MutaLog: a Tool for Mutating Logic Formulas

Christopher Henard, Mike Papadakis, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
Luxembourg, Luxembourg
Email: {christopher.henard, michail.papadakis, yves.letraon}@uni.lu

***Abstract*—Assessing the quality of a test suite is an important step of the testing process. Indeed, it is necessary to ensure that the different test cases target all the critical parts of the system. Model-based testing is a famous technique to perform testing. It uses a model of the system under test. Most of these models include logic formulas. Such formulas encompasses constraints to be satisfied within a system, e.g., an expected behavior or particular conditions to be fulfilled at a given stage of the execution. One way to evaluate the quality of a test suite with respect to these logic constraints is to use mutation analysis. This technique has been proven to be effective for evaluating the quality of a test suite in both model-based and non-model-based testing. However, while many mutation analysis tools exist, none of them performs on logic formulas. Towards this direction, this paper introduces MutaLog, an open source tool which allows performing mutation analysis on logic formulas.**

I. Introduction

Testing a system is important to ensure both its correctness and reliability [1]. In order to provide a high confidence in the testing process, it is important to make certain that the quality of the test suite is high. Getting the expected output with a test suite targeting only the trivial part of the system is pointless. As a result, it is necessary to guarantee that the different test cases are targeting all the critical parts of the system.

Model-based testing is a famous technique to perform testing [2]. To this end, it uses a model of the system under test from which test cases are derived. Most of the software models include logic formulas. Such formulas encompasses constraints to be satisfied within a system, e.g., an expected behavior or particular conditions to be fulfilled at a given stage of the execution. For instance, the activation of a particular module in the system requires another module to be enabled.

One way to evaluate whether such model constraints are satisfied during the testing process is to use mutation analysis [3], [4], [5]. Mutation analysis aims at evaluating the quality of test suites. It is generally applied on programs and it involves the modification of the original software artifact into altered versions. Each of this altered version is called a mutant. Each mutant encompasses a defect willingly introduced. The test suite is then evaluated on these mutants to establish whether or not they are able to reveal the introduced defects. With this respect, the artifact on which we apply mutation analysis is a logic formula. Thus, a mutant is an altered version of this formula.

Mutation analysis has been applied to various models, e.g. Feature Models [6], Finite State Machines [7], Petri nets [8] or security policies metamodels [9]. While tools exist to perform mutation analysis on programs and software models, there is no tool available to the authors knowledge to perform mutation analysis of logic formulas. Here, it should be noted that the scope of the tool is bounded to formulas derived from models. This means that the formulas represent constraints that must always hold. In this respect, the present paper introduces MutaLog, an open source tool which allows performing mutation analysis on logic formulas.

The remainder of this paper is organized as follows. Section II shows the adoption of MutaLog. Section III presents the concepts underlying the approach performed by the tool and Section IV describes the approach itself. Section V introduces the functionalities, architecture and usage of the tool. Finally, Section VI discusses related work before Section VII concludes the paper.

II. Adoption of MutaLog

MutaLog has been adopted in several work. In [10], it is used to transform the logic formula representing a model for software product line in order to produce a more accurate model of the system. The transformation are made on the logic constraints of the model. Another application of the tool aims at creating mutants of a logic formula to assess the quality of software product line test suites [6]. The results of this work show that the mutants employed by the tool are effective to characterize a test suite. In particular, the diversity of the test suite is linked with its ability to detect mutants.

Recently, it has been shown that measuring the number of model-based defects gives a stronger correlation to code-level faults than measuring the number of the exercised t-wise interactions in the context of Combinatorial Interaction Testing [11]. The defects introduced in the models are alterations of logic constraints, such as the mutation operators implemented by the tool.

III. Background

A. Logic Formulas

In this paper, we use boolean logic formulas expressed in Conjunctive Normal Form (CNF). Such formulas are a conjunction of n clauses $C_1, ..., C_n$, where a clause is a disjunction of m literals. A literal is a variable or its negation. Thus, a CNF formula ϕ is of the form:

$$\phi = \bigwedge_{i=1}^{n} \underbrace{\left(\bigvee_{j=1}^{m} l_j \right)}_{\text{clause}}, \text{ where } l_j \text{ is a literal.}$$

For instance, the following boolean CNF formula $\psi = (a \vee b) \wedge c \wedge (b \vee \neg c)$ has 3 variables: a, b and c, and 3 clauses: $(a \vee b)$, c and $(b \vee \neg c)$.

B. Test Case and Test suite

A test case is an assignment of the variables of a given formula. For instance, $tc_1 = \{a = false, b = true, c = true\}$ is a test case for the above-mentioned formula ψ.

We say that a test case satisfies a formula if the formula is evaluated to $true$ with respect to the assignment of the test case. For instance, evaluating the formula ψ with the assignment of tc_1 leads to $\psi = (false \vee true) \wedge true \wedge (true \vee false) = true$. As a result, tc_1 satisfies ψ. On the contrary, $tc_2 = \{a = true, b = true, c = false\}$ does not satisfy the formula ψ since it is evaluated to $\psi = (true \vee true) \wedge false \wedge (true \vee true) = false$.

Similarly, we say that a test suite satisfies a formula ψ if all the test cases of the test suite satisfies ψ. A test suite does not satisfy a formula ψ if at least one test case of this test suite does not satisfy ψ. Since the formulas used by the tool are dervied from models, the formula under test must be evaluated to $true$.

C. Invalid Formula

We say a formula is invalid if there is no test case that can satisfy it. For instance, the formula $\psi = a \wedge \neg a$ is invalid, i.e. can never be $true$. In this work, since logic formulas are derived from models, invalid ones are meaningless.

D. Mutation Analysis

Mutation analysis forms a powerful technique with various applications like software testing [12], [13] and debugging [14]. It is applied by producing altered (mutant) versions of the programs artifacts like source code, specification models, etc. [12], [13]. The underlying idea of this approach is to evaluate the ability of test cases to reveal behavior differences between the original (unaltered) and the mutated (altered) artifact versions.

The mutated versions represent possible defects of the artifact under test and they are produced based on a set of well defined rules called mutant operators [13]. Mutant operators are defined on "syntactic descriptions to make syntactic changes to the syntax or objects developed from the syntax" [13]. Mutation analysis denotes the process of introducing these mutants. The ability of the test cases to reveal the introduced mutants is examined in order to use this approach for testing purposes (mutation testing). If a mutant can be detected by a test case, the mutant is called killed. Otherwise, it is called live. Therefore, measuring the ratio of the killed mutants to the totally introduced ones results in a quality measure of the testing process, called mutation score. This measure demonstrates the ability of the tests to detect errors:

$$\text{Mutation score} = \frac{\text{mutants killed}}{\text{total mutants - invalid mutants}}.$$

It should be noted that invalid formulas are not taken into account since formulas are derived from models. In the context of this paper, mutants are produced by applying a set of mutant operators to the logic formula. The test case evaluation is performed by checking whether the test cases satisfy the formula, i.e. whether the formula is evaluated to $true$. Consequently, a mutant is said to be killed if its formula is not satisfied, i.e. if the formula is evaluated to $false$.

IV. Approach

The mutation analysis approach for logic formulas performed by the tool operates as follows. First, mutants of the considered logic formulas are created by applying different operators. Then, it is evaluated which mutants are killed by the test cases of the test suite. Finally, the mutation score is returned.

In this section, we will consider the following logic formula:

$$\psi = (a \vee b) \wedge (\neg a \vee c)$$

and the following test suite:

$$T = \{tc_1, tc_2\}, \text{ where}$$
$$tc_1 = \{a = true, b = true, c = true\} \text{ and}$$
$$tc_2 = \{a = true, b = false, c = true\}$$

as a running example.

A. Creation of Mutants of the Logic Formula

The first step of the approach creates mutants of the logic formula by applying different mutation operators on it. The

Table I
Mutant Operators for CNF Logic Formulas

Mutation Operator	Action
Literal Omission (LO)	A literal is removed
Literal Negation (LN)	A literal is negated
Clause Omission (CO)	A clause is removed
Clause Negation (CN)	A clause is negated
AND Reference (AR)	An AND operator is replaced by OR
OR Reference (OR)	An OR operator is replaced by AND

Table II
APPLICATION OF THE MUTANT OPERATORS TO THE EXAMPLE FORMULA $\psi = (a \vee b) \wedge (\neg a \vee c)$. THE UNDERLINED MUTANTS ARE THOSED KILLED BY THE TEST SUITE $T = \{tc_1, tc_2\}$, WHERE $tc_1 = \{a = true, b = true, c = true\}$ AND $tc_2 = \{a = true, b = false, c = true\}$

Literal Omission	Literal Negation	Clause Omission	Clause Negation	AND Reference	OR Reference
$\underline{LO_1 = b \wedge (\neg a \vee c)}$	$\underline{LN_1 = (\neg a \vee b) \wedge (\neg a \vee c)}$	$CO_1 = (\neg a \vee c)$	$\underline{CN_1 = \neg a \wedge \neg b \wedge (\neg a \vee c)}$	$AR_1 = (a \vee b \vee \neg a \vee c)$	$\underline{OR_1 = a \wedge b \wedge (\neg a \vee c)}$
$LO_2 = a \wedge (\neg a \vee c)$	$LN_2 = (a \vee \neg b) \wedge (\neg a \vee c)$	$\underline{CO_2 = (a \vee b)}$	$\underline{CN_2 = (a \vee b) \wedge a \wedge \neg c}$		$\underline{OR_2 = (a \vee b) \wedge \neg a \wedge c}$
$LO_3 = (a \vee b) \wedge c$	$LN_3 = (a \vee b) \wedge (a \vee c)$				
$\underline{LO_4 = (a \vee b) \wedge \neg a}$	$\underline{LN_4 = (a \vee b) \wedge (\neg a \vee \neg c)}$				

6 mutation operators implemented by the tool are taken from [6], [15], [16] and presented in Table I. The first four operators respectively remove and negate one literal or one clause within the formula. The last two operators transforms an AND into an OR and vice-versa.

The application of these operators to the example formula ψ give the mutants listed in Table II. Here, it should be noted that the application of the operators can add or remove clauses within the formula, and that none of the resulting mutant is an invalid formula.

B. *Evaluation of the Mutation Score*

The second step of the approach checks whether for each mutant created there is one test case in the test suite that kills it. To this end, a satisfiability (SAT) solver is used to evaluate the satisfiability of the formula given the variables assignment of the test case. SAT solver are widely used to evaluate constrained boolean expressions, e.g. [17]. As a result, for each mutant, it is evaluated whether there is one test case that does not satisfy the formula. When such a test case exist, the mutant is said to be killed.

Considering the example, the 9 mutants underlined in Table II are killed by the test cases. It means that the two assignments of tc_1 and tc_2 satisfy the other mutants. Thus, the mutation score achieved by this test suite is $\frac{9}{15} = 60\%$.

V. THE MUTALOG TOOL

MutaLog is an open source[1] Java application of around 2,000 lines of code[2] in its current version. It implements the mutation analysis approach for logic formulas presented in the previous section.

A. *Features*

MutaLog allows performing the following actions:

- Loading a logic formula from a file. MutaLog supports the DIMACS (Conjunctive Normal Form) format,
- Visualizing the clauses of the formula,
- Creating mutants of the formula by applying the operators presented in the previous section,
- Loading a test suite and visualize the test cases,
- Evaluating the mutation score of the test suite towards the formula mutants.

[1]The code source is available at `https://github.com/christopherhenard/mutalog`.

[2]Measured with `cloc` `http://cloc.sourceforge.net/`.

B. *Architecture*

1) General: MutaLog implements the Model-View-Controller (MVC) architecture [18]. MVC allows separating the internal representation of the information and its logic (the model) from the graphical representation (the view) and the user's interaction with it (the controller(s)). Thus, the model, which corresponds to the business logic of the application is observed by the view, which graphically represents the model. When the user performs an action on the view, a controller acts on the model, thus changing its internal representation and updating the view. The main advantage of MVC is the separation of concerns and the code reusability.

To implement the MVC architecture, the application is organized into packages. Thus, the Java classes are grouped into a conceptually coherent way depending on their functionalities. Thus, the `core` package corresponds to the model while the `gui` package contains the classes related to the views and controllers. Figure 1 represents the package architecture of the application. The tool also makes use of common design patterns [19], like or the *adapter* pattern to map the model of complex graphical component to the tool's model or the *observer* pattern to implement the MVC architecture.

2) MutaLog's Main Components: The architecture of MutaLog is presented on Figure 2. MutaLog takes as input the logic formula and the test suite. From the logic formula, the formula mutants are created by the mutant operators. Then, using a satisfiability (SAT) solver, the test suite is evaluated towards the mutants. Finally, the mutation score is returned.

3) The `core` package: This package contains the classes related to the business logic of the application (the model). The class diagram of this package and its sub-packages is

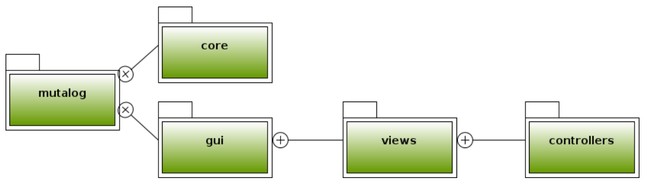


Figure 1. MutaLog's Packages Architecture

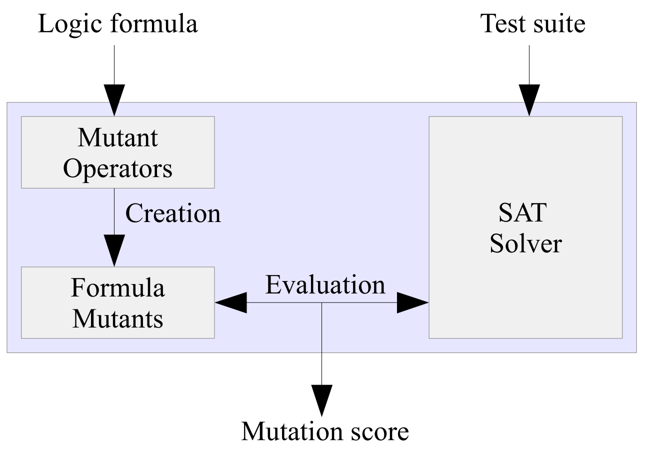


Figure 2. MutaLog's Main Components Architecture

represented on Figure 4. In the following, we describe the role of each class:

- `core.ModelMutaLog`: This class is the model of the application, it contains the main business logic, like the possibility to load a logic formula,
- `core.CNFFormula`: This class represents a logic formula under the CNF format,
- `core.Clause`: This class represents a clause of the CNF formula,
- `core.Literal`: This class represents a literal of a clause,
- `core.Mutant`: This class represents a mutated logic formula,
- `core.TestSuite`: This class represents a test suite for a logic formula,
- `core.Test`: This class represents a given test case of the test suite.

C. Libraries

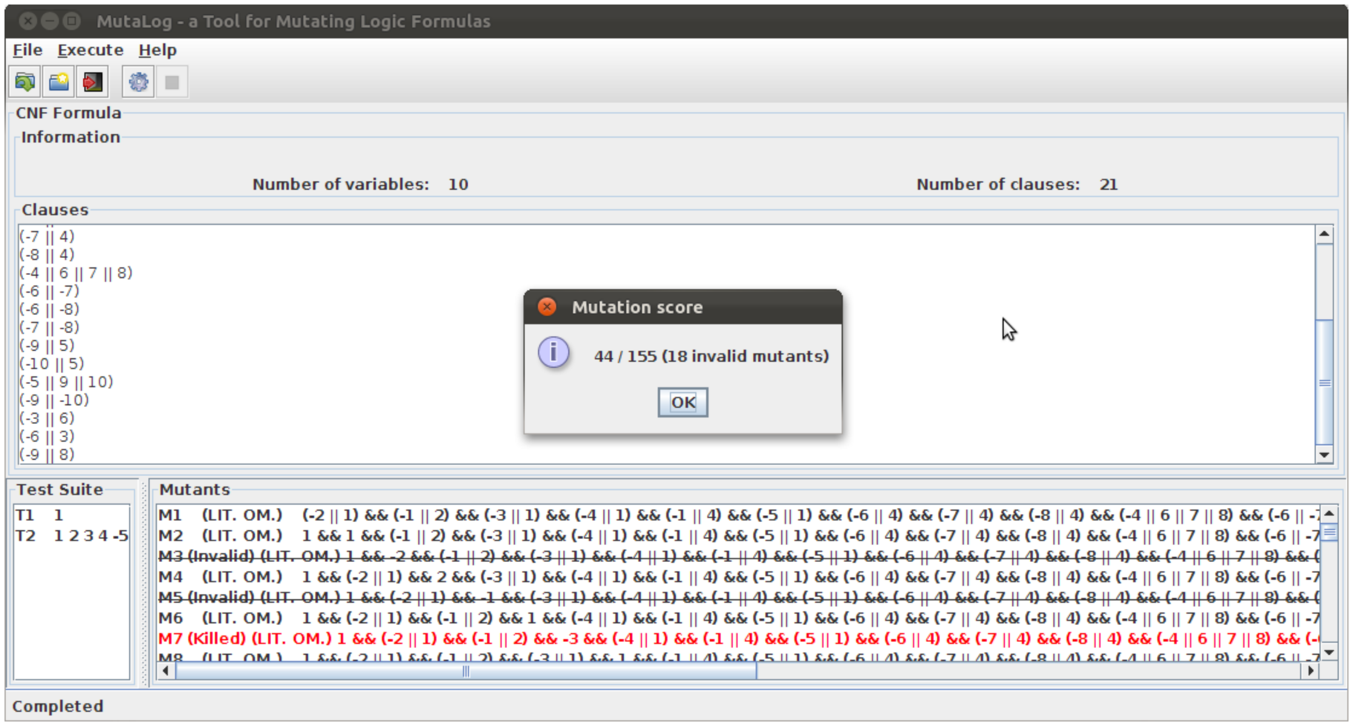
MutaLog uses two external libraries in addition to the standard Java library:

- Sat4j[3], which provides an efficient SAT solver for Java and which is used to evaluate whether an assignment satisfies a formula and to remove invalid mutants,
- JCommander[4], which simplifies the parsing of command line arguments and which is used for the command line interface of the tool.

D. Usage

The MutaLog tool can be used with both a command line interface (CLI) and graphical user interface (GUI). The command line interface eases the use of the tool in a scripting or automated context while the GUI is more user-friendly. Figure 3 provides a screenshot of the GUI. The main window of the application contains a tool bar which provides a quick access to the main functions of the tool.

[3]`http://www.sat4j.org/`
[4]`http://jcommander.org/`



Number of variables: 10   Number of clauses: 21

```
(-7 || 4)
(-8 || 4)
(-4 || 6 || 7 || 8)
(-6 || -7)
(-6 || -8)
(-7 || -8)
(-9 || 5)
(-10 || 5)
(-5 || 9 || 10)
(-9 || -10)
(-3 || 6)
(-6 || 3)
(-9 || 8)
```

44 / 155 (18 invalid mutants)

```
T1   1
T2   1 2 3 4 -5
```

M1 (LIT. OM.) (-2 || 1) && (-1 || 2) && (-3 || 1) && (-4 || 1) && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (-6 || -7
M2 (LIT. OM.) 1 && 1 && (-1 || 2) && (-3 || 1) && (-4 || 1) && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (-6 || -7
~~M3 (Invalid) (LIT. OM.) 1 && -2 && (-1 || 2) && (-3 || 1) && (-4 || 1) && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (~~
M4 (LIT. OM.) 1 && (-2 || 1) && 2 && (-3 || 1) && (-4 || 1) && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (-6 || -7
~~M5 (Invalid) (LIT. OM.) 1 && (-2 || 1) && -1 && (-3 || 1) && (-4 || 1) && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (~~
M6 (LIT. OM.) 1 && (-2 || 1) && (-1 || 2) && 1 && (-4 || 1) && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (-6 || -7
M7 (Killed) (LIT. OM.) 1 && (-2 || 1) && (-1 || 2) && -3 && (-4 || 1) && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (-
M8 (LIT. OM.) 1 && (-2 || 1) && (-1 || 2) && (-3 || 1) && 1 && (-1 || 4) && (-5 || 1) && (-6 || 4) && (-7 || 4) && (-8 || 4) && (-4 || 6 || 7 || 8) && (-6 || -7

Completed

Figure 3. MutaLog's Graphical User Interface

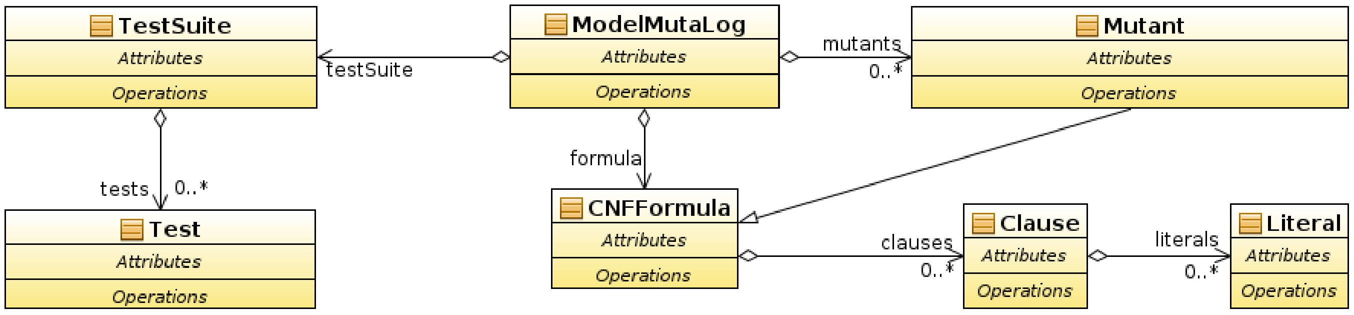


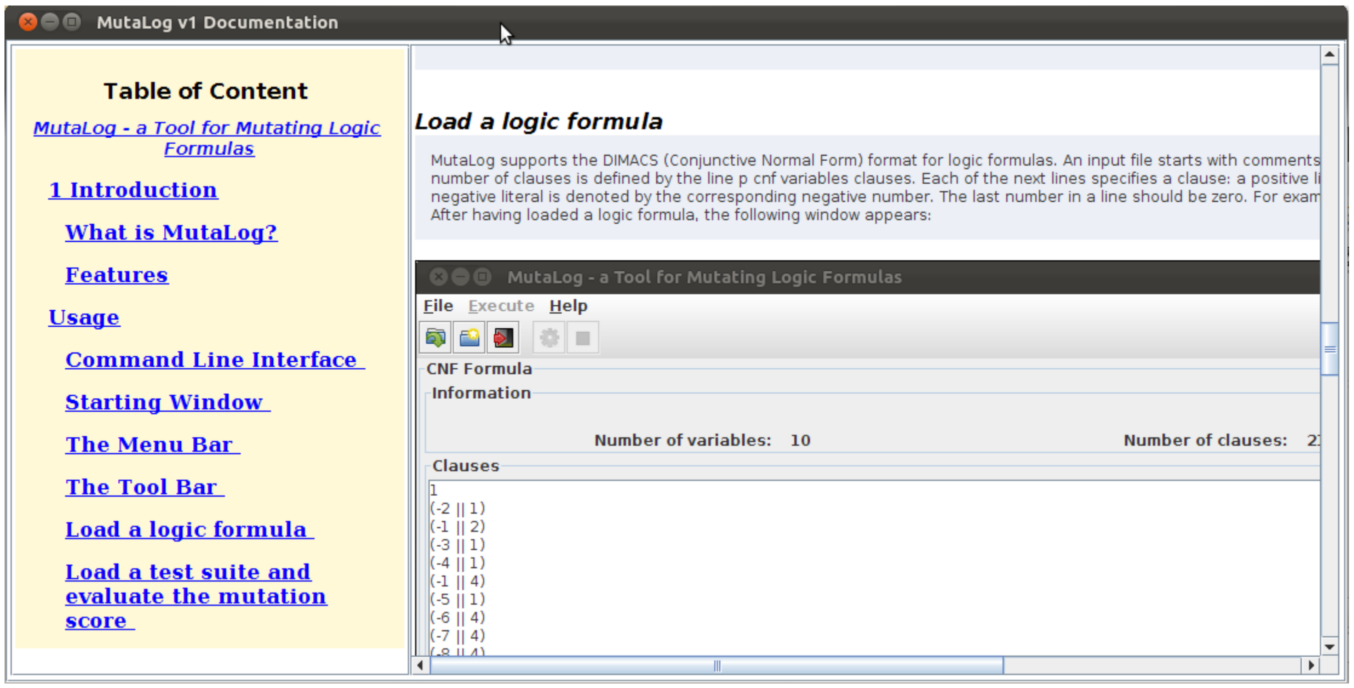
Figure 4. MutaLog's Core Classes

In terms of end use, an HTML user guide (see Figure 5) is embedded in the application to help the user using the tool. In terms of development, the source code and the Javadoc documentation are available. As the tool is open source, it can be used as a library as well. Finally, it is planned to build an Eclipse plugin to integrate the use of the tool within the Eclipse environment.

VI. Related Work

Mutation testing and analysis have been widely applied to test specifications or models [12]. Such models include, Petri nets [8], security policies metamodels [9] or Finite State Machines [7]. In this paper, we focus on logic formulas. Such formulas can represent various models, e.g., Feature Models [6] for Software Product Lines.

Several work on mutation analysis are related to the present one. Andrew *et al.* [3] use mutation analysis to produce mutants and to show that they be used to predict the detection effectiveness of real faults. In this paper, we do not focus on whether or not the generated mutants are representative of real defects. Gargantini and Fraser proposed a method in order to generate tests for possible faults of boolean expressions [20]. Here, we do not focus on test generation. We evaluate a test suite according to altered boolean formulas. Whether the altered versions represent possible faults of boolean expression is not evaluated in this paper. In the context of logic-based testing, Kaminski *et al.* [15], [21] use a logic mutation approach to generate only subsuming higher order logic mutants. This approach aims at designing tests depending on logical expressions. In this paper, mutation operators are applied on a logic formula. The objective is only to evaluate the quality of a test suite according to the mutants of the formula.



MutaLog v1 Documentation

Table of Content

Load a logic formula

MutaLog supports the DIMACS (Conjunctive Normal Form) format for logic formulas. An input file starts with comments
number of clauses is defined by the line p cnf variables clauses. Each of the next lines specifies a clause: a positive li
negative literal is denoted by the corresponding negative number. The last number in a line should be zero. For exam
After having loaded a logic formula, the following window appears:

MutaLog - a Tool for Mutating Logic Formulas

File Execute Help

CNF Formula

Information

Number of variables: 10 Number of clauses: 2

Clauses

```
1
(-2 || 1)
(-1 || 2)
(-3 || 1)
(-4 || 1)
(-1 || 4)
(-5 || 1)
(-6 || 4)
(-7 || 4)
```

Figure 5. MutaLog's Documentation

Finally, regarding mutation analysis tools, several tool exist, such as Proteum/fl [22], μJava [23] or Javalanche [24]. However, all the existing tools perform mutation analysis of programs. To the best of the authors knowledge, there is no tool which performs mutation analysis of logic formulas.

VII. Conclusion and Future Work

In this paper, we introduced an open source and freely distributed research tool, MutaLog, which performs logic mutation. The tool allows loading a logic formula, creating various mutants of the formula and evaluating the mutation score of a given test suite. To the best of the authors knowledge, MutaLog is the first tool available which allows performing mutation analysis of logic formulas.

The version of MutaLog presented in this paper is the first one. In future, MutaLog will be extended with additional features, including:

- Addition of new mutation operators,
- Support of additional formats than CNF for boolean logic,
- Support of additional logics, e.g., first order logic,
- Integration into an Eclipse plugin.

Finally, we invite researchers, students and developers to use our tool and/or to contribute to its development:

`http://research.henard.net/SPL/MutaLog/`.

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed., 2008.

[2] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *WEASELTech*, 2007, pp. 31–36.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[4] R. Geist, A. J. Offutt, and F. C. H. Jr., "Estimation and enhancement of real-time software reliability through mutation analysis," *IEEE Trans. Computers*, vol. 41, no. 5, pp. 550–558, 1992.

[5] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," vol. 54, no. 9, 2012, pp. 915–932.

[6] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Assessing software product line testing via model-based mutation: An application to similarity testing," in *ICST Workshops*, 2013, pp. 188–197.

[7] J.-h. Li, G.-x. Dai, and H.-h. Li, "Mutation analysis for testing finite state machines," in *ISECS*, 2009, pp. 620–624.

[8] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. Wong, "Mutation testing applied to validate specifications based on petri nets," in *ICFDT*, 1996, pp. 329–337.

[9] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti, "Xacmut: Xacml 2.0 mutants generator," in *ICST Workshops*, 2013, pp. 28–33.

[10] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Towards automated testing and fixing of re-engineered feature models," in *ICSE*, 2013, pp. 1245–1248.

[11] M. Papadakis, C. Henard, and Y. L. Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing," *ICST*, 2014.

[12] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649 –678, sept.-oct. 2011.

[13] J. Offutt, "A mutation carol: Past, present and future," *Information & Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[14] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, pp. n/a–n/a, 2013.

[15] G. K. Kaminski, U. Praphamontripong, P. Ammann, and J. Offutt, "A logic mutation approach to selective mutation for programs and queries," *Information & Software Technology*, vol. 53, no. 10, pp. 1137–1152, 2011.

[16] M. F. Lau and Y. T. Yu, "An extended fault class hierarchy for specification-based testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 247–276, Jul. 2005.

[17] P. Arcaini, A. Gargantini, and E. Riccobene, "Optimizing the automatic test generation by sat and smt solving for boolean expressions," in *ASE*, 2011, pp. 388–391.

[18] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988.

[19] L. Rising, *The patterns handbook: Techniques, strategies, and applications*, 1998.

[20] A. Gargantini and G. Fraser, "Generating minimal fault detecting test suites for general boolean specifications," *Information & Software Technology*, vol. 53, no. 11, pp. 1263–1273, 2011.

[21] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[22] M. Papadakis, M. E. Delamaro, and Y. L. Traon, "Proteum/fl: A tool for localizing faults using mutation analysis," in *SCAM*, 2013, pp. 94–99.

[23] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[24] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *ESEC/FSE*, 2009, pp. 297–298.