Mutation Testing in Practice using Ruby

Nan Li, Michael West, and Anthony Escalona
Research and Development
Medidata Solutions
{nli, mwest, aescalona}@mdsol.com

Vinicius H. S. Durelli
University of Groningen
v.h.serapilha.durelli@rug.nl

Abstract—Mutation testing is an effective testing technique to
detect faults and improve code quality. However, few practitioners
have adopted mutation testing into practice, which raises several
questions: Are tests capable of killing mutants useful? What is
the main hindrance to adopting mutation testing in practice?
Can practitioners really integrate mutation testing into real-world
agile development processes?

In this paper, we present two major contributions. First, based on our analysis and knowledge of Ruby, we devised eight new mutation operators for Ruby. Second, we applied mutation testing to an industrial Ruby project at Medidata and reported the lessons learned from the study. We confirmed that mutation-adequate tests are useful and could improve code quality from the perspective of practitioners and found long mutation execution time hinders the agile process. In addition, we used an enterprise-level Amazon cloud-computing technique to reduce the computational cost of running mutants. Considering availability of a mutation testing tool with our suggested features, we argue that mutation testing can be used in practice.

I. INTRODUCTION AND MOTIVATION

When carrying out mutation testing [1], the program under test (PUT) is mutated, being turned into slightly different versions of itself, called mutants. Mutation operators define the rules on how to make these changes to the original program. If a given test makes a mutant behave differently from the original program, then the mutant is said to be “killed”. A mutant is said “equivalent” when the behavior of the original program is equivalent to that of the mutant, and hence equivalent mutants cannot be killed. A mutation-adequate test set kills all non-equivalent mutants. The mutation score is the proportion of killed mutants to the total amount of non-equivalent mutants.

Empirical studies have shown that mutation testing is more effective than many other coverage criteria in terms of fault detection [2–5]. Nevertheless, the main activities involved in carrying out mutation testing (i.e., killing mutants, identifying equivalent mutants, and executing tests against mutants) are notably costly. Gopinath et al. [6] even claimed that mutation testing is not used by real-world developers with any frequency. Moreover, they concluded that developers should use statement coverage instead.

The controversy lead us to study why mutation testing cannot be used in industry. We think some questions are still not answered from the perspective of practitioners. (i) Is it hard for practitioners to understand mutation testing? (ii) Who should perform mutation testing, developers or testers? (iii) Are mutation tests meaningful to developers? Do the mutation tests help developers detect faults and refactor the code? We have not seen any positive confirmation about these two questions from real-world practitioners in previous studies. (iv) Can mutation testing fit in agile development processes? Does mutation testing hinder the continuous integration (CI) methodology? (v) If running mutants on personal computers takes too much time, could we leverage any matured enterprise-level cloud-computing technique to reduce the execution time? In addition, we would like to study Ruby mutation operators because we did not find any papers to study Ruby mutation operators previously, to the best of our knowledge.

Medidata produces clinical-trial software in many languages including Ruby. At Medidata we use coverage criteria as metrics to measure code quality. To the best of our knowledge, only a few statement coverage tools are available for Ruby such as simplecov [7] and there are no mature branch coverage tools. To ensure software quality, we strive to achieve above 90% statement coverage for our products. Furthermore, we are actively seeking advanced testing methods to improve code quality, as our products are monitored by the U.S. Food and Drug Administration (FDA). Thus, we set out to investigate how to use mutation testing in real world development environments.

We used a tool called mutant [8], which is the only mutation testing tool that is actively maintained for Ruby.1 To distinguish the name from terms like “mutation” and “mutants”, we refer to it as muRuby throughout this paper. We studied the mutation operators provided by muRuby. In addition, based on our experience with Ruby, we proposed eight new mutation operators that mimic developers’ mistakes.

We applied mutation testing to a real-world software product at Medidata. We presented the results and lessons from the study, and we discovered what hinders the agile process the most while using mutation testing. We summarized how practitioners should perform mutation testing and what features a successful mutation testing tool should have. We also applied the enterprise-level cloud computing, the Amazon Elastic Compute Cloud (Amazon EC2) [10], to mutation testing for the first time, reducing the test execution time.

The paper is organized as follows. Section II discusses related work. Section III introduces muRuby’s mutation operators and the mutation operators we devised. Section IV presents the experience and lessons learned from the study. Section V concludes the paper and outlines future work.

1Jia et al. [9] introduced another Ruby mutation testing tool is, heckle, which is no longer actively maintained.
II. RELATED WORK

To the best of our knowledge, few practitioners have studied how to apply mutation testing in practice. Shelton et al. [11] applied mutation testing in test-driven development (TDD) cycles to investigate whether adding criteria-based tests hinders agile processes. However, the subject program was not an industrial project and was not developed in real-world agile settings. This paper applied mutation testing to a software product that was developed in an agile development cycle. Untch proposed three categories for mutation testing strategies: do-fewer, do-smarter, and do-faster [12]. Some do-smarter approaches proposed parallelizing the test execution of mutants over multiple machines [13–16]. However, none of the proposed techniques can be adapted directly by industry. In this paper, we use an enterprise-level cloud-computing technique, Amazon EC2, to reduce the computational cost of mutation testing.

III. RUBY MUTATION OPERATORS

We examined the mutation operators (first-order) of muRuby and analyzed their effectiveness. Then, based on our experience with Ruby, we devised several new mutation operators that are not covered by muRuby.

A. Existing Ruby Mutation Operators

muRuby implements 54 mutation operators, focusing on language features and constructs. Some operators mutate the reserved words true, false, and nil. Some operators change control flow statements as if, while other operators act upon smart collections (e.g., arrays and hashes) and data types (e.g., floating-point data types).

Before using muRuby in our study, we wanted to get a sense of how effective the mutation operators are. So we compared this set of mutation operators with the set implemented by muJava [17] and the extended statement deletion (SDL) operator [18]. muJava follows the selective mutation approach [19], defining 12 mutation operators [20]: Arithmetic Operator Replacement (AOR), Arithmetic Operator Insertion (AOI), Relational Operator Deletion (ROD), Relational Operator Insertion (ROI), Logical Operator Replacement (LOR), Logical Operator Insertion (LOI), Assignment Operator Replacement (ASR), and Conditional Operator Replacement (COR), Conditional Operator Insertion (COI), Conditional Operator Deletion (COD), Shift Operator Replacement (SOR), Logical Operator Replacement (LOR), Logical Operator Insertion (LOI), and Assignment Operator Replacement (ASR).

The selective mutation operators used in muJava focus on mutating six different kinds of constructs: arithmetic, relational, conditional, shift, logical, and assignment operators. Since most programming languages have six basic constructs, other languages take advantage of the selective mutation operator set. However, muRuby partially implements ROR, COR, and COD.

ROR [21] defines that any of the six equality, relational, and conditional operators (>, >=, <, <=, ==, ! =) in a given statement should be replaced by other operators in the set and the whole expression is replaced with true and false. muRuby implements ROR slightly differently. More specifically, the first four relational operators >, >=, <, and <= are implemented as follows. If an expression contains a > operator, the operator is replaced by >= and == and the whole relational expression is replaced by true and false. However, > is not replaced by <, ! =, and <=. Similarly, <= is replaced by < and == and the whole expression is replaced by true and false. As for == and ! =, the expression is replaced by true, true, and false. Based on Kaminski et al.’s fault hierarchies for ROR mutation [22], for one relational operator, we only need three mutants to detect other seven ROR mutants. Thus, muRuby may fail to detect (i) mutant != for operators > and <; (ii) mutants <= and >= for operator ==; (iii) and mutants < and > for operator !=.

muRuby has mutation operators to replace and delete the conditional operators &&, ||, and, and, or when a predicate has two clauses. If a predicate has more than two clauses, the first two clauses will be combined with parentheses and considered to be one big clause. Essentially, the operators and and or are semantically equivalent to && and ||. The main difference is that && and || have higher precedence than and or. Additionally, some mutants are created by replacing the expression on the left or right side by a relational or conditional operator. These mutants are valid because the interpreter evaluates everything but false and nil to true.2

muRuby does not implement selective mutation operators in a conventional way. Thus, we assume that a mutation-adequate test set created by muRuby cannot fully tap into the full potential of the conventional selective mutation. Nevertheless, muRuby does implement the SDL operator similarly to the extended SDL by Deng et al. [18]. The SDL operator, which was initially defined for C [23] and for Ada [24], deletes single statements. Deng et al. built on this idea extending the SDL operator to other control structures including while, if, for, switch, return statements, and try-catch blocks. The extended SDL operator considers all possible cases: boolean conditions and statements inside the scope of control flow structures are also deleted. Nested control flow structures are treated recursively. For example, for an if structure, the SDL operator creates mutants by deleting the boolean expression, deleting every inner statement, deleting the whole if block, and deleting the else branches, if any. Besides the additional rules proposed by Deng et al., muRuby also creates additional mutants. For instance, an inner statement of a control flow structure is deleted and replaced by nil. Therefore, the mutation operators used in muRuby are at least as effective as the extended SDL mutation operator. Deng et al. [18] and Delamaro et al. [25] showed that SDL-adequate tests can detect 92% of mutants generated by the mutation operators for Java and C, respectively. All in all, considering all the other mutation operators that muRuby implements, we argue that the current set of mutation operators is fairly effective.

B. New Ruby Mutation Operators

Most mutation operators supported by muRuby are commonly seen in mutation tools for statically typed languages. Existing mutation operators do not mutate constructs that are particularly relevant to dynamically typed languages. We surmise that additional operators are needed to provide a more comprehensive set of operators, which truly reflects the types of faults that occur in programs written in Ruby. We list eight new mutation operators below.

2In Ruby every value is an object. nil, for example, is an instance of the NilClass.
• Bang (!) mutation operator: In Ruby, methods with a trailing exclamation mark in their names, usually called bang methods, modify their receivers in place. In contrast, methods without the bang perform actions and return a new object. A mutation operator could explore this Ruby convention by removing the exclamation point from bang methods and adding trailing exclamation points to non-bang methods.

• Single-quoted and double-quoted strings mutation operator: String values can be included in either single quotes or double quotes. Single-quoted strings require almost no processing but double-quoted strings demand more processing. Initially, Ruby looks for sequences that start with a backslash character (e.g., \n) and replaces them with some binary value. Afterwards, double-quoted strings are also subject to expression interpolation: occurrences as #{expression} are replaced by the value of the expression. We believe that a mutation operator that changes single-quoted string to double-quoted and vice versa would produce interesting mutants.

• Array index mutation operator: In Ruby, negative numbers can be used to index from the end of the array. For instance, an index of -1 returns the last element of the array. If the index is greater or less than the size of the array, Ruby returns nil. It would be interesting to have a mutation operator that adds a minus sign to positive array indexes or turns negative array indexes into positive ones by doing the opposite.

• Parallel assignment mutation operator: If the right side of an assignment has more than one value, separated by commas, all the values on the right side are collected into an array. If the left side has only one element, the array is assigned to that element; if the left side has more than one value or values followed by a comma, values on the right side match against successive elements on the left side. Excessive elements on either side of assignment expressions are discarded. For instance, c, = 1, 2 assigns 1 to c. An array containing both values on the right-hand side (i.e., [1, 2]) is assigned to c when the comma on the left-hand side of the expression is removed. Such an operator could remove values from the right side of the assignment expression when there are more elements on the right side than on the left side. The operator could also remove elements from the left side when there are more elements on the right side of the expression. Moreover, the proposed operator could also use the splat operator. The splat operator can be used to extract various elements from a multiple assignment expression. For instance, the expression a, *b = 1, 2, 3, 4 assigns 1 to a and [2, 3, 4] to b. Similarly, *a, b = 1, 2, 3, 4 assigns [1, 2, 3] to a and 4 to b. Thus, the operator could delete elements from one side at a time and then add the splat operator to one of the elements on the side with fewer elements.

• Symbol/string replacement mutation operator: Symbols are key elements of the Ruby programming style. Symbols represent names in program code and they are created using the :name and :"string" literals

syntax. There is one important difference between symbols and strings: symbols are immutable and their internal values are unique. Due to their characteristics, symbols lend themselves well to being used as hash keys in Ruby. In most cases, however, strings and symbols can be used interchangeably. For instance, novice programmers may use strings as hash keys instead of symbols. As a result, we believe that a mutation operator that replaces symbols by their corresponding strings and vice versa could yield valuable mutants.

• Return mutation operator: In Ruby, it is not necessary to use return to return an object from a method. So an operator that removes the return keyword or adds it to statements in a method could potentially yield effective mutants.

• Freezing mutation operator: The freeze method prevents users from changing a given object. Any attempt to modify a frozen object causes a RuntimeError exception. We conjecture that a mutation operator that takes advantage of the freeze method can produce interesting mutants. This operator adds an invocation to or removes the freeze method from an object.

• Object evaluation mutation operator: Ruby has three confusing methods kind_of?, is_a?, and instance_of?. The first two are synonymous, checking whether an object is an instance of the target class or its subclass. However, instance_of? returns true only when the object is an instance of the target class, not a subclass. We proposed an operator that changes kind_of? and is_a? to instance_of? and replaces calls to instance_of? by kind_of? and is_a?.

IV. Industrial Experience: Mutation Testing in the Real World Using Ruby

The goal of this study is to investigate the problems when applying mutation testing in practice, including whether mutation tests are useful to developers and how to apply mutation testing in agile development processes. The first author is a development engineer in test. The second author is a development manager, who architected and developed the subject program we examined during the study. The third author is a test manager. In this study, we set up a real development environment for a real-world Ruby product at Medidata and applied mutation testing to this product.

A. Subjects

The subjects are Ruby classes and modules from one Medidata project. This project has a server that records, manages, stores, and retrieves clinical audits from clinical trial systems. The client of the project generates and stores audits locally and asynchronously writes the audits to the server. Eight classes/modules from the project’s client were selected as the subjects in this study.

B. Procedure

In this study, we applied mutation testing in an agile team that did a two week time-box referred to as a sprint in agile
scrum methodology. Participants in the agile team created stories that specified activities and added concrete sub-tasks to the stories using a project management tool, JIRA [26]. For this sprint, we had one story: applying mutation testing to the subjects. The first author generated most of the mutation tests, working closely with the second and third authors. The second and third authors evaluated mutation tests and provided support in identifying equivalent mutants, running tests using Amazon EC2, and analyzing the results.

Before the sprint, the first and third authors set up the development environment and understood the subjects and associated tests, with the help of the second author. The authors also learned how to use muRuby. At Medidata, we control and manage product source code using GitHub, a web-based repository hosting service. We forked a new branch from one of the latest stable releases. This branch has achieved 96% statement coverage by RSpec behavior-driven tests [27]. The procedure of the study is shown below.

1) For each subject under test (SUT), ran the existing test sets against muRuby mutants locally and on cloud five times, recorded the time and computed the average, and wrote down the mutation scores and live mutants.
2) For each SUT, developed mutation tests to kill the mutants and identified equivalent mutants by hand.
3) For each SUT, ran the mutation-adapted test tests locally and on cloud five times, recorded the time, and computed the average.

C. Results and Lessons Learned

The first column of Table I represents the eight subjects. The second column shows the lines of code. The third column gives the number of original tests, followed by the original tests’ statement coverage in the fourth column. The fifth column shows the number of each subject’s total mutants. The sixth column MS-Original means the mutation score of the original tests. The next two columns show the additional mutation tests and equivalent mutants. The last column presents the mutation score of all the tests. The Ave row computes the arithmetic means for the columns and Total row sums the numbers for the other columns.

As shown in Table I, the subject 1 has 98 original tests but has only 27 mutants. The reason is that this subject is a Ruby singleton class and most methods of this class are defined inside a structure class << self ... end. But muRuby does not generate mutants for this structure. We have two findings from Table I. First, we noticed that the equivalent mutant ratio is relatively low, 1.8%, considering 54 mutation operators implemented in muRuby. This number is usually between 10%-20% for other languages. Second, the mutation score (0.99) of all the tests is very high because 0.9 is usually considered to be high.

In the study, we generated 38 additional mutation tests to kill 425 mutants that were not killed by the original tests and identified 31 equivalent mutants. The analysis of the tests and mutants revealed two important results. First, the mutation tests are meaningful to developers, although we did not find faults. Using mutation testing forced us to generate different test inputs. Even if we used the same test inputs as the original tests in the mutation tests, we checked different program states to kill mutants. We already knew that tests that have the same inputs but different test oracles differ in detecting faults [28]. Conversely, we can use mutation testing to decide what test oracles we should write. Since mutation tests have different test inputs and check different program states, the authors thought the mutation tests add value to the existing test set, from the perspective of practitioners. Although a few mutants were hard to kill, we did not feel generating a mutation test took significantly more time than writing a normal test case for most mutants.

Secondly, we confirmed that equivalent mutants could help developers to refactor the source code to improve the quality. A previous study by Shelton et al. has reported that mutation tests could detect weaknesses [11]. A weakness is a deficiency that does not affect the functionality, but may affect other software aspects including reducing performance, making maintenance harder, or making an application programming interface (API) difficult to use. For example below, a method initialize has two parameters. The first line is the original signature of the method and the second line is a mutant, which removes the &block parameter. Users have to attach a block to a method call if the method has yield. Then yield invokes the statements in the block when initialize is called. Thus, whether having &block as a parameter does not affect the method execution at all. Developers should refactor the code to remove &block.

```ruby
- def initialize(timeout, &block)
+ def initialize(timeout)

  ... yield

  ... end
```

Identifying equivalent mutants helped us remove dead code. We found that the mutants of one statement (which is shown below) could not be killed because the local variable sleep_seconds was never used anywhere. In this statement, if the hash options does not have the key represented by sleep_seconds, the double pipe operator is used to assign I to sleep_seconds. We partially did not notice this because the statement coverage tool reported that it was covered. Therefore, using mutation testing helped us discover more weaknesses than using statement coverage.

```ruby
sleep_seconds = options[sleep_seconds] || 1
```

Table II shows the time for executing tests against all the mutants by using the Amazon EC2. The first column shows the computers we used, followed by the number of the cores and memory of the computers in the second and third columns, respectively. The first row represents the MacBook Pro we are currently using for development and testing. The remaining rows are Amazon EC2 instances. MBP i7 has four 2.3GHz Intel i7 cores and 16 GB DDR3 memory. Amazon provides many kinds of EC2 instances. We chose four compute-optimized instances, c3.xlarge, c3.2xlarge, c3.4xlarge, and c3.8xlarge, and four general purpose instances, m3.medium, m3.large, m3.xlarge, and m3.2xlarge. EC2 instances are easy to set up. Starting or stopping an instance takes less than one minute. The last four columns show four subjects that had long execution time. Their numbers match the subject numbers in Table I. Note that the execution time for each subject for each computing.

---

3A test oracle determines whether a test passes by checking the program state. An example of a test oracle is an assertion in a unit test.
instance was measured five times and Table II shows only the arithmetic mean of the measures. The time was rounded in seconds.

Table II shows that only c3.4xlarge and c3.8xlarge were much faster than MBP i7. We should not use m3.medium, m3.large, m3.xlarge, and c3.xlarge since they were slower. We should also not use m3.2xlarge and c3.2xlarge since using them did not reduce the execution time significantly but costs go up. C3.4xlarge and c3.8xlarge were much faster than MBP i7, taking 57% and 41% of the execution time by MBP i7 on average. When executing all the tests against the mutants, c3.4xlarge and c3.8xlarge took 58% and 43% of the time used by MBP i7 on average. The detailed data for this is not shown due to space. The pricing for using on-demand c3.4xlarge and c3.8xlarge instances is $0.840 and $1.680 per hour as of January 2015. Thus, using c3.4xlarge is more cost-effective than using m3.xlarge. Developers should use c3.8xlarge if the cost is not a problem (the cost is $1.68 * 8 hours *10 days = $134.4 for one sprint). Additionally, c3.8xlarge was not twice as fast than c3.4xlarge. Matz’s Ruby Interpreter (Ruby MRI)’s global interpreter lock (GIL) synchronizes the executions of threads and allows exactly one thread to execute at a time. So muRuby might not fully perform parallel computation on Amazon EC2 instances.

Besides the above results, we learned lessons about how to apply mutation testing in practice. First, it was not very difficult for the second and third authors to understand mutation testing. Furthermore, we felt that developers should generate mutation tests at the unit-testing level because killing mutants requires practitioners to fully understand the code. In most companies, testers only write integration and acceptance tests that have APIs calls and they are not familiar with detailed implementation.

Second, we think that the main hurdle that prevents practitioners from using mutation testing is the long execution time. Table II shows that MBP i7 took almost 40 minutes for subject 6. In the agile process, we want to have instant feedback for the continuous integration that requires developers to frequently integrate latest changes to an existing code repository. We want mutation tests that can be quickly validated against mutants and picked up by CI build systems. It would hinder agile iterations if this process takes a long time. Obviously, muRuby has too many mutation operators. To get instant feedback, we can either use a small set of mutation operators or faster computers. Thus, selective mutation and cloud-computing techniques are promising. For this study, we expect that running tests against all mutants for a middle-size class (less than 50 LOC) should not take more than two minutes.

Next, we summarize the five points about what features a successful mutation testing tool should have. (i) The tool should be easy to install and use, being compatible with the development environment and having a command line interface. (ii) The tool should support parallel computing. (iii) The tool should allow users to choose which mutation operators to use. Users can use just SDL or a selective set instead of using all mutation operators. (iv) If a mutant causes an infinite loop, the tool should stop the execution and kill this mutant automatically. (v) It would be nice if the tool could remember which mutants were killed in previous executions so tests will not be executed against the killed mutants again. muRuby needs to be improved to implement the last three aspects.

From this study, we found out practitioners can add meaningful tests and improve code quality by using mutation testing. Since the mutation tests added value to the existing test set, we have merged these tests to the main development branch after this sprint. Thus, time spent on killing mutants and identifying equivalent mutants is acceptable. However, long test execution against mutants hinders the continuous integration in the agile process. We need a good mutation testing tool that must have selective mutation operators and fully support parallel computing.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Core</th>
<th>Mem</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBP i7</td>
<td>4</td>
<td>16</td>
<td>23</td>
<td>40</td>
<td>2382</td>
<td>63</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>4</td>
<td>238</td>
<td>316</td>
<td>17895</td>
<td>632</td>
</tr>
<tr>
<td>m3.large</td>
<td>2</td>
<td>8</td>
<td>75</td>
<td>139</td>
<td>8444</td>
<td>202</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>12</td>
<td>42</td>
<td>74</td>
<td>4547</td>
<td>108</td>
</tr>
<tr>
<td>m3.2xlarge</td>
<td>8</td>
<td>30</td>
<td>24</td>
<td>39</td>
<td>2357</td>
<td>60</td>
</tr>
<tr>
<td>c3.xlarge</td>
<td>4</td>
<td>8</td>
<td>42</td>
<td>74</td>
<td>85644</td>
<td>107</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>15</td>
<td>23</td>
<td>39</td>
<td>2318</td>
<td>57</td>
</tr>
<tr>
<td>c3.4xlarge</td>
<td>16</td>
<td>30</td>
<td>14</td>
<td>23</td>
<td>1305</td>
<td>33</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>60</td>
<td>9</td>
<td>16</td>
<td>1165</td>
<td>21</td>
</tr>
</tbody>
</table>

D. Threats to Validity

One threat to external validity is that we used one project at Medidata and we cannot be sure this subject is representative. We used muRuby mutation operators to generate and kill mutants. Using different mutation operators or tools could cause different results. We used AWS EC2 instances in the U.S. east region, using EC2 instances in a different region could change the results. Another internal threat is that the authors identified equivalent mutants by hand. Mistakes during the identification of equivalent mutants could affect the results.

<table>
<thead>
<tr>
<th>Subject</th>
<th>#LOC</th>
<th>#Original Tests</th>
<th>SC</th>
<th>#All Mutants</th>
<th>MS-Original</th>
<th>#Mutation Tests</th>
<th>#Equivalent</th>
<th>MS-All</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120</td>
<td>98</td>
<td>75%</td>
<td>27</td>
<td>0.96</td>
<td>1</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>3</td>
<td>100.0%</td>
<td>64</td>
<td>0.23</td>
<td>4</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>2</td>
<td>80.0%</td>
<td>204</td>
<td>0.53</td>
<td>7</td>
<td>5</td>
<td>100.0%</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>2</td>
<td>100.0%</td>
<td>114</td>
<td>0.96</td>
<td>2</td>
<td>2</td>
<td>100.0%</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>9</td>
<td>100.0%</td>
<td>145</td>
<td>0.99</td>
<td>2</td>
<td>1</td>
<td>100.0%</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>25</td>
<td>100.0%</td>
<td>852</td>
<td>0.69</td>
<td>20</td>
<td>13</td>
<td>0.98</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>1</td>
<td>100.0%</td>
<td>10</td>
<td>1.00</td>
<td>0</td>
<td>1</td>
<td>100.0%</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>8</td>
<td>100.0%</td>
<td>300</td>
<td>0.90</td>
<td>2</td>
<td>9</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

TABLE I. RESULTS OF APPLYING MUTATION TESTING.
V. CONCLUSIONS AND FUTURE WORK

In this paper, we examined the mutation operators implemented by muRuby. Based on our Ruby programming experience and its dynamically typed features, we developed eight new Ruby mutation operators. Then we used muRuby to generate mutation-adequate tests for a Medidata project within an agile development sprint. The results showed that mutation testing adds value to an existing test set that has achieved 97% statement coverage. Specifically, the added mutation tests had different test inputs and test oracles, and identifying equivalent mutants forces developers to refactor the code by removing weaknesses. We found that using the enterprise-level Amazon EC2 reduced the cost of mutation testing in terms of test execution time. We also provided our lessons learned from this study. With a tool that has our suggested features, we believe mutation testing can be used in real-world agile development environments.

In the future, we would like to conduct experiments to study the cost-effectiveness of Ruby mutation operators, implement new mutation operators, and create a selective set of mutation operators. We would also like to improve muRuby based on the lessons from the study.

ACKNOWLEDGEMENT

We offer our sincerest gratitude to Isaac Wong, VP of Platform Architecture at Medidata, for supporting this research, and Nidhi Agrawal, Agile Coach, for reviewing the paper. Vinicius Durelli would like to thank the financial support provided by CAPES (grant number BEX 1714/14-7).

REFERENCES