SMutant: A Tool for Type-Sensitive Mutation Testing in a Dynamic Language

Milos Gligoric Sandro Badame Ralph Johnson
Department of Computer Science
University of Illinois, Urbana, IL 61801, USA
{gliga, badame1, rjohnson}@illinois.edu

ABSTRACT

A mutation testing tool takes as input a system under test and a test suite and produces as output the mutation score of the test suite. The tool systematically creates mutants by making small syntactic changes to the system under test and executes the test suite to determine which mutants give different results from the original system. Almost all mutation testing tools have been developed for statically typed languages. The lack of tools for dynamically typed languages may be rooted in additional challenges that are caused by the lack of precise type information until the program is executed. Existing tools for dynamically typed languages mostly focus on mutation of literals because the type of literals are known statically.

This paper presents SMutant, the first mutation testing tool for Smalltalk programs. In addition to literal replacement, SMutant supports many mutation operators that are commonly seen in tools for statically typed languages, such as operator replacement. Instead of applying mutations statically, SMutant postpones mutating until execution and applies mutations dynamically, when the types are available. Also, SMutant enables the user to define new mutation operators by sending a single message. The tool automatically generates code to support new mutation operators.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools*

General Terms: Reliability

Keywords: Mutation testing, dynamic languages

1. INTRODUCTION

Mutation testing [1, 3, 4] is a method for measuring the quality of a test suite. A mutation testing tool takes as input a *system under test* (SUT) and a *test suite*, then proceeds in two phases: (1) systematically creates *mutants* in the SUT and (2) executes the test suite to check how many mutants are *killed*; a mutant is killed if the result of the mutated code differs from the result that is obtained by running the original SUT, otherwise the mutant is *live*. In the first phase, mutants are created by applying *mutation operators* that perform small syntactic changes in the SUT. For example, a mutation operator can be defined to replace + with -. In the second phase, a tool tracks killed mutants. The ratio of the killed mutants to all mutants, known as *mutation score*, is used to measure the effectiveness of a test suite.

Mutation testing for statically typed languages has been actively researched since its introduction [3, 4]. The main focus was on measuring the effectiveness of mutation operators. In particular, mutation operators have been proposed and evaluated for a number of languages including Ada, C, Cobol, C#, Fortran, Java, and SQL. A few of the mutation testing tools that have been developed are Mothra (Fortran), Javalanche (Java), and CREAM (C#). These tools perform mutations *statically*, since type information is known before a program is executed. The survey report by Jia and Harman [7] gives an overview of mutation testing.

In contrast, mutation testing for dynamically typed languages has been little explored. The key reason for this may lay in the nature of these languages – *type information is not available until a program is executed.* Therefore, existing tools for dynamically typed languages ignore mutation operators that require type information and are commonly used for statically typed languages. For instance, the same operator + can be used for string concatenation and number addition, but there is no - for strings; thus, an expression v1 + v2 cannot be replaced with v1 - v2 until we know that v1 and v2 are numbers rather than strings. For example, a mutation testing tool for Python, Pester [9], focuses on literal replacement. Similarly, a tool for Ruby, Heckle [5], focuses on literal replacement and expression replacement (e.g., replace "while" with "unless"). In both cases no type information is used. Recently [2], there has been work on defining mutation operators for JavaScript and proposal that mutations should be applied *dynamically*, during the execution of a program. However, no tool was developed.

This paper presents our tool, SMutant, for mutation testing of Smalltalk programs. In addition to replacement of literals, the tool supports many mutation operators commonly used for statically typed languages [7], such as arithmetic operator replacement. Instead of applying mutations statically, SMutant waits until the type information is available (at runtime) and *dynamically* applies mutations [2]. To the best of our knowledge, SMutant is the first tool that performs this kind of mutation testing. Another novelty that comes with SMutant is a support for defining new mutation

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

operators by sending a single message[1] to SMutant. The tool *automatically* generates the code that is needed to support new mutation operators as described in detail in Section 3.3.

The design goals for SMutant were: (1) support for mutation operators for dynamic languages that are common for statically typed languages; (2) support for extending the set of predefined mutation operators without knowing details of the tool; (3) no modifications are required to the base Smalltalk image; and (4) making the tool practical by implementing known techniques from other tools (e.g. running mutation in a separate thread to avoid infinite looping).

SMutant is publicly available at the most popular site for Smalltalk: `http://www.squeaksource.com/smutant/`.

2. SMALLTALK BACKGROUND

Before we discuss SMutant in more detail, we provide a brief, high-level overview of the characteristics of Smalltalk that are relevant for the following sections.

Smalltalk is an object-oriented, dynamically typed, reflective programming language. As opposed to most programming systems, Smalltalk does not separate application data (objects) and code (classes). Smalltalk systems store objects and classes in an *image*, that can be loaded by the virtual machine to restore an earlier state. Therefore, Smalltalk can be seen as a "living" system, which can be extended at run time, saved, and loaded at later point.

Being a "pure" object-oriented language, everything – including code – is an object in Smalltalk. Querying or modifying a state of an object is performed by sending messages that can be seen as being equivalent to a method invocation in Java. Unlike many other languages, Smalltalk does not have primitive types or predefined messages; for example, + is only the name of a message and it can be defined for any object in the system. Any message can be sent to any object. When a message is sent, the system checks if a message is defined in the receiving object; if the message is not defined, the `doesNotUnderstand` message is invoked by the system, which throws an exception by default.

Commonly, code is written as a body of a message. However, the Smalltalk image provides a *Workspace* that can be used for evaluating snippets of code. A Workspace is similar to a "Read-eval-print loop" available for many languages (e.g., Python). Note that code evaluated in Workspace can affect the state of the system.

A typical Smalltalk image comes with many frameworks and tools. One of the available frameworks is *SUnit*, a unit testing framework. SUnit allows for writing tests and assertions in Smalltalk. It has served as a basis for all xUnit tools, including JUnit. As in JUnit (version 3), SUnit requires (1) a class, which extends `TestCase`, that defines test cases; and (2) each test case in a separate method, whose name starts with "test". Optionally, the user can define `setUp` and `tearDown` methods that are executed before and after each test case, respectively. Just as with many other frameworks, SUnit is accompanied with a GUI, called TestRunner, that enables the user to easily select and execute test cases. The following sections describe similarities (Section 3.2) and the dependency between SMutant (MutationTestRunner) and SUnit (TestRunner) (Section 3.7).

[1]Sending a message in Smalltalk can be seen as equivalent to a method invocation in Java (see Section 2).

```
MatrixMutationScenario>>classesToMutate
    ^ { Matrix }
MatrixMutationScenario>>classesWithTests
    ^ { MatrixTest }
MatrixMutationScenario>>mutationFactories
    ^ {RetroReplacementOperatorFactory singleton
                    original: #+ replacement: #-}
MatrixMutationScenario>>isCopy
    ^ false
```

Figure 1: Mutation scenario for `MatrixTest` class.

3. SMUTANT DETAILS

We next describe some features, design decisions, and implementation of SMutant in more detail.

3.1 Mutation Scenario

A *mutation scenario* is used to specify (1) which classes belong to the SUT (classes to be mutated) and (2) which classes belong to the test suite (classes for which mutation score is measured). To define a mutation scenario, the user has to extend the `MutationScenario` class and to override two methods to specify (1) and (2). As an example, suppose that the goal is to measure the mutation score of `MatrixTest`, which is a test suite for the `Matrix` class. (Both classes are available in the Pharo Smalltalk image [10].) The `MatrixMutationScenario` class, which is shown in Figure 1, specifies classes that belong to a SUT by overriding `classesToMutate` and classes that belong to a test suite by overriding `classesWithTests`. The user can also specify which mutation operators should be used by overriding `mutationFactories` (default implementation includes all implemented operators). The method `isCopy` is described in Section 3.6.

Note that SMutant has requirements similar to those of SUnit, which are described in Section 2. However, defining a mutation scenario is much simpler, since the user only has to select proper classes to mutate and the mutation operators from the appropriate/available sets.

3.2 GUI Design Inspired by TestRunner

As mentioned, SUnit includes TestRunner, a GUI that can be used to select test suites to run and obtain details about the results. Similarly, SMutant includes MutationTestRunner, a GUI that can be used to select mutation scenarios to run and obtain details about the results. Figure 2 shows MutationTestRunner. The left-most pane lists all the categories that include mutation scenarios. When a particular category is selected, appropriate mutation scenarios appear in the middle pane (e.g., `MatrixMutationScenario`). The status bar (top-right pane) shows number of all mutants, number of killed mutants, number of live mutants, and mutation score. These numbers are updated dynamically as mutation testing is in progress (similar to TestRunner). The red color of the status bar indicates that there are live mutants, which is analogous to the red color in TestRunner when some tests do not pass. A green color is used when all mutants are killed. If some mutants are alive, the middle right-most pane lists the live mutants with the location where the mutant was created (including class and method name). The right-most pane at the bottom can be used to specify mutation factories for the selected mutation scenarios (Section 3.8). Finally, the button at the bottom can be used to execute the selected mutation scenarios.

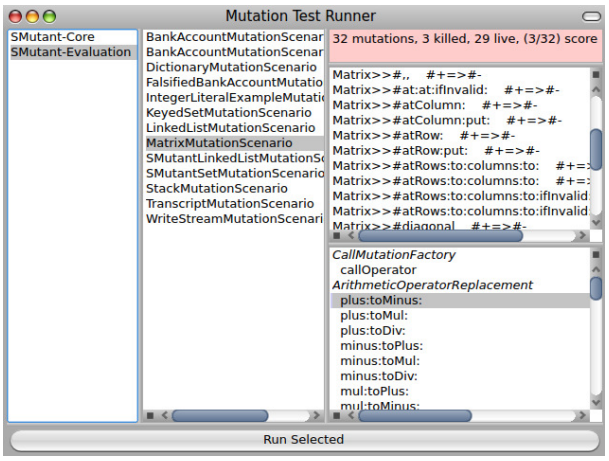

Figure 2: MutationTestRunner can be used to select mutation scenarios to run and show details about the results. The design closely mimics the design of TestRunner.

3.3 Applying Mutations Dynamically

When a mutation scenario is run, by either using MutationTestRunner or invoking the appropriate method from SMutant, the tool proceeds with the three steps.

Analysis. The tool analyzes the code under test (i.e., list of classes that are returned by the `classesToMutate` method) to find potential places to create mutants (e.g., + symbols, if the mutation operator "arithmetic replacement operator: + to - if arguments are Numbers" is selected). It is important to note that a set of mutants is created statically. Namely, this phase is performed before any code is executed. In other words, *a set of mutants does not depend on the test suite and the types that are used in test cases.* Unlike mutants in statically typed languages, each mutant is surrounded with proper type checks. This approach ensures consistent calculation of mutation score to those reported by the tools for statically typed languages. As in existing tools, the test suite does not have to be executed for the mutants that would be created at the locations that are not executed when running the test suite with the original SUT.

Insert type checks. SMutant integrates additional checks at the places of potential mutants to ensure that the arguments of messages are of appropriate type. In this phase, rather than copying the class that is mutated, we use powerful Smalltalk reflection mechanism and replace only mutated method. The changes are automatically reverted when the mutation testing is done, so the user neither has to worry about implementation details nor about saving/copying/restoring the code.

Execution. For each potential mutant, identified in the first phase, the test suite is executed and mutation is *applied* only if one of the inserted checks is satisfied (e.g., invokes - instead of + if both arguments are Numbers). Note that the same message (e.g., +) at the same place in code can be sent with arguments of different types during the same execution. The tool applies mutation only (and always) when types are appropriate. This approach closely mimics mutation testing for statically typed languages; the mutant is created at a specific location in the code and is applied whenever the mutated code is executed.

An alternative to dynamic mutation testing would be mutation testing with support of a type inferencer. The first step would include type inference of the SUT. In the second step, mutants would be created in the same fashion as for statically typed languages wherever the type of arguments is unique (e.g., it is always Number).

3.4 Is Type Checking Necessary?

Do we need to surround mutants with type checks as described in the previous section? In some cases, we do not. Recall from Section 2 that any message can be sent to any object. When the message is inappropriate for the receiver object, the system sends `doesNotUnderstand` message to the same object, which throws an exception by default. If a mutant leads to `doesNotUnderstand`, it can be killed either because of the exception or likely incorrect result if `doesNotUnderstand` is overridden. However, this approach is not applicable in general. First, granularity of mutation operators is lost. Namely, it would be impossible to define: "arithmetic replacement operator: + to - if arguments are Numbers". Therefore, the analogy with mutation testing tools for statically typed languages would be lost. Which granularity is actually necessary remains to be seen, and SMutant enables performing the experiment. Second, although type checking may not be needed for some mutation operators, for others it is mandatory. For example, "logical negation of boolean variable" [2] requires type information.

3.5 User-Defined Mutation Operators

SMutant supports many mutation operators that are commonly implemented in the tools for statically typed languages, such as literal replacement, arithmetic operator replacement, or relational operator replacement. While implementing SMutant, the following became obvious: (1) some traditional mutation operators can be merged into a single mutation operator in Smalltalk and (2) defining replacement operators follows a pattern that is easy to automate.

Some traditional mutation operators can be merged in one mutation operator in Smalltalk because operators such as + or -, which have a special treatment in most programming languages, are treated the same as other messages. In other words, + or - are just names of the messages and can be defined for any type. This fact makes "arithmetic replacement operators" and "method replacement" the same mutation operators in Smalltalk. We implemented only one (generic) mutation operator for this case. Additional analysis of traditional mutation operators is needed to detect other similar cases that may hold in Smalltalk.

Implementation of replacement mutation operators, such as arithmetic and logical mutation operators, follows the same pattern. Since the number of operators (e.g., arithmetic) is limited in most statically typed languages, the replacement mutation operators are commonly written manually. Defining replacement mutation operators in Smalltalk is somewhat different, because one message (including what other languages commonly consider to be operators) can be replaced with any other message (provided that they have the same number of arguments). Therefore, defining all possible replacement mutation operators is not practical due to the number of possible replacements. SMutant includes standard sets of replacement mutation operators (e.g., arith-

```
defineUserOperator := DefineUserOperator new.
defineUserOperator original: #+.
defineUserOperator replacement: #-.
defineUserOperator runtime: #plusToMinus.
defineUserOperator firstType: Number.
defineUserOperator secondType: Number.
defineUserOperator define.
```

Figure 3: Code to evaluate in Workspace in order to define new mutation operator.

metic replacement operators) and includes support for user-defined mutation operators. User can decide, based on the application, which mutation operators to define. User only has to specify the name of the original message (i.e., the message to be replaced), replacement message, and type of the arguments that must hold to enable application of mutation. The code in Figure 3 can be used to define "arithmetic replacement operator + to - when arguments are Numbers".

It is important to mention that the list of mutation operators, available in MutationTestRunner (Figure 2), is automatically updated, after the message (Figure 3) is sent. Also code that is needed to support new operators is automatically generated.

3.6 Mutation Testing Library Classes

Mutation Testing of library classes (e.g., Set) introduces an additional challenge since these classes may be in use by the image, the runtime, or the mutation testing tool itself. Therefore, direct application of mutation testing on library classes could result in unpredictable and usually incorrect behavior. Existing tools do not specifically deal with this issue, but leave it up to the user to copy the class and perform necessary changes (e.g., renaming). SMutant comes with support for testing classes that are used by the image. Instead of direct application of mutation testing as described earlier, the tool copies the classes (`classesToMutate` and `classesWithTests`), and performs necessary changes to the new classes before applying the mutations. The user can easily turn this feature on by returning "true" from `isCopy` method (see Figure 1). Caution is needed with `isCopy` option; whenever possible this option should be off, since it introduces additional overhead. Note that the classes are copied only once, at the beginning of a mutation testing run.

3.7 Implementation Details

SMutant is developed in Pharo [10] Smalltalk. Nevertheless, we expect that the tool should work, after minor modifications (e.g., renaming), with the Squeak [11] implementation of Smalltalk. It is important to mention that no single line of code has to be modified/deleted/added in the base image in order to support SMutant. Changes that are needed to support GUI, which is optional, are made automatically when the tool is installed by the user. These changes are reverted when the tool is uninstalled. In addition, SMutant, as many other tools for mutation testing [6,8], runs mutated version of the code in a separate thread and kills the thread if it did not finish in the given amount of time, which prevents infinite loops, deadlocks, and livelocks. SMutant deploys SUnit framework, described in Section 2, to perform mutation testing: for each mutant, SMutant creates a test case with the appropriate values that is executed by SUnit.

3.8 Limitations

There are several features that can be improved. First, specification of mutation scenarios should be separated from running mutation scenarios (Figure 2). We have designed a separate GUI, which has to be integrated in SMutant, for specifying mutation scenarios, because manually writing a list of mutation operators may be quite tedious, although not difficult. Second, at this point, defining new mutation operators can be done only by sending a single message (Figure 3). We plan to design a simple GUI to simplify defining new mutation operators. Finally, the applicability of defining new mutation operators is to be determined.

4. CONCLUSIONS

We presented SMutant, a tool for mutation testing of programs written in Smalltalk. Because the type information is not known before the program is executed, SMutant applies mutations dynamically (at run-time), when information about the types becomes available. Also, SMutant supports defining new mutation operators without requiring the user to know any of the implementation details of SMutant. The code that is needed to support new mutation operators is automatically generated. SMutant is available at `http://www.squeaksource.com/smutant/`.

Acknowledgments

We thank Vilas Jagannath and Darko Marinov for great comments on this work. Aleksandar Milicevic and Rohan Sharma provided comments on an early draft of this paper. We also thank the fellow students of CS 598REJ at the University of Illinois at Urbana-Champaign for constructive discussions on the material presented in this paper. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-0746856, CNS-0958199, and CCF-1012759.

5. REFERENCES

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

[2] L. Bottaci. Type Sensitive Application of Mutation Operators for Dynamically Typed Programs. In *ICST Workshops*, 2010.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 4(11), 1978.

[4] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE TSE*, 3(4), 1977.

[5] Heckle. `http://seattlerb.rubyforge.org/heckle/`.

[6] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting. Jumble Java Byte Code to Measure the Effectiveness of Unit Tests. In *MUTATION*, 2007.

[7] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE TSE*, 99(6), 2010.

[8] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2004.

[9] Pester. `http://jester.sourceforge.net/`.

[10] Pharo. `http://www.pharo-project.org/`.

[11] Squeak. `http://squeak.org/`.