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 \( C_1, \ldots, C_n \), where a clause is a disjunction of \( m \) literals. A literal is a variable or its negation. Thus, a CNF formula \( \phi \) is of the form:

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

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

B. Test Case and Test suite

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

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 \( \psi \) with the assignment of \( tc_1 \) leads to \( \psi = (\text{false} \lor \text{true}) \land \text{true} \land (\text{true} \lor \text{false}) = \text{true} \). As a result, \( tc_1 \) satisfies \( \psi \). On the contrary, \( tc_2 = \{ a = \text{true}, b = \text{true}, c = \text{false} \} \) does not satisfy the formula \( \psi \) since it is evaluated to \( \psi = (\text{true} \lor \text{true}) \land \text{false} \land (\text{true} \lor \text{true}) = \text{false} \).

Similarly, we say that a test suite satisfies a formula \( \psi \) if all the test cases of the test suite satisfies \( \psi \). A test suite does not satisfy a formula \( \psi \) if at least one test case of this test suite does not satisfy \( \psi \). Since the formulas used by the tool are derived 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 \( \psi = a \land \neg a \) 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, specifications, 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:

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

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:

\[
\psi = (a \lor b) \land (\neg a \lor c)
\]
and the following test suite:

\[
T = \{ tc_1, tc_2 \}, \text{ where }
\]

\[
tc_1 = \{ a = \text{true}, b = \text{true}, c = \text{true} \} \text{ and } \]
\[
tc_2 = \{ a = \text{true}, b = \text{false}, c = \text{true} \}
\]
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>
<thead>
<tr>
<th>Mutation Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal Omission (LO)</td>
<td>A literal is removed</td>
</tr>
<tr>
<td>Literal Negation (LN)</td>
<td>A literal is negated</td>
</tr>
<tr>
<td>Clause Omission (CO)</td>
<td>A clause is removed</td>
</tr>
<tr>
<td>Clause Negation (CN)</td>
<td>A clause is negated</td>
</tr>
<tr>
<td>AND Reference (AR)</td>
<td>An AND operator is replaced by OR</td>
</tr>
<tr>
<td>OR Reference (OR)</td>
<td>An OR operator is replaced by AND</td>
</tr>
</tbody>
</table>

Table I

MUTANT OPERATORS FOR CNF LOGIC FORMULAS
A. Features

the previous section.

mutation analysis approach for logic formulas presented in

2,000 lines of code

assignments of

Table II are killed by the test cases. It means that the two

test case that does not satisfy the formula. When such a test

result, for each mutant, it is evaluated whether there is one

assignment of the test case. SAT solver are widely used

evaluate the satisfiability of the formula given the variables

mutant created there is one test case in the test suite that

is an invalid formula.

clause within the formula. The last two operators transforms

operators respectively remove and negate one literal or one

from \[6\], \[15\], \[16\] and presented in Table I. The first four

6 mutation operators implemented by the tool are taken

from \[6\], \[15\], \[16\] and presented in Table II. The first four

operators respectively remove and negate one literal or one

clause within the formula. Here, it should be

noted that the application of the operators can add or remove

clauses within the formula. The last two operators transforms

an AND into an OR and vice-versa.

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 oper-

  ators 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 func-

tionalities. 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) Mutalogs’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

\[\begin{align*}
\text{model} & \rightarrow \text{logbook} \\
\text{gui} & \rightarrow \text{observers} \\
\text{controllers} & \rightarrow \text{views} \\
\text{mutalog} & \rightarrow \text{core}
\end{align*}\]

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.

http://www.sat4j.org/
http://jcommander.org/

Figure 3. MutaLog’s Graphical User Interface
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.

Table of Content

1 Introduction
   What is MutaLog?
   Features
Usage
   Command Line Interface
   Starting Window
   The Menu Bar
   The Tool Bar
   Load a logic formula
   Load a test suite and evaluate the mutation score

Figure 5. MutaLog’s Documentation
Finally, regarding mutation analysis tools, several tools 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