Analysis of the Development Process of a Mutation Testing Tool for the C++ Language

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez
UCASE Software Engineering Group, Department of Computer Science and Engineering
University of Cádiz, Cádiz, Spain
Email: {pedro.delgado, inmaculada.medina, juanjose.dominguez}@uca.es

***Abstract*—Mutation testing is a fault-based software testing technique to measure the quality of a test suite depending on its ability to detect faults in the code. This technique has been applied to an assortment of languages of very diverse nature since its inception in the late 1970s. However, the researchers have postponed its development around C++ in favor of other mainstream languages. This paper aims to survey the mutation testing research regarding C++, studying the existing tools and approaches. To the same extent, we discuss the different aspects that should be taken into account in the construction of a comprehensive mutation tool for this language, from the analysis of the code to the execution of the mutants. In addition, we expound how the technique can be assessed so that it can contribute effectively in the composition of a complete test suite. The findings in this paper pose that the construction of a mutation tool for this language is complex, but still realizable.**

Keywords-Mutation testing; Mutation tool; C++.

I. INTRODUCTION

Mutation testing is a fault injection technique used to determine the ability of a test suite locating errors in the code [1]. The effectiveness of error detection of a test suite is defined as the percentage of faults that can be detected by their test cases. The technique involves the creation of *mutants*, i.e., versions of the original program with a simple syntactic change. These faults are injected in the code through the mutation operators, which are based on common mistakes made by programmers in a certain programming language.

Mutation testing aims to ensure that a test suite is able to detect all those typical mistakes when comparing the output of the original program and the mutated version for the same test cases; a mutant is *killed* when the output is different for at least one test case, but remains *alive* if the output keeps unaltered. In this latter case, either a new test case is needed to detect the fault or the mutant is completely *equivalent* to the original program.

Mutation testing is a white-box testing technique, so it has to be studied around a particular language. Thus, the technique has been successfully applied to several languages, finding tools that automate the generation of mutants for a wide range of them [2]. Unlike other languages, C++ is clearly behind in the research and in practice. In literature, most research on mutation testing has been focused on procedural programming paradigm. For object-oriented (OO) languages, the research has put the focus on Java or C# [3]. In contrast, the few existing works about mutation testing and C++ [4], [5] are unfinished and various matters are pending.

The overall goal of this paper is to analyze the state of the art in order to devise the construction of a mutation tool for C++. The matters that should be handled for that purpose have been surveyed, from the composition of a catalog of mutation operators to the execution of tests and the manner to assess the results. Consequently, the information introduced throughout the sections lays the foundations for the application of mutation testing to C++, exposing an approach to follow in the future based on the abstract syntax tree for the insertion of errors.

The current state regarding the mutation operators, the mutation tools and the language itself are explored in Section II. Section III deals with the operators at different levels, the steps to accomplish in the creation of a mutation tool and the techniques to mutate the code. In Section IV, the way to gear the evaluation of the results in order to judge the operator behavior is commented, giving special emphasis to the issues that can arise in the analysis of results. Finally, the last section presents the conclusions and also the future work.

II. RELATED WORK

This section looks in depth the existing background around C++ and mutation testing.

A. Mutation Operators

To illustrate the underlying idea in mutation testing, we can consider the C++ code fragment below:

$$if(a > 100)\{...\}$$

If we have defined a mutation operator replacing the relational operator '$>$', the fragment above can be modified creating a mutant like the following:

$$if(a < 100)\{...\}$$

A set of mutation operators can be defined for each of these levels [2], [6]:

- **Unit level**: Standard operators applied indistinctly to a function or method, checking its correctness. These operators are usually known as *traditional operators*.
- **Class level**: This level deals with the mutation of OO features.
- **Integration level**: Intermediate level between the unit and the system levels, checking the function invocations.

 ISBN: 978-1-61208-346-9

- **Multi-class level**: Operators at this level are intended to test a complete program: interactions among functions, classes, etc.

As advanced in the introduction, we can state that the development of mutation testing with respect to C++ is underrepresented. Regarding a particular set of mutation operators for this language, the research accomplished is really scarce as we cannot find a comprehensive catalog of operators.

However, two attempts have been performed up to now. The first work in [4] composed a set of traditional operators; it is named in [7], a paper aiming to supply the equivalence of operators among different languages. These operators are based on the operators defined for Ada and the Fortran operators used by the tool *Mothra* [8]. This approach cataloged the operators in four blocks: *operand replacement*, *operator insertion*, *arithmetic operator replacement* and *relational operator replacement*. These groups and their operators are shown in Table I. On the other hand, several mistakes regarding OO features have been enumerated around C++ in [5]. This paper poses five categories of possible faults: *Inherit*, *Associate*, *Access*, *Object* and *Member*. Nevertheless, the three first blocks are applied to the Unified Modeling Language (UML) specification and only the errors belonging to the *Object* and *Member* groups refer to the C++ code. The faults exposed in that paper are summarized in Table II. The research regarding both approaches seems given up as no new progress has been published since then.

B. Mutation Tools

At present there are a variety of tools for several languages implementing mutation testing through different techniques [2], as *Mothra* for Fortran [8], *MuJava* for Java [9] or *MILU* for C [10]. Concerning C++, only commercial tools can be found tackling the mutation analysis for the C++ code: *Insure++* [11] from Parasoft, *PlexTest* [12] from ItRegister and *Certitude* [13] from SpringSoft. These three tools appear in this known survey around mutation testing [2]. In that paper, it is also stated that the *ESTP* tool can be applied to C++, but it is only devoted to the C language actually.

TABLE I. STANDARD OPERATORS PROPOSED BY ZHANG [4].

Block	Operator	Description
Operand Replacement Ops.	OVV	Variable replaced by a variable
	OVC	Variable replaced by a constant
	OVA	Variable replaced by an array reference
	OVP	Variable replaced by a pointer reference
	OCV	Constant replaced by a variable
	OCC	Constant replaced by a constant
	OCA	Constant replaced by an array reference
	OCP	Constant replaced by a pointer reference
	OAV	Array reference replaced by a variable
	OAC	Array reference replaced by a constant
	OAA	Array reference replaced by an array reference
	OAP	Array reference replaced by a pointer reference
	OAN	Array name replaced by an array name
	OPV	Pointer reference replaced by a variable
	OPC	Pointer reference replaced by a constant
	OPA	Pointer reference replaced by an array reference
	OPP	Pointer reference replaced by a pointer reference
	OPN	Pointer name replaced by a pointer name
Operator Insertion Ops.	IBO	Binary Operators Insertion
	IOU	Unary operator insertion
Arithmetic Operator Replac. Ops.	AOR	Arithmetic operator replacement
Relational Operator Replac. Ops.	ROP	Relational operator replacement

TABLE II. FAULTS IDENTIFIED BY DEREZIŃSKA [5] FOR THE *Object* AND *Member* CATEGORIES.

Category	Description
Object	Calls a same function member from a different object of the same class.
	Calls a function from an object of a different class, but both classes have the common base class.
	Calls a function from the derived class instead of the base class.
Member	Calls a different (complementary) function member.
	Calls a function inherited from the base class.
	Swaps calling of function members in a class.
	Swaps calling of functions inherited from one class.
	Accesses the different data in the same object.

Table III displays a summary with the main characteristics of the tools discussed (shown in [2]). All of them are applicable to C and C++, as both languages share much of their syntax. A description of these commercial products is given below:

- *Insure++*: This tool uses mutation testing as one more technique to enhance the software quality. Its approach is somewhat different from classical mutation testing because it only creates *functionally equivalent mutants*. These equivalent mutants are expected to pass the tests (are still alive after their execution against the test suite) instead of failing. Insure++ only performs some standard mutations as mentioned in [14].
- *PlexTest*: This product implements a *highly selective* mutation testing. Thus, it only performs the mutation of deletion, i.e., the removal of an instruction. This approach tries to avoid the generation of equivalent mutants present in non-selective and full-selective mutation testing.
- *Certitude Functional Qualification System*: This tool combines the mutation testing technique with static analysis, qualifying a program functionally and discovering faults that might not be detected otherwise. Although this product has also been used for the analysis of software systems, it is now addressing the microelectronics industry in order to simulate the performance of a digital circuit before the design is finally accomplished.

As a conclusion of the information shown, we can state that these tools are different in terms of the operators supported and the techniques to accelerate the testing process, but they are not applicable to the whole language. Moreover, the products presented are not absolutely centered on mutation testing (as Insure++) or they are not only used for C++ (as Certitude).

Java and C# are the OO languages that have been mostly tackled by researchers. The construction of frameworks for these OO languages are mainly based on the insertion of faults directly in the bytecode, like *MuJava* [9], and also on the reflection mechanism to analyze the original program and

TABLE III. EXISTING MUTATION TESTING TOOLS FOR C++.

	Application	Year	Character	Available
Insure++	C/C++	1998	Source Code Instrumentation	Commercially
Plextest	C/C++	2005	General	Commercially
Certitude	C/C++	2006	General	Commercially

determine where the operators can be used. Derezińska et al. [14] has proposed and built two different tools: *CREAM*, using a parser-based approach, and *ILMutator*, which manipulates both the meta-data and the intermediate code generated from C# to insert the faults modeled by the operators. The former approach is better in identifying where the operators can be applied and complying with the correctness conditions, but the latter is more efficient as no recompilation is needed.

C. The C++ Language

C++ is a complex programming language when compared to other similar general purpose languages, such as Java or C#. This language includes a great variety of alternatives. Besides, many of the features are confusing for the programmers, being usual that they neglect using some of them. Thus, we found surprising that this popular language has been omitted in the field of mutation testing up to now. Because of being so popular, we consider that it is worthwhile addressing the application of this technique to C++ in order to harness the contributions of mutation testing for this language, which presents various complicated characteristics for the programmers.

In this regard, we can mention several particular features, such as the destruction and construction of objects, the existence of pointers and references, the use of exception handling or the inclusion of templates. Multiple inheritance is another characteristic which can lead to the creation of new specific mutation operators as well as should be taking into account in general when tackling class-level operators relating inheritance, as more than a single class needs to be considered.

C++ is a language in continuous change and some standards have been approved since 1998, when the first standard appeared. A new standard, *C++11* [15], was ratified in 2011 to replace the previous standard, *C++03*. It represents the first substantial change since 1998. Besides some minor modifications, the *C++11* standard presents important changes. This standard also provides new features and extends the *C++ Standard Library*. Currently, the changes introduced by this standard are taking place gradually and even compilers are not fully adapted to the new proposals. Nonetheless, the *C++14* standard is already being prepared to provide the language with new functionalities.

III. DEVELOPMENT PROCESS OF A C++ MUTATION TOOL

Mutation testing confronts two main challenges when constructing a mutation tool. The technique to insert mutations in the code can be useful to handle certain features of the language, but might not cover all its elements. In the case of C++, as a mainstream language which provides a great range of alternatives (unlike specification languages, for instance), it is necessary to tackle the structures of the language in an uniform manner so that the mutations are the expected ones in every operator addessing the same elements. Secondly, the generation and execution of mutants to obtain the results suppose a high computational cost, especially when considering the size of today's programs. These and other concerns are commented in the following subsections.

A. Catalog of Operators

C++ is a multi-paradigm programming language which is considered as an enhancement of C with new features, such as the manipulation of objects [15]. Mutation testing can be applied at different levels of the language, as mentioned in Section II-A. Taking into account the dimension of C++, it seems challenging to apply this technique to every level of the language all at once. In addition, the more operators are included in the process, the more mutants will be produced, with a consequent increase in the computational expenses. Thus, a new catalog of operators can be composed for each different level so that they can be used conveniently.

The research around C is the starting point concerning traditional operators. The similarity between these two programming languages is well-known, as C++ is derived from C. Most of the works and tools created for C are based on mutation operators described by Agrawal et al. in [16]. In the aforementioned paper, the operators are separated into four main categories: *statement, operator, variable* and *constant mutations.* Because of the compatibility with C, all operators can be adapted to the language (see Table IV). However, some of them should be slightly modified in their implementation with respect to some characteristics in C++, such as the possibility to declare a variable at any point in a block.

According to the class level, we should survey the existing mutation operators regarding the main OO features in other languages, undertaking the two following steps:

1) To observe how the particular characteristics of C++ alter the operators, determining if they can be adapted to the language or they have to be rejected instead.
2) Whenever possible, to add specific operators for this language and its distinguishing features (see Section II-C).

Many papers centered on this level can be found and sets of mutation operators have been defined around the OO programming [17], [18]. A similar survey to this paper around OO languages studies the class mutation operators [3]. In Table V, we expose the catalog of operators defined by Offutt et al. for Java [17], which is one of the most prominent and complete set of class mutation operators. Regarding this list and the aforementioned step (1), most of these operators could be adapted with some changes but the same purpose (for example, the *super* keyword is not used in C++), but others are likely to be excluded, such as the last four operators: they refer to methods for which there is no a convention in C++.

To conclude, we have to note that the number of levels of the language is not fixed at all. For instance, another level of operators could be created only for the *C++ Standard Library*, which provides a great range of extra functionalities commonly used. Thus, the level chosen by the user depends on the program characteristics and the type of testing intended to perform. In other words, the subset of operators to be applied should be selected after a preprocessing of the program under

 ISBN: 978-1-61208-346-9

TABLE IV. MUTATION OPERATORS FOR THE C LANGUAGE PROPOSED BY AGRAWAL ET AL. [16].

Block	Operator	Description
Statement mutations	**STRP**	Trap on Statement Execution
	STRI	Trap on if Condition
	SSDL	Statement Deletion
	SRSR	return Statement Replacement
	SGLR	goto Label Replacement
	SCRB	continue Replacement by break
	SBRC	break Replacement by continue
	SBRn	Break Out to nth Enclosing Level
	SCRn	Continue Out to nth Enclosing Level
	SWDD	while Replacement by do-while
	SDWD	do-while Replacement by while
	SMTT	Multiple Trip Trap
	SMTC	Multiple Trip Continue
	SSOM	Sequence Operator Mutation
	SMVB	Move Brace Up or Down
	SSWM	Switch Statement Mutation
Operator mutations	***Obom (Binary Operator Mutations)***	
	Ocor	Comparable operator replacement
	Oior	Incomparable operator replacement
	Ouor (Unary Operator Mutations)	
	Oido	Increment/Decrement
	OLNG	Logical Negation
	OCNG	Logical context negation
	OBNG	Bitwise Negation
	OIPM	Indirection Operator Precedence Mutation
	OCOR	Cast operator replacement
Variable mutations	**Vsrr**	Scalar Variable Reference Replacement
	Varr	Array Reference Replacement
	Vtrr	Structure Reference Replacement
	Vprr	Pointer Reference Replacement
	VSCR	Structure Component Replacement
	VASM	Array reference subscript mutation
	VDTR	Domain Traps
	VTWD	Twiddle Mutations
Constant mutations	**CRCR**	Required Constant Replacement
	Cccr	Constant for Constant Replacement
	Ccsr	Constant for Scalar Replacement

TABLE V. MUTATION OPERATORS AT THE CLASS LEVEL PROPOSED BY OFFUTT ET AL. [17].

Block	Operator	Description
Encapsulation	**AMC**	Access modifier change
Inheritance	**IHI**	Hiding variable insertion
	IHD	Hiding variable deletion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	*super* keyword insertion
	ISD	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	**PNC**	*new* method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCD	Type cast operator deletion
	PCC	Cast type change
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-specific features	**JTI**	*this* keyword insertion
	JTD	*this* keyword deletion
	JSI	*static* keyword deletion
	JSD	*static* keyword deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOC	Reference assignment and content assignment replacement
	EOA	Reference comparison and content comparison replacement
	EAM	Acessor method change
	EMM	Modifier method change

test (PUT), so that the operators can be adjusted as much as possible to the concrete application.

B. Applying Mutations

Mutation testing process can be subdivided into three principal stages: the analysis of the code, the generation of the mutants and the execution of the test suite. For the development of a mutation tool, the implementation of these three steps is necessary, so we describe the purpose of these parts below:

- **Analyzer**: This module first acts getting the program under test and the set of operators defined. Its role is to determine the mutation operators that can be applied, that is, the mutation locations for each of the operators.
- **Generator**: Taking into account the analysis accomplished in the previous phase, the function of the generator is to create the mutant for its later execution. In the production of the mutant, the generator should take care of the correctness conditions so that the mutant generated is not an invalid mutant, i.e., it can be compiled because the program has no errors.
- **Test suite runner**: This module executes the mutants against a test suite defined for the PUT, finally classifying them into alive (the mutant has not been detected by any test case), dead (the mutant has been killed by one or more test cases) or invalid.

The two first phases are quite related as they require an implementation technique that actually allows us to detect and then create a faulty version of the program intended to test (the third step will be addressed in the Section III-C). The options shown in Section II-B to mutate the code are not available in C++, that is, there is nothing similar to the bytecode and a parser-based approach cannot be used resorting to the reflection mechanism in order to check the state of objects at runtime. In the case of the approach used in *ILMutator* [14], neither that kind of intermediate code exists in C++. However, a similar technique to insert faults into the code can be followed as the compilers use an internal representation in the form of abstract syntax tree (AST). Bearing in mind the complexity of C++, we consider that reusing this representation can be fruitful instead of parsing the high-level source code; the AST seems to be much more comprehensible than the intermediate code used in C#, which limits the application of the operators. Notwithstanding, recompilation will be needed, so we cannot leverage this approach for a more efficient mutation system as in C#.

Regarding the generator, we discern two basic options for the mutant creation. The first is the generation of the mutants so that each one can be run as a standalone program, making a duplicate of the original version but containing the mutated files. The second option supposes creating new files only for those where the mutation was included, using the same build system of the original program. In the former, a copy of every file involved in the project will be needed for each mutant, but we can resort to different techniques to save space in disk. In the latter, the mutation tool has to be aware of the build system to perform the appropriate changes so that the mutated files are compiled, which could be a cumbersome task.

 ISBN: 978-1-61208-346-9

C. Execution of Tests

The goal of mutation testing is to determine how good is a test suite defined for a program. Thus, we need to automate the execution of the mutants against the test suite. The process stars assuring that the test suite is successfully passed by the original program. Then, the mutants are run and a comparison can be performed in order to observe whether the changes introduced in the program have been detected or not.

Nevertheless, the connection between the test suite defined by the tester and the mutation tool can be a concern when it comes to automate the test suite execution: the test suite should be able to provide the results so that the tool can retrieve and process them. On the one hand, there is no a prevailing testing framework for C++ currently, unlike Java for instance, where *JUnit* is broadly used. Hence, we can decide to give support for one or more frameworks, but we advocate the design of a library which can be used whatever the way the test suite was developed. This latter approach allows for the execution of tests consistently as the same methods will be used to report the output. On the other hand, when testing mutants generated from class mutation operators, posing different test scenarios of the usage of objects is needed. Besides, various kinds of test cases are required within the scenarios depending on the need of testing that concrete situation. This aspect is different from other unit tests or languages where some values are provided to the program and simply an output is expected.

IV. How to Evaluate the Results

In this section, we discuss several matters to be considered when assessing the results obtained with mutation testing.

A. Quality of Operators

The evaluation of the catalog of mutation operators is a significant step in the application of mutation testing to check if the operators are really effective for the purpose of the technique. The most basic and used calculation for the adequacy level of the test suite is the mutation score in each particular mutation operator. The mutation score indicates the percentage of dead mutants versus the number of non-equivalent mutants. In other researches, further dimensions have been studied. Smith et al. [19] proposed to determine the quality of mutation operators by relating the possible states of the mutants after the test suite execution: killed by the initial test suite, killed by a new test case, killed by a new test case specifically defined to kill another mutant or not killed.

Estero-Botaro et al. [20] defined some terms to evaluate the operators for Web Services Business Process Execution Language (WS-BPEL) 2.0, like "weak mutant", which is killed by every test case in the test suite, or "resistant mutant", which is killed by a single test case. Thus, the quality of a mutation operator can be determined analyzing some conditions. In summary, the operator should generate a low number of invalid and equivalent mutants (as they do not help in the mutation analysis), but produce few weak mutants as well as the more resistant mutants the better.

Derezińska [18] exposed an idea of the effectiveness for the class mutation operators (specifically for C#), posing various questions (gathered in Table VI) that should be answered to deem the usefulness of an operator. In addition, this author qualifies a test suite calculating the quotient of the number of test which killed mutants generated by an operator over all test runs performed on these mutants (see Equation 1).

TABLE VI. Questions posed by Derezińska [18] to assess the effectiveness of mutation operators.

	Questions
1	Does an operator can be applied in real programs to simulate faults of programmers?
2	Are any invalid mutants generated by an operator?
3	Does an operator generate many equivalent mutants?
4	Is an operator effective in assessing the quality of given test cases? If a mutant is not killed by a given test suite, is it easy to create test cases which kill it?

$$Effectiv. = \frac{Killed\,test\,runs}{(Total\,mut. - Equiv.\,mut.) * Total\,tests} * 100 \qquad (1)$$

These different kinds of measuring the quality of the testing process should be taking into account when applying this technique to C++, depending on the evaluation intended to perform as well. The operators producing a great amount of mutants should be also studied in depth because they can entail much computation.

B. Issues in the Assessment of Results

When evaluating the results obtained from the application of mutation testing, several issues should be considered. Firstly, the different behavior of the programs can alter in a great degree the results produced with an operator. For example, an operator can involve the creation of many mutants in some cases whereas there are no mutants in others. Thus, we have to take care of this matter in order to generalize results.

Secondly, the initial test suite deserves special attention. Especially in the mutants at the class level, the test suite may not cover every class or member that has been injected a fault. Hence, the mutant will not be killed by any test case, but they cannot be considered as equivalent. This fact occurs very often in C++, where a program is formed by different source files and classes, some of them providing secondary functionalities. This issue has been tackled by Segura et al. in [21], defining the term "uncovered mutant" as that mutant whose fault is not exercised by a test suite. These uncovered mutants should not been computed when assessing the results. Likewise, the authors of that paper also notice the possibility of executing some duplicated mutants when the fault is inserted in a class which is reused in more than a single file. Therefore, either the creation of duplicated mutants is handled in order to avoid them in the generation stage, or they are manually omitted in the execution of the tests.

Finally, the identification of equivalent mutants is still an undecided problem, so the alive mutants are visually analyzed to determine which of them are actually equivalent. This is a harsh and tedious task and, in several situations, it is not easy to assure whether a mutant is definitely equivalent or not. This matter can be even more pronounced in the case of C++ because of its features. According to the great size of current

 ISBN: 978-1-61208-346-9

programs leading to a high number of mutants, the analysis of each one of them can be rather costly in time. Because of this, Segura et al. [21] creates a new classification of mutants: "undecided". Within this group are cataloged those mutants whose study exceeds an established threshold of time without reaching a final conclusion. Anyway, the implementation of a technique to reduce the time execution or the number of mutants should be considered.

V. CONCLUSION

This paper has supplied a comprehensive survey of the state of mutation testing with regard to the C++ programming language, which has not been almost tackled in the literature and neither in the practice so far. The paper provides useful information about the mutation operators and various research fields where further investigation should be accomplished before conceiving the creation of a mutation tool for this language: the implementation technique to inject the syntactic faults in the program, the execution of tests and the evaluation of the empirical results.

With respect to all these matters, we can summarize that a set of operators can be composed at different levels of the language, being possible to adapt the operators from other similar languages. The class level regarding OO features is probably the area most interesting; this paradigm is widely used in C++ and further research is needed at this level to obtain more concluding results around class mutation operators. Likewise, we consider promising the approach of reusing the abstract syntax tree generated by a compiler because it ensures a complete coverage of the features of the language.

In the future, we intend to start out the development of a mutation tool, first composing a set of class mutation operators. In addition, we aim to study the possibilities of the abstract syntax tree to found the mutation locations as well as appropriately mutate the code. Once the mutants can be created, a complete and automated tool can be constructed to perform some experiments to evaluate the operator quality and the usefulness of the technique in C++. Another salient issue is the high number of mutants that can be produced, so the usage of a cost reduction technique may direct the future research.

VI. ACKNOWLEDGMENTS

This paper was partially funded by the research scholarship PU-EPIF-FPI-PPI-BC 2012-037 of the University of Cádiz and by the MoDSOA research project (TIN2011-27242) under the National Program for Research, Development and Innovation of the Ministry of Science and Innovation (Spain).

REFERENCES

[1] M. R. Woodward, "Mutation testing - its origin and evolution," Information and Software Technology, vol. 35, no. 3, Mar. 1993, pp. 163–169. [Online]. Available: http://dx.doi.org/10.1016/0950-5849(93)90053-6

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," Software Engineering, IEEE Transactions on, vol. 37, no. 5, Oct. 2011, pp. 649 –678.

[3] Z. Ahmed, M. Zahoor, and I. Younas, "Mutation operators for object-oriented systems: A survey," in Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on, vol. 2, feb. 2010, pp. 614 –618.

[4] H. Zhang, "Mutation operators for C++," retrieved: April, 2014. [Online]. Available: http://people.cis.ksu.edu/~hzh8888/mse_project/

[5] A. Derezińska, "Object-oriented mutation to assess the quality of tests," in Proceedings of the 29th Conference on EUROMICRO. Belek, Turkey: IEEE Computer Society, 2003, pp. 417–420.

[6] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at the multi-class and system levels," Science of Computer Programming, vol. 78, no. 4, 2013, pp. 364–387, special section on Mutation Testing and Analysis (Mutation 2010) and Special section on the Programming Languages track at the 25th ACM Symposium on Applied Computing.

[7] J. Boubeta-Puig, A. García-Domínguez, and I. Medina-Bulo, "Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages," in Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE. Berlin, Germany: IEEE, 2011, p. 398–407, print ISBN: 978-1-4577-0019-4. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.52

[8] K. N. King and A. J. Offutt, "A FORTRAN language system for mutation-based software testing," Software - Practice and Experience, vol. 21, no. 7, 1991, pp. 685–718.

[9] Y. S. Ma, J. Offutt, and Y. Kwon, "MuJava: an automated class mutation system," Software Testing, Verification and Reliability, vol. 15, no. 2, 2005, pp. 97–133.

[10] Y. Jia and M. Harman, "MILU: a customizable, Runtime-Optimized higher order mutation testing tool for the full c language," in Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference, Aug. 2008, pp. 94 –98.

[11] "Insure++: C/C++ testing tool, detect elusive runtime memory errors - Parasoft," retrieved: April, 2014. [Online]. Available: http://www.parasoft.com/insure

[12] "PlexTest ITRegister," retrieved: April, 2014. [Online]. Available: http://www.itregister.com.au/products/plextest

[13] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research," in Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007, Sep. 2007, pp. 203 –209.

[14] A. Derezinska and K. Kowalski, "Object-oriented mutation applied in common intermediate language programs originated from C#," in Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, 2011, pp. 342–350.

[15] S. B. Lippman, J. LaJoie, and B. E. Moo, C++ Primer, Fifth Edition, 5th ed. Addison-Wesley, 2013.

[16] H. Agrawal and et al., "Design of mutant operators for the C programming language," Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, Tech. Rep., Mar. 1989.

[17] J. Offutt, Y. S. Ma, and Y. R. Kwon, "The class-level mutants of MuJava," in Proceedings of the 2006 International Workshop on Automation of Software Test, K. Anderson, Ed. Shanghai (China): ACM, May 2006, pp. 78–84.

[18] A. Derezińska, "Quality assessment of mutation operators dedicated for C# programs," in Proceedings of VI International Conference on Quality Software, P. Kellenberger, Ed. Beijing (China): IEEE Computer Society, Oct. 2006, pp. 227–234, ISSN 1550-6002.

[19] B. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," Empirical Software Engineering, vol. 14, no. 3, 2009, pp. 341–369. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9083-7

[20] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative evaluation of mutation operators for ws-bpel compositions," in Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, 2010, pp. 142–150.

[21] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés, "Mutation testing on an object-oriented framework: An experience report," Information and Software Technology, vol. 53, no. 10, 2011, pp. 1124 – 1136, special Section on Mutation Testing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584911000826

 ISBN: 978-1-61208-346-9