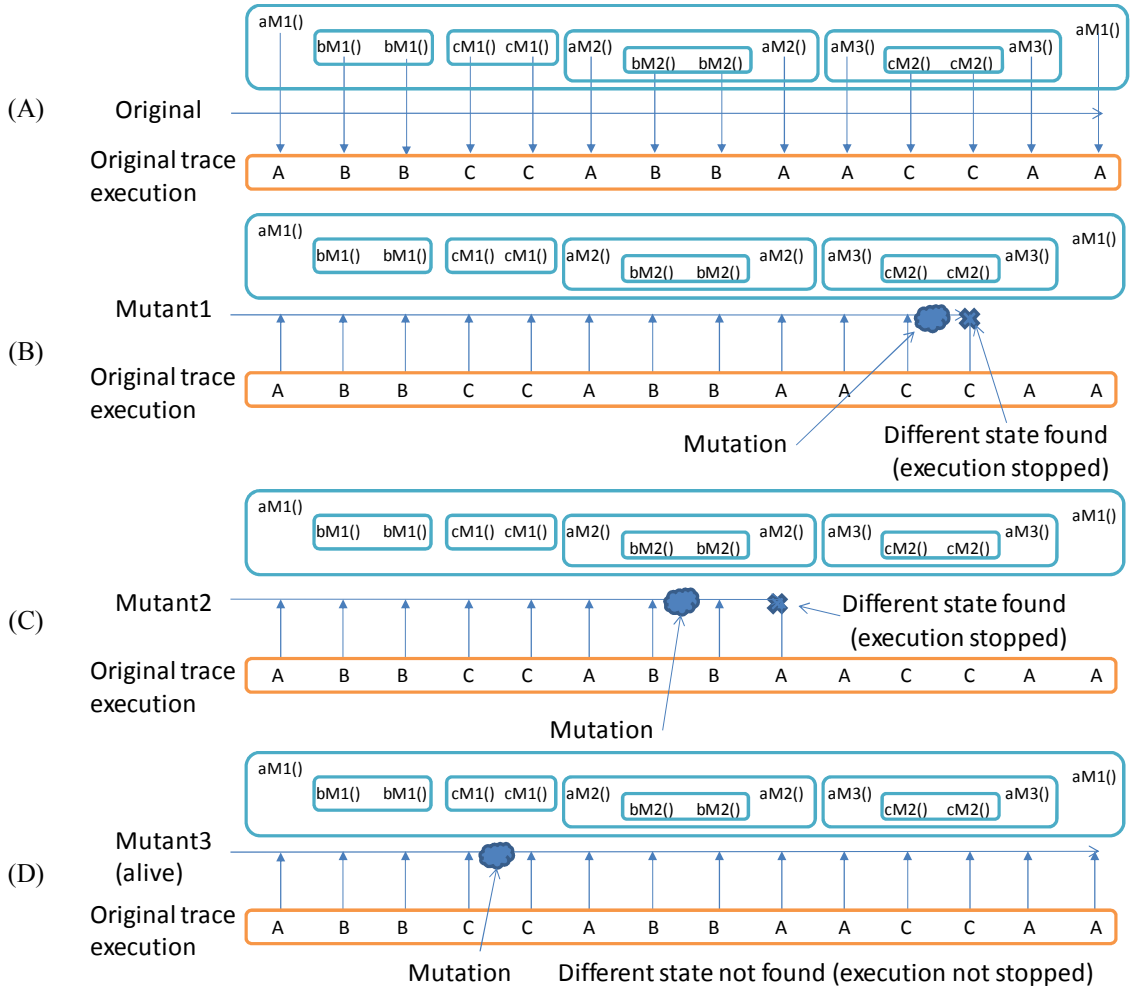



Figure 6. (A) Execution of the original system, (B) execution of mutant 1, (C) execution of mutant 2, (D) execution of mutant 3

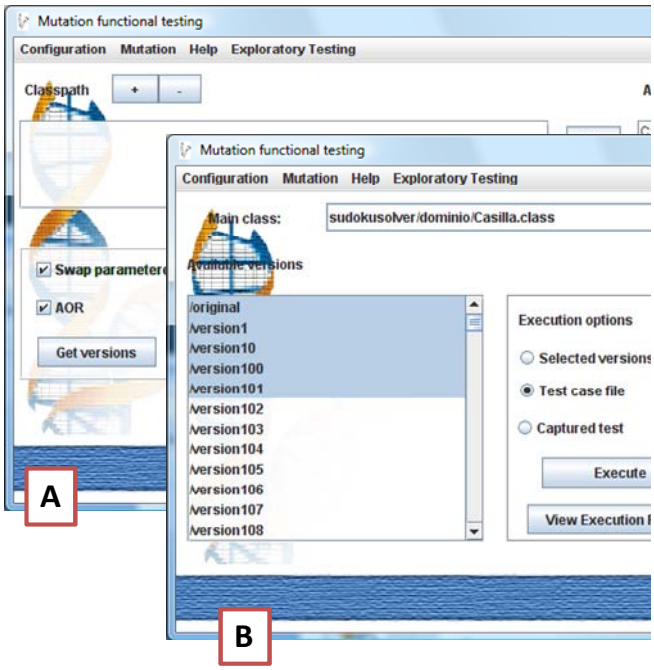



Figure 7. Bacterio screen shots

behavior of the operators. Finally, a solution for detecting equivalent mutants has not been implemented in Bacterio. This work is ongoing.

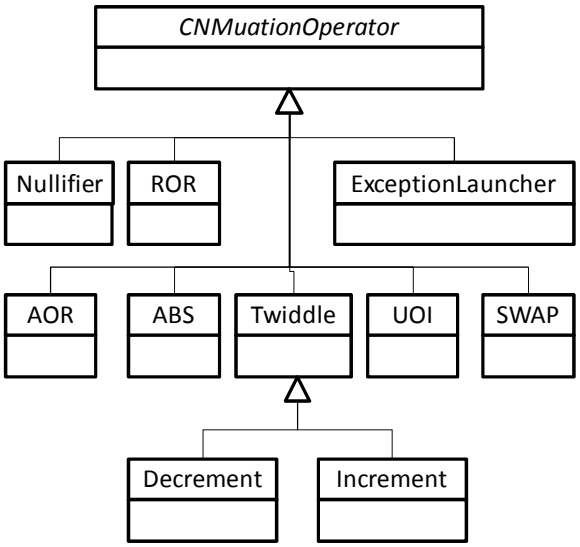



Figure 8. Class diagram of mutation operators

## VI. A TOOL FOR SYSTEM LEVEL MUTATION: BACTERIO

System level mutation is being implemented in a tool named Bacterio. Bacterio mutates Java programs, implementing the ideas in Sections III and IV. The next paragraphs describe which solutions Bacterio uses.

The main problem to apply mutation at the system level is to compare the original system with each mutant as in Section IV.A. Bacterio uses flexible weak mutation. Bacterio instruments the Java bytecode to store a trace of the execution and compares the execution of each mutant with the stored trace.

Bacterio also solves the problem related to managing an entire system (section IV.B). Bacterio both approaches: replace the complete system and only the mutated object (as discussed in section IV.B). The first approach stores one copy of the entire system for each mutant (the space intensive approach). The second approach stores a copy of the class that has the mutated part and replaces this during execution. The two approaches have been implemented to allow them to be compared empirically. Figure 7 shows Bacterio's two main screens: (A) mutant generation screen and (B) mutant execution screen.

Some of the mutation operators included in the current version of Bacterio (available at *http://alarcos.esi.uclm.es/testing/*) implement traditional changes (AOR, ROR, ABS and UOI), whereas others are inspired by the interface mutation operators defined by Ghosh and Mathur [7] (Swap, Parameter increment, Parameter decrement, Nullify and Throw exceptions). We are currently developing operators for system and functional levels. For this, all mutation operators must be specializations of the CNMutationOperator class (Figure 8), an abstract type that defines the common structure and

## VII. CONCLUSIONS

This paper presents a new mutation technique for multi-class and system level testing. The tester can perform mutation analysis on complete systems to test functionalities provided by some components or subsystems that interact together. Also, this kind of mutation establishes a base to apply mutation testing to test emergent properties such as security, usability, safety, etc.

This paper also analyzes the problems of applying mutation to the system level and proposes solutions, the most important being flexible weak mutation. This new method to perform mutation solves several problems with applying mutation to the system level.

These ideas are being implemented in a tool, Bacterio, to demonstrate feasibility. Also, experiments are being designed to evaluate the effectiveness of the ideas presented here.

However, there is still much work to do. A major task is to implement and improve the mutation operators. As mutation is a fault based technique, we are analyzing typical faults at the system level and designing mutation operators to simulate them. Also there is work related with the detection of equivalent mutants, so we have to design a solution to find equivalent mutants at system level.

Finally, we plan to extend this work to test other characteristics such as security and usability.

## VIII. REFERENCES

[1] Offutt, J., P. Ammann, and L. Liu. *Mutation Testing Implements Grammar-Based Testing*. Second Workshop on Mutation Analisys (Mutation 2006). 2006: Raleigh, NC. p.93-102

[2] Ammann, P. and J. Offutt, *Introduction to software testing*. 2008: Cambridge University Press.

[3] Jia, Y. and M. Harman, *An Analysis and Survey of the Development of Mutation Testing*. 2009, CREST Centre, King's College London: London, UK.TR-09-06

[4] DeMillo, R., R.J. Lipton, and F.G. Sayward, *Hints on test data selection: Help for the practicing programmer.* IEEE Computer, 1978. **11**(4): p. 34-41.

[5] Ma, Y.-S., J. Offutt, and Y.R. Kwon, *MuJava: an automated class mutation system.* Software Testing, Verification and Reliability, 2005. **15**(2): p. 97-133.

[6] Delamaro, M., J. Maldonado, and A. Mathur, *Interface mutation: An approach for integration testing.* IEEE Transactions on Software Engineering, 2001. **27**(3): p. 228-247.

[7] Ghosh, S. and A.P. Mathur, *Interface mutation.* Software Testing, Verification and Reliability, 2001(11): p. 227-247.

[8] DeMillo, R. and J. Offutt, *Constraint-Based Automatic Test Data Generation.* IEEE Transactions on Software Engineering, 1991. **17**(9): p. 900-910.

[9] King, K.N. and A.J. Offutt, *A Fortran language system for mutation based software testing.* Software: Practice and Experience, 1991. **21**(7): p. 685-718.

[10] Howden, W.E., *Weak mutation testing and completeness of test sets.* IEEE Transactions on Software Engineering, 1982. **8**(4): p. 371–379.

[11] Girgis, M. and M. Woodward. *An integrated system for program testing using weak mutation and data flow analysis*. 8th international conference on Software engineering. 1985. London, England: IEEE Computer Society Press Los Alamitos, CA, USA. p.313-319

[12] Marick, B. *The weak mutation hypothesis*. Symposium on Testing, analysis, and verification. 1991. Victoria, British Columbia, Canada: ACM New York, NY, USA. p.190-199

[13] Offutt, A.J. and S.D. Lee, *An Empirical Evaluation of Weak Mutation.* IEEE Transactions on Software Engineering, 1994. **20**(5): p. 337-344.

[14] Woodward, M. and K. Halewood. *From weak to strong, dead or alive? An analysis of some mutation testing issues*. Second Workshop on Software Testing, Verification, and Analysis. 1988. Banff, Canada. p.152-158

[15] Offutt, J. *A practical system for mutation testing: help for the common programmer*. IEEE International Test Conference on TEST: The Next 25 Years. 1994: IEEE Computer Society Washington, DC, USA. p.824 - 830

[16] Polo, M., M. Piattini, and S. Tendero, *Integrating techniques and tools for testing automation.* Software Testing, Verification and Reliability, 2007. **17**(1): p. 3-39.

[17] Barbosa, E.F., J.C. Maldonado, and Auri Marcelo Rizzo Vincenzi, *Toward the determination of sufficient mutant operators for C.* Software Testing, Verification and Reliability, 2001. **11**(2): p. 113-136.

[18] Delamaro, M.E., J.C. Maldonado, and A. Vincenzi. *Proteum/IM 2.0: An Integrated Mutation Testing Environment*. 1st Workshop on Mutation Analysis (MUTATION'00). 2001. San Jose, California. p.91-101

[19] Irvine, S., T. Pavlinic, L. Trigg, J. Cleary, S. Inglis, and M. Utting. *Jumble Java Byte Code to Measure the Effectiveness of Unit Tests*. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. 2007. Windsor, UK: IEEE Computer Society Washington, DC, USA. p.169-175

[20] Schuler, D. and A. Zeller. *Javalanche: efficient mutation testing for Java*. European Software Engineering Conference (ESEC)/Foundations of Software Enginering (FSE). 2009. Amsterdam. p.297-298

[21] SpringSoft, *Certitude Functional Qualification System*. 2009, USA. http://www.springsoft.com/assets/files/Datasheets/SS_Certitude_DS_D2.pdf, Last access: 8th March 2010

[22] Offutt, J. and W. Xu. *Generating Test Cases for Web Services Using Data Perturbation*. Workshop on Testing, Analysis and Verification of Web Services. 2004. Boston, Massachusetts.

[23] Untch, R., A. Offutt, and M. Harrold. *Mutation analysis using program schemata*. International Symposium on Software Testing, and Analysis. 1993. Cambridge, Massachusetts: ACM Press. p.139-148

[24] DeMillo, R., E. Krauser, and A. Mathur. *Compiler-integrated program mutation*. Fifteenth Annual Computer Software and Applications Conference. 1991. p.351-356

[25] Ma, Y., Offutt, J, *Description of method-level mutation operators for Java*. 2005, Public online. http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf, Last access: 12 January 2010

[26] Bowser, J., *Reference manual for Ada mutant operators*. 1988, Georiga Institute of Technology: Atlanta, Georgia.Technique Report GITSERC-88/02

[27] Agrawal, H., R. DeMillo, R. Hathaway, W.M. Hsu, W. Hsu, E. Krauser, R.J. Martin, A. Mathur, and E. Spafford, *Design of Mutant Operators for the C Programming Language*. 1989, Purdue University: West Lafayette, Indiana

[28] Derezinska, A., *Advanced mutation operators applicable in C# programs*, in *Software Engineering Techniques: Design for Quality*. 2006, Springer. p. 283-288.

[29] Tuya, J., M. Suárez-Cabal, and C. la Riva, *Mutating database queries.* Information and Software Technology, 2007. **49**(4): p. 398-417.

[30] Offutt, A. and J. Pan, *Detecting Equivalent Mutants and the Feasible Path Problem.* Wiley's Software Testing, Verification, and Reliability, 1997. **7**(3): p. 165-192.

[31] Hierons, R., M. Harman, and S. Danicic, *Using program slicing to assist in the detection of equivalent mutants.* Software Testing, Verification and Reliability, 1999. **9**(4): p. 233-262.

[32] Grün, B.J.M., D. Schuler, and A. Zeller. *The Impact of Equivalent Mutants*. IEEE International Conference on Software Testing, Verification, and Validation Workshops. 2009. Denver, Colorado, USA: IEEE Computer Society Washington, DC, USA. p.192-199

[33] Polo, M., M. Piattini, and I. García-Rodríguez, *Decreasing the cost of mutation testing with second-order mutants.* Software Testing, Verification and Reliability, 2008. **19**(2): p. 111-131.

[34] Mresa, E. and L. Bottaci, *Efficiency of mutation operators and selective mutation strategies: An empirical study.* Software Testing Verification and Reliability, 1999. **9**(4): p. 205-232.