2016 IEEE International Conference on Software Quality, Reliability and Security

Mutation Integration Testing

Mark Grechanik
University of Illinois at Chicago
Chicago, IL 60607
drmark@uic.edu

Gurudev Devanla
University of Illinois at Chicago
Chicago, IL 60607
gdevan2@uic.edu

***Abstract*—In *integration testing*, integrated software modules or components are evaluated as a whole to determine if they behave correctly. *Mutation testing* is recognized as one of the strongest approaches for evaluating the effectiveness of test suites, and it is important to generate effective mutants efficiently for integration tests. However, it is difficult to generate integration mutants that create an error state in one component with certain assurances that this error state will affect computations in some other components. Unfortunately, little research exists that addresses this big and important problem to improve the quality of integration test suites.**

In this paper, we propose a theory and a solution for generating mutants that specifically target integration tests. We formulate a fault model for integration bugs that uses static dataflow analysis to obtain information about how integrated components interact in an application. Integration mutants are generated by applying mutation operators to instructions that lie in dataflow paths among integrated components. We implemented our approach and evaluated it on five open-source applications. In comparison to μJava, our approach reduces the number of generated mutants by up to approximately 19 times with a strong power to determine inadequacies in integration test suites.

I. Introduction

Integration testing has emerged as a major testing approach, since the majority of serious software defects are not isolated in single components and many catastrophic problems occur in interactions among different components [24], [29], [49], [55]. In integration testing, integrated software modules or components are evaluated as a whole to determine if they behave correctly [1, page 6] [6, page 21]. In general, in an integration test, interactions among two or more different components are tested [39]. For object-oriented software, integration tests invoke methods that belong to different classes, which exchange data as a result of these invocations [11]. The larger the project, the more important integration testing is [9], [53], since integration tests are reported to have a higher defect removal efficiency [8], [29], [33].

Mutation testing is recognized as one of the strongest approaches for evaluating the effectiveness of test suites [14], [25]. The code of the *application under test (AUT)*, P, is modified by applying *mutation operators* to the AUT's code to create a buggy but syntactically correct version of the AUT, P', i.e., a *mutant*. Running the test suite for P on P' should fail some tests, i.e., the mutant is killed. Otherwise, the test suite is deemed not adequate to find bugs and it should be enhanced with new tests that can kill mutants that were not killed with the previous version of the test suite.

It is difficult to generate effective mutants that target integration tests only. Deciding where in the AUT's code to apply mutation operators to target integration tests requires specific knowledge of how different components interact in applications. That is, an integration mutant should create an error state in one component with assurances that this error state will result in a failure in some other components. Generating mutants by applying mutation operators to statements and expressions whose values never reach integrated components leads to wasted time and effort, and doing so increases the cost and reduces the effectiveness of mutation testing. A fundamental problem of software testing is how to automatically find instructions in the AUT to which to apply mutation operators to generate effective mutants that find inadequacies in integration test suites efficiently.

A mutation integration testing tool is effective and efficient if it generates only those mutants that create error states in some components that eventually propagate to other components. Our key insight is based on guiding mutation integration testing using data propagation paths through the AUT. Using static analysis we analyze how different components interact in the AUT, i.e., we compute dataflow paths that contain statements and expressions using which components exchange data. We show that applying mutation operators to instructions that lie in these dataflow paths will often result in error states that are likely to eventually cause failures in other components. This paper makes the following contributions.

- We formulated an integration component fault model using which we created an approach for generating mutants that target integration test suites.
- We implemented and evaluated a tool for *Java Mutation Integration Testing (jMINT)* on five open-source Java applications with sizes from 1.3KLOC to 27KLOC with integration test suites, some of which are written by programmers and others generated using FUSION [45]. jMINT generates up to approximately 19 times fewer mutants than μJava, a mutation system for Java [38], [42] and these generated integration mutants have a strong power to determine inadequacies in integration test suites.
- Our tool and experimental results are publicly available at http://www.cs.uic.edu/~drmark/jmint.htm.

II. The Problem

In this section, we explain an example, state the nature of integration bugs, and provide the problem statement.

978-1-5090-4127-5/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS.2016.47

```
1  input x, y, z
2  A.m( x, y );
3  if ( z > 0 ) {
4       C.m( A.useValue() ) } else {
5       B.m( A.useValue() ) }
```

Fig. 1: A pseudocode example of component interactivity.

A. Illustrative Example

In our empirical investigation, three graduate students at the University of Illinois at Chicago studied close to 700 bugs that were randomly chosen from Mozilla and Apache bug repositories. We summarize these integration bugs in an example of the pseudocode that is shown in Figure 1. An idea of this example is to show how components are integrated in the application. Line `1` specifies that input variables `x`, `y`, `and z` are initialized with some values. In line `2`, the method `m` of the instance of the class `A` is invoked with the parameters `x` and `y` and it computes some result that is stored internally in an object of this class. If the value of `z` is positive, components `C` and `A` interact in in line `4` where this value computed in line `2` is passed to the object of the class `C` using the method `useValue` of the class `A`. Otherwise, components `B` and `A` interact in line `5`. We say that the class `A` is integrated with the class `C` in line `4` and with the class `B` in line `5`.

An integration bug can result from an error that is outside the scope of the integrated classes. Suppose that an error is made where the operator ">" in the conditional expression of line `3` is replaced with the operator "<". Then, the components `A` and `B` will interact instead of the components `A` and `C`. This example shows that a small semantic error leads to changing the control flow at runtime that results in not invoking proper integrated components.

A different type of an integration bug can be illustrated by mutating the body of the method `A.m`, so that it computes some incorrect internal value. In this case, the computation state becomes incorrect, and the incorrect value from the class `A` is passed to the methods `m` of the components `B` or `C`.

B. Nature of Integration Bugs

Integration bugs manifest themselves when executing the code of the application where an *integration fault* in one component results in an error state is produced in this component that leads to a failure in a different software component. A key property of integration bugs is that it is difficult to find them using *unit testing* where implementations of methods that belong to the same class are tested individually in isolation [16], [40]. This property is confirmed in multiple studies [8], [29] including the recently released Google dataset of test suite results[1]. Failure rates for medium and large size tests (i.e., integration tests) in the Google dataset are higher by the order of several magnitudes than for small tests (i.e., mostly unit tests). A reason for this gap in the bug-finding powers is that, by the definition of unit testing, a unit test for a given method does not contain assertions that check if an erroneous state affects other components.

[1]https://code.google.com/p/google-shared-dataset-of-test-suite-results/

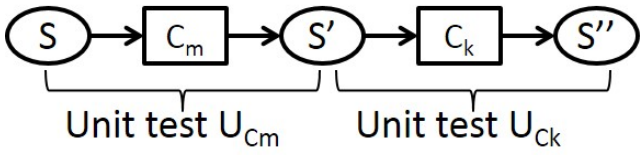

Fig. 2: An integration model of components C_m and C_k.

Consider a model in Figure 2 where two components C_m and C_k are integrated in the context of some software application. The input to C_m is the state, S, (i.e., a set of locations and their values), which is transformed by some instructions in C_m to the state, S', which is the input to the component C_k that transforms this state into S''. Programmers create corresponding unit tests, U_{C_m} and U_{C_k} for the components C_m and C_k. These unit tests have assertions that check preconditions and postconditions. Specifically, the unit test, U_{C_m} checks the input state, S, if it satisfies preconditions and the output state, S' if it satisfies postconditions. The unit test, U_{C_k} uses the state, S' as its input state to check its preconditions and it checks the output state, S'' if postconditions are satisfied. Using this model we can define a notion of the integration error.

Definition 1 *An integration error between two components C_m and C_k is an error state, S'' that is produced by C_k and that results from a fault in some executed instruction, $i \in C_m$. The fault in i results in the state, S' that is correct w.r.t. assertions in U_{C_m}, and some instruction, $j \in C_k$ uses values from S' that result in the error state, S'', and there exist no assertions in U_{C_k} that detect the error state, S' to prevent S''.*

Thus, a fundamental nature of integration bugs is in the *architectural mismatch*, where programmers make incorrect assumptions about various integrated components in software applications [17]. When creating unit tests for methods of a given component, a programmer often does not have enough information on how these methods will be used by other programmers who will integrate this component with their components. For example, suppose that in the pseudocode in Figure 1 the method `m` of the class `A` is invoked in line `2` and it returns an integer, $]-\infty,+\infty[$. However, different programmers who implement the methods `m` of the classes `B` and `C` may assume that the returned integer value is always positive. It is practically infeasible for programmers to create a comprehensive set of assertions for each unit test that will check all aspects of the state to ensure that it does not violate any assumptions for methods of all software components. Even if these programmers create precondition assertions in their unit tests that check for the sign of the input parameter, an integration test is needed to reveal that the method `m` of the class `A` produces a state that is erroneous for the methods `m` of the classes `C` and `B`.

C. The Problem Statement

In this paper, we address a fundamental problem of mutation integration testing – how to determine the effectiveness of integration test suites efficiently. A root of this problem is that

applying mutation operators indiscriminately to all instructions results in a very large number of mutants, many of which are not related to integration tests. In general, the number of generated mutants is proportional to the number of classes and references in object-oriented programs [39]. Therefore, applying various mutation operators to all expressions and statements in nontrivial applications results in the order of millions of mutants. Since executing a test suite on each mutant takes a sizeable amount of time, it is infeasible to execute a test suite on all mutants. Our goal is to reduce the number of generated mutants by concentrating only on those that are specific to integration components. In this paper, we address two subproblems: one is to generate fewer mutants and the other is to increase the likelihood that the generated mutants will retain the power of determining that integration test suites are inadequate by ensuring that generated mutants result in integration errors according to Definition 1.

III. Our Solution

In this section, we describe our key idea, discuss our fault model, and explain the architecture of our solution for generating integration mutants.

A. Key Ideas

Our key idea is twofold. First, we detect dataflow paths using which integrated components exchange data. We ignore indirect integrations of components (e.g., when components exchange data by writing it into and reading it from files or external peripheral and network devices). In general, it is not possible to automatically and statically find all dataflow paths using which integrated components exchange data, especially in the presence of virtual dispatch, recursion, and loops. Our goal is to use conservative static analysis to detect some dataflow paths that have strong potential at runtime to exchange data among different components.

Second, once some dataflow paths between components are determined, our idea is to apply mutation operators only to those statements and expressions that lie in the dataflow paths. Since our goal is to produce only those mutants that target integration test suites specifically, we forego the completeness of mutant generation to gain integration mutant specificity.

B. A Fault Model For Integration Testing

In this section, we introduce a fault model and the theory of mutation integration testing. A fault model includes constraints, abstractions, and actions that specify incorrect or unacceptable behavior of an engineered system [15], [52]. With respect to integrated software components, a fault model describes violations of different properties that specify how these components interoperate (i.e., exchange data).

1) Integration Points: Throughout the rest of the paper we will use the term *integration point* to designate locations in the applications where different components exchange data.

Definition 2 *An integration point in the program, P, is an instruction, $i \in P$ where the data, d is used (i.e., its memory location is accessed) by some method of the object o_r of the class C_r and d is defined (i.e., a value is stored in its memory location) by a method of the object o_s of the class C_s at some instruction $j \in P$ is s.t. $r \neq s$. We call the classes C_r and C_s integrated iff $\exists d$ s.t. $def(d) \in C_s \wedge use(d) \in C_r, r \neq s$ and the execution of j precedes the execution of i.*

Consider the pseudocode in Figure 1 where some field of the class `A` is defined by invoking the method `m` of the class `A` in line `2`. Components `B` and `C` use this value in lines `4-5` and these are integration points. Mutating this program in a way that this value is affected or the paths to the integration points are modified is a goal of a mutation integration testing.

2) State Propagation: The state, S, of a program is the set of all locations and their values. We model each instruction in a program as a *transducer* that takes in the state, S and outputs a new state, S'. The input to the program is the initial state, S_0, it is propagated through a path that contains a sequence of transducers to reach some integration point, i.e., $S_0 \hookrightarrow S_1 \hookrightarrow^* S_n$. Not all paths end at an integration point, i.e., some computations may use methods of a single class and produce the output for the AUT. Using mutation operators, some transducer (i.e., corresponding instructions within the AUT) is modified, so that when the mutant is executed, the value at some location differs from the one that is produced by the original version of the program. That is, once mutated transducer is executed, it outputs an incorrect state. If this incorrect state is propagated through some integration point and no integration test fails, a test suite is deemed inadequate and should be enhanced with a new test that kills this mutant.

In this paper, we use a producer-consumer model with data flow analysis to describe how the state propagates through a program. We consider that the state of a program is modified by executing the assignment operation (when calling a method, an implicit assignment operator is invoked to assign values of actual parameter expressions to the variables in the scope of the method). The *right-hand side (RHS)* of the assignment contains uses of variables, i.e., their values are consumed to produce a new state by assigning new values to the variable on the *left-hand side (LHS)* of the assignment. When calling a method, the values of the input variables are consumed in the body of the method to produce new values that result in state modifications. Assignments and method calls can be viewed as transducers that take the program state as the input and output a modified program state that is in turn used as the input to some other transducer. This is the essence of state propagation from the inputs to a program to some integration points.

3) Tracking The State With DataFlow Analysis: To increase the effectiveness of mutation integration testing, mutation operators should be applied only to program instructions that result in seeding faults in the states that propagate to integration points. Our idea is to use dataflow analysis to compute some paths in a program where variables are defined and used [31]. A definition of a variable, x, designated as $x\downarrow$, is a location where a value for x is stored into memory; a use of a variable, x, designated as $x\uparrow$, is a location where x's value is accessed. Defining a variable results in

INIT

$$\langle \Sigma, \mathcal{C}, (\mathcal{P}[\mathcal{F} \mapsto \bot] \mapsto \emptyset), \mathtt{E}[init(x)]; S\rangle \hookrightarrow \langle \Sigma[x\downarrow], \mathcal{C}, \mathcal{P}[\mathcal{C}(l_i)]; S'\rangle$$

ASSIGN

$$\langle \Sigma, \mathcal{C}, \mathcal{P}, \mathtt{E}[x = \mathtt{E}[y \mapsto v]]; S\rangle \hookrightarrow \langle \Sigma[x\downarrow; y\uparrow], \mathcal{C}, \mathcal{P}[\mathcal{P}[y\uparrow] \cup \mathcal{C}(l_i)]; S'\rangle$$

BRANCH

$$\frac{\{l_j\} \in \mathtt{dom}(\mathcal{C}) \qquad 1 \leq j \leq n \qquad \mathcal{C}(l_j) = S' \qquad \mathtt{E}[x\uparrow \otimes y\uparrow] \qquad \otimes \in \mathtt{relop} \qquad l_j \neq l_j}{\langle \Sigma, \mathcal{C}, \mathcal{P}, \mathtt{E}[if(x\uparrow \otimes y\uparrow) l_1, \ldots, l_n]; S\rangle \hookrightarrow \langle \Sigma[x\uparrow; y\uparrow], \mathcal{C}, \mathcal{P}[\mathcal{P}[x\uparrow] \cup \mathcal{C}(l_j, \mathcal{F} \mapsto \mathcal{F} \vee \mathtt{0x10}); \mathcal{P}[y\uparrow] \cup \mathcal{C}(l_j)]; S'\rangle}$$

MCALL

$$\frac{\langle \Sigma, \mathcal{C}, \mathcal{P}, \mathtt{E}[o.m(y_1, \ldots, y_n)]; S\rangle \qquad \mathtt{E}[x_i = y_i] \qquad 1 \leq i \leq n \qquad T = \mathtt{typeof}(o) \qquad x_i \in \mathtt{memberof}(T) \cup \mathtt{memberof}(T.m)}{\langle \Sigma, \mathcal{C}, \mathcal{P}, \mathtt{E}[o.m(y_1, \ldots, y_n)]; S\rangle \hookrightarrow \langle \Sigma[x\downarrow; y\uparrow], \mathcal{C}, \mathcal{P}[\mathcal{P}[y\uparrow] \cup \mathcal{C}(l_j); S'\rangle}$$

INTPOINT

$$\frac{\langle \Sigma, \mathcal{C}, \mathcal{P}, \mathtt{E}[o_p, o_q]; S\rangle \qquad \mathtt{typeof}(o_p) \neq \mathtt{typeof}(o_q)}{\langle \Sigma, \mathcal{C}, \mathcal{P}, \mathtt{E}[o_p\downarrow \mapsto v_p; o_q\uparrow \mapsto v_q]; S\rangle \hookrightarrow \langle \Sigma[o_p\downarrow \mapsto v_p; o_q\uparrow], \mathcal{C}, \mathcal{P}[\mathcal{P}[o_q\uparrow] \cup \mathcal{C}(l_j), \mathcal{F} \mapsto \mathcal{F} \vee \mathtt{0x01}]; S'\rangle}$$

$$\mathtt{typeof} : o \to T \qquad \mathtt{memberof} : T \to \{m\} \qquad \mathtt{dom} : \mathcal{C} \to \{l\}$$

Fig. 3: Operational semantics. The rule INIT is executed when the variable x is defined first. The path, $\mathcal{P}$ records instructions, l from the codespace, $\mathcal{C}$ that designates the dataflow from the first definition of some variable, x, to branches and integration points. The flag, $\mathcal{F}$ of the path, $\mathcal{P}$, specifies the type of integration faults that could be injected by mutating instructions, l, along this path – type 0x01 means that some values of the state can be changed and type 0x10 means that different integration points can be reached and type 0x11 means that both faults can be produced. The rule ASSIGN connects the use of the variable, y in the expression on the right hand side to the definition of the variable, x on the left hand side. The rule BRANCH shows the evaluation of a conditional expression that uses variables x and y that are connected using some relational operator that is designated with the symbol $\otimes$. The rule MCALL describe how method calls are evaluated, specifically, how copying input parameter values results in stitching data flow paths that lead to the uses of the input parameter variables in the method. Finally, the rule INTPOINT specifies that the integration point is reached by a given dataflow path. E stands for the context in which a given rule is applied.

a state change; using a variable in the RHS expression in an assignment statement leads to the flow of its value to the definition of some variable on the LHS of the statement. We designate this data flow that leads to the state propagation as $S[y\downarrow := x\uparrow; x \mapsto v] \hookrightarrow S'[y \mapsto v]$. A *def-clear path* with respect to a variable, x, is the set of all instructions l in a program from some instruction, l_i to some instruction, l_j where $x\downarrow \in l_i$ and $\forall k, i < k \leq j, x\downarrow \notin l_k$. A *def-use path* with respect to a variable, x, is a def-clear path with the set of all instructions l in a program from some instruction, l_i to some instruction, l_j where $x\downarrow \in l_i$ and $\exists k, i < k \leq j, x\uparrow \in l_k$. A purpose of def-use paths is to detect the archetypical situation where some values of the locations that constitute the program's state are used to modify this state.

A state transfer instruction, l_s is an instruction on some def-use path with respect to different variables, x, y where $x\downarrow \in l_s \wedge y\uparrow \in l_s$, e.g., $x := y * y$.

Definition 3 *A state transfer path is a sequence of def-use paths $P = p_1, p_2, \ldots, p_n$ that contain instructions, $p_k = l_1, l_2, \ldots, l_m$, $1 \leq k \leq n$. For each consecutive pairs of def-use paths, p_i, p_j, $\exists l_r \in p_i \wedge l_q \in p_j$ such that $x\downarrow \in l_r \wedge x\uparrow \in l_q$.*

Essentially, we define a *state transfer path* as a concatenation of def-use paths, where a variable defined in one path is used in some other path via state transfer instructions.

Definition 4 *An integration path is the state transfer path, $P = p_1, p_2, \ldots, p_n$ where $l_{\in} p_n$ is an integration point.*

Using a static dataflow analysis we can compute integration paths whose instructions will be used by mutation operators. Our hypothesis is that applying mutation operators to instructions in integration paths will likely lead to erroneous states that will be propagated to some integration points.

4) Operational Semantics: We use a simplified Java language semantics to define the operational semantics for mutation integration problem. We do not handle exceptions, reflection, interactions with networks and peripheral devices and native calls. Operational semantics rules are shown in Figure 3. A label $l \in \mathcal{C}$ refers to a program location in the program (or code heap, $\mathcal{C}$ that maps program labels to sequences of operations) and it is associated with one operation. We represent a Java program as a sequence of labels: $l_1, \ldots, l_{exit}$. The store, $\Sigma : Var \to Flow$ binds variable names to their abstract values, where the symbol Var denotes the domain of program variables and the symbol $Flow$ denotes the domain of abstract values. An uninitialized variable stores the abstract value $\bot$. The evaluation relation, defined by the reduction rules in Figure 3, has the form $\langle \Sigma, \mathcal{C}, \mathcal{P}[\mathcal{F}], \mathtt{E}[\mathtt{somexp}]; S\rangle \hookrightarrow \langle \Sigma[v], \mathcal{C}, \mathcal{P}[\mathcal{C}(l_i)]; S'\rangle$, read "Executing the expression someexp from the program with the store Σ and the initial state S at the instruction l_i from the

code heap $\mathcal{C}$ and the current set of state transfer paths $\mathcal{P}$ with the integration fault flag, $\mathcal{F}$, leads to executing these instructions for the expression and producing the resulting value, v, and the program transitions to a new state S'. We define the transition $\hookrightarrow \subseteq \langle \Sigma, \mathcal{C}, \mathcal{P}, \mathtt{E}[]; S\rangle \hookrightarrow \langle \Sigma[], \mathcal{C}, \mathcal{P}[]; S'\rangle$. Helper functions include the function `typeof` that returns the type of an object; the function `memberof` that checks if a given variable is the member of the set of the variables of a given type, T, and the function `dom` that returns the set of instructions for the given code heap, $\mathcal{C}$.

The operational semantics is driven by our fault model and Definition 3 and Definition 4. The rules include initialization of a variable (`INIT`), assignment statement semantics (`ASSIGN`), method call rule (`MCALL`) where the values of the parameter input variables are copied, thus transferring a part of the state into the method call, branching statement semantic rule (`BRANCH`) and a semantic rule for handling execution paths that reach integration points (`INTPOINT`).

A main idea for these rules is to capture instructions in the state transfer path, $\mathcal{P}$, whose flag, $\mathcal{F}$ can be assigned any of the values $\{\bot, \mathtt{0x01}, \mathtt{0x10}\}$ or their combination using the boolean operator $\vee$. Initially, the value of $\mathcal{F}$ is assigned $\bot$, i.e., uninitialized. When the rule `BRANCH` is applied, the value of $\mathcal{F}$ is $\vee$ed with `0x10`, meaning that the path in $\mathcal{P}$ leads to multiple branches. It means that applying mutation operators to instructions in this path may result in an error state that changes the execution path leading to a different integration point, i.e., a situation in our fault model where components will interact when they are not supposed to because of the error that affects the control flow path. Alternatively, when the rule `INTPOINT` is applied, the value of $\mathcal{F}$ is $\vee$ed with `0x01`, meaning that the path in $\mathcal{P}$ leads to an integration point. It means that applying mutation operators to instructions in this path may result in an error state that is eventually consumed by an integrated component.

C. Algorithm For Java Mutation Integration Testing (jMINT)

Our algorithm for generating integration paths for generating mutants with Java Mutation Integration Testing jMINT is shown in Algorithm 1. The algorithm's procedure **`ComputeAllIntegrationPaths`** takes as its input the next available program instruction, $l \in \mathtt{dom}(\mathcal{C})$ and it outputs the set of integration paths, $\mathcal{P}$. The body of this procedure spans Line 1–7; the global path variable is initialize in Line 2, the set of the input variables is obtained in Line 3, and for all input variables, the procedure **`AddIntegrationPaths`** is called in Lines 4–6 and it computes all integration paths that are appended to the global set of integration paths.

The algorithm's procedure **`AddIntegrationPaths`** in Line 8 takes an input a variable v and returns an integration path $\mathcal{P}$ if one exists for the v as described by the rules. At Line 9 a new path $\mathcal{P}$ is initialized using the **`CreateNewPath`** procedure. In Line 10 all uses of v is obtained with the procedure **`GetUses`**. Between Lines 11 and 25, each use expression returned by **`GetUses`** is identified with one of the rules (*ASSIGN*, *BRANCH*, *INTPOINT*). If the variable

Algorithm 1 jMINT's algorithm for computing integration paths.

```
1:  ComputeAllIntegrationPaths( l ∈ dom(C) )
2:  P ← ∅ {Initialize paths}
3:  V ← GetInputVars(l) {Get the set of variables}
4:  for all v_i ∈ V do
5:     P → P∪AddIntegrationPaths( v_i )
6:  end for
7:  return P
8:  AddIntegrationPaths( V )
9:  P_V ← CreateNewPath(V)
10: GetUses(V) ↦ l_V{We obtain instruction labels where
    variables are used.}
11: for all l ∈ l_V do
12:    E ← GetExpType(l)
13:    if E==ASSIGN then
14:       P_V ← P_V ∪ l
15:       B ← GetLHS(l) {Get the variable defs on the left-
          hand side of the assignment}
16:       for all b_j ∈ B do
17:          P_V → P_V∪AddIntegrationPaths( b_i )
18:       end for
19:    else if E==BRANCH then
20:       P_V[F] ← P_v[F] ∨ 0x10
21:    else if E==INTPOINT then
22:       P_V[F] ← P_v[F] ∨ 0x01
23:    else if E==EXIT then
24:       break
25:    end if
26: end for
27: if P_V[F] ∧ 0x01==true then
28:    return P_V
29: else
30:    return ∅
31: end if
```

$\mathcal{P}_v$ was assigned the flag `0x01`, then the procedure returns the integration path $\mathcal{P}_v$ for the variable v.

The algorithm's procedure **`AddIntegrationPaths`** recursively calls itself in Line 17, where it passes variables that are defined in the LHS of the assignment statements to comply with Definition 4. This recursive call composes multiple def-use paths into a state transfer path, so that a previous def-use path where some variable, b_i is defined, is composed with some subsequent def-use path, where some variable, b_j is defined using the variable, b_i. If a state transfer path ends at an integration point, it is appended to the set of all integration paths, otherwise, the last instruction of the given path terminates the recursion and eventually the algorithm.

Lemma 1 *The loop invariant between lines 11-26, will maintain the value of $\mathcal{P}$, such that $\forall l \in \mathcal{P}_v$, l is of the form $E[x = E[y \rightarrow v]]$ or $E[if(x \uparrow \otimes v \uparrow)]$ or $E[o.m(y_1, \ldots, y_n)]$, that is, **`GetExpType`**(l) satisfies either ASSIGN, BRANCH or INTPOINT rules.*

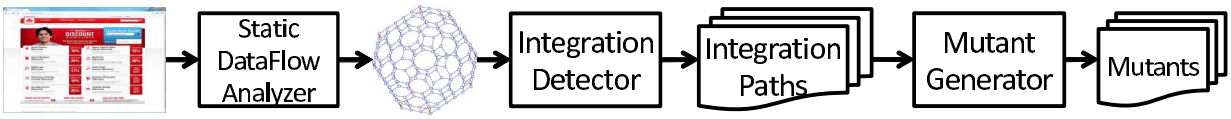

Fig. 4: jMINT's architecture and workflow.

Proof: A new instruction l is only added to $\mathcal{P}$ only if E satisfies either *ASSIGN*, *BRANCH* or *INTPOINT* rules from the operational semantics for a given variable v. ■

Lemma 2 *The function* **`AddIntegrationPaths`** *will return $\mathcal{P}_v$ if $\exists l \in \mathcal{P}_v$ s.t. satisfies the rule INTPOINT is satisfied.*

Proof: Line number 28 is only executed if $\mathcal{P}_v$ has the flag `0x01` set. Moreover, this flag is set only if $\exists l$ such that **`GetExpType`**$(l) = INTPOINT$ [lines 21-22] ■

Theorem *If $l_{exit} \in \texttt{dom}(\mathcal{C})$ is executed and the algorithm returns the set of paths, $\mathcal{P}$, that contains only integration paths.*

Proof: It trivially follows from Lemma 1 and Lemma 2 that on the execution of $l_{exit} \in \texttt{dom}(\mathcal{C})$ the algorithm returns the set of paths, $\mathcal{P}$ that only contains integration paths. ■

D. jMINT Architecture And Workflow

The architecture of jMINT is shown in Figure 4. The input to jMINT is the bytecode of the AUT. The Static DataFlow Analyzer produces a *dataflow graph (DFG)* that contains def-use paths in the AUT. Since our goal is to determine def-use paths that reach integration point, Integration Detector traverses DFG to compose def-use paths into state transfer paths and it checks which of these paths contain integration points. Once integration paths are computed, they are inputted to the Mutant Generator that applies mutation operators to the operations in the integration paths thus producing mutants.

E. Generating Integration Mutants

In this section, we describe mutation operators that we implemented in jMINT and we discuss how we use heuristics to reduce the number of generated equivalent mutants.

1) Mutation Operators: For jMINT, we picked a subset of mutation operators defined and implemented in μJava [37]. These operators target object-oriented features of Java (i.e., encapsulation, inheritance, polymorphism) and they are specifically designed to address class mutation. Since they have a stronger impact on the state of the AUT, we chose these operators to show that our approach makes it more effective and efficient to use these operators for mutation integration testing. In selecting mutation operators, we used a previous study that evaluated impacts of different mutation operators in object-oriented programs [28], [50] and a study that shows the usefulness of the class mutation operators [35].

The description below also provides the implementation details as to how these mutation operators are applied within the context of integration points described earlier in the paper.

As a first step in identifying potential mutants affecting integration points jMINT performs inter-procedural analysis using the procedure in [47]. During this step all ud-chains that span across two different types/classes are identified. The span of ud-chain across integration points determines the scope and applicability of each mutant operator as determined by rules in Figure 3.

The mutants listed below are described in detail in [36].In the description below we describe when and why a particular mutant is applicable at the identified integration point. In describing the mutants we will use the terms, X which refers to type X, m_x when referring to a method in class X, i_x when referring to an instance members of type X, v_{m_x} to refer to an variable defined in m_x and finally x refers to an instance of type X. Terms $X(...)$ refer to call to initialize an instance of X.

- EAM changes the name of the accessor method into some other syntactically compatible method. Given $m_a \in A$, if $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[v_{m_x} \mapsto b.m_b^0] \in \mathcal{C}(li)$ and m_b^0 and m_b^1 have a overloading relationship and act as accessor methods defined as *getXXX*, then the EAM operator replaces $E[v_{m_x} \mapsto b.m_b^0]$ with $E[v_{m_x} \mapsto b.m_b^1]$
- IOD deletes the declaration of some overriding method. Given $m_a \in A$, class B, $Parent(B) = P_b$ if $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[v_{m_x} \mapsto b.m_b] \in \mathcal{C}(li)$ and m_b and m_p have a overridding relationship, then the IOD operator deletes the definition of m_b.
- IPC deletes the call to the constructor `super`. Given $m_a \in A$, if $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[B(...)] \in \mathcal{C}(li)$, then the IPC operator generates a mutant by deleting the call to super in all forms of $B(...)$.
- JID deletes explicit initializations of class members. Given $m_a \in A$,if i_a and v_{m_a} share the same name, $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[i_a, b.m_b] \in \mathcal{C}(li)$, then the operator JID deletes any $E[i_a \mapsto \perp]$ that exists in all forms of $A(...)$
- JTD deletes uses of the keyword `this`. Given $m_a \in A$,if i_a and v_{m_a} share the same name, $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[v_{m_a}, b.m_b] \in \mathcal{C}(li)$, then the operator JTD replaces $E[v_{m_a}, b.m_b]$ with $E[i_a, b.m_b]$
- JTI inserts the keyword `this`. Given $m_a \in A$,if i_a and v_{m_a} share the same name, $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[i_a, b.m_b] \in \mathcal{C}(li)$, then the operator JTI replaces $E[i_a, b.m_b]$ with $E[v_{m_a}, b.m_b]$
- OMD deletes overloading method declarations. Given $m_a \in A$, if $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[v_{m_x} \mapsto b.m_b] \in \mathcal{C}(li)$ and m_b^0 and m_b^1 have a overloading relationship, then the OMR operator

deletes the method m_b. The expectation is that the overloaded version of m_b will be called by runtime.

OMR replaces the body of an overloading method with the body of some other method.
Given $m_a \in A$, if $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[v_{m_x} \mapsto b.m_b^0] \in \mathcal{C}(li)$ and m_b^0 and m_b^1 have a overloading relationship, then the OMR operator replaces $E[v_{m_x} \mapsto b.m_b^0]$ with $E[v_{m_x} \mapsto b.m_b^1]$

PNC replaces the name of the instantiated class in the operator `new` with the name of some other class.
Given $m_a \in A$, b is of type B, if $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[b \mapsto B(...)] \in \mathcal{C}(li)$, then the PNC operator generates a mutant by replacing $E[b \mapsto B(...)]$ with $[b \mapsto C(..)]$ where C is any subclass of B.

PRV changes operands in assignment statements.
Given $m_a \in A$, p is of type $Parent(B)$, if $\exists l_i \in m_a$ such that $\mathcal{C}(li) \in \mathcal{P}$ and $E[p \mapsto B(...)] \in \mathcal{C}(li)$, then the PNC operator generates a mutant by replacing $E[p \mapsto B(...)]$ with $[p \mapsto C(..)]$ where C is any subclass of P

2) Handling Equivalent Mutants: Related to the problem of effectiveness and efficiency of integration mutation testing is a problem of detecting *equivalent mutants*, i.e., those mutants that can never result in anomalous behavior [39]. Detecting equivalent mutants is undecidable and very expensive in general. However, multiple evidence show that using program slicing and impact analysis can improve the probability of generating non-equivalent mutants [22], [27]. Specifically, mutants that impact more statements and expressions are less likely to be equivalent. We partially address the problem of generating equivalent mutants in the context of jMINT, where we implement two heuristics that are based on the evidence that mutants that impact more statements and expressions are less likely to be equivalent. Our solution is influenced by Javalanche, a mutation framework that assesses the impact of individual mutations [48].

Our first heuristic is to give higher priority for applying mutation operators to instructions in integration paths where the same variables are used more than once. The more often a variable is used on average, the higher the probability that we will give a higher priority for generating a mutant for this integration path. Our second heuristic is to rank integration paths using three score levels. The highest score is given to an integration path that has more variables that are used in conditional expressions, thereby address the flow value integration fault in our integration fault model.

The intuition behind assigning the highest score is that applying mutation operators to instructions that use these variables can change the execution path for the same input values leading to different computation that is unlikely to be equivalent to the one in the original program. The lower score is assigned to integration paths where variables are used in as parameters in methods – we assume that methods contain many instructions and applying mutants to methods and their receivers are less likely to lead to equivalent mutants.

TABLE I: Characteristics the subject applications: their names followed by their versions and lines of code, the number of detected state transfer paths and number of tests.

AUT Name	Ver	KLOC	State xfer paths	#Tests
MONGO driver	4.0.0	27	6,750	795
NANOXML	2.2.1	7	704	191
SCRIBE	1.3.5	3	457	128
JCON	0.0.1	1.5	183	56
SHTUTXML	1.0.0	1.3	124	49

The cumulative score of the integration path is the linear combination of its scores and it determines its rank that is used against a predefined threshold value to select integration paths that have the smaller probability of generating equivalent mutants.

IV. Experimental Evaluation

In this section, we pose research questions (RQs), describe subject applications, explain our methodology and variables, and discuss threats to validity.

A. Research Questions

We seek to answer the following research questions.

RQ1: Does jMINT generate fewer mutants when compared to competitive approaches?

RQ2: Is it practical to use jMINT in that it does not require significantly more resources to generate mutants when compared to competitive approaches?

RQ3: Is jMINT effective in generating mutants that result in integration errors using Definition 1?

Recall that our goal is to balance multiple objectives: a)we want to generate fewer mutants than μJava; b) we want these generated mutants to target integration paths in AUTs, and c) we want the generated mutants to expose deficiencies in integration test suites, not unit test suites. Thus, each RQ is based on a rationale that addresses these objectives.

The rationale for RQ1 is to determine the effectiveness of jMINT by comparing the number of generated mutants against μJava, which we selected as the baseline approach. Our goal is to show that jMINT is more effective than this baseline approach, since jMINT applies mutation operators to instructions in the AUT that are part of integration paths, while μJava applies mutation operators exhaustively.

The rationale for RQ2 is to determine if jMINT requires significantly more resources and time to analyze the AUT when compared to competitive approaches. Since dataflow analyses are computationally intensive, more resources may be required, however, our goal is to see if the demand for additional resources does not render jMINT impractical.

Finally, the rationale behind RQ3 is exploratory and its goal is to determine if jMINT produces mutants many of which result in integration errors. Ultimately, jMINT should pinpoint deficiencies in integration test suites for subject AUTs. That is, jMINT should not produce mutants of which 100% is killed using immature integration test suites and that do not provide 100% statement coverage. At the same time, we want to see if faults that result from applying mutation operators result in

integration state paths, which will show that jMINT generates mutants for which integration tests are designed and built.

B. Subject Applications And Their Test Suites

The subject applications are publicly available and widely used and their characteristics are given in Table I. These applications come with some unit and integration tests. The first three columns designate the name of the AUTs, their versions from the Sourceforge repository, and their sizes in the KLOC. The fourth column specifies the number of the state transfer paths in the AUTs and the final column gives the total number of tests.

MONGO is a Java driver application for the popular database called MongoDB. This library is readily available on GitHub with more than 150 contributors to the project. SCRIBE is popular application that is available on GitHub with over 45 contributing members who forked this project more than 700 times. SCRIBE is an OAuth library for Java based applications. JavaConsole (JCON) is an application from SourceForge that is written for system administrators with the intention to be plugged into various other applications to enable monitoring and managing various aspects of Java runtime. Nanoxml is a small non-validating parser for Java. Finally, Shtutxml is a java based tool that provides a layer to serialize XML documents.

While many open-source applications are available with unit tests, it turned out to be a difficult exercise to find applications with well-designed integration test suites. Constructing finely granular integration tests is a laborious effort that requires time and resources, which makes it difficult to justify especially in open-source where participants care more about adding new features and fixing bugs, and writing integration tests requires a collaborative effort among different programmers who wrote different components.

To enhance the original test suites for the subject applications we used more than a dozen graduate students at UIC who contributed unit test cases for our subject applications. In addition, we used these tests as seed tests for FUSION [45] to generate more integration tests. Essentially, FUSION generates integration tests by combining unit tests using an object-relational model that it re-engineers from the source code of the application. Test oracles are not generated as part of FUSION tests and exceptions are only thrown in situations where the sequence of instructions executed in the tests transforms the state of the AUT to an error state. Using FUSION has a dual positive effect, since it allows us to evaluate this new tool and show that sophisticated integration test generation tools are needed to create integration test suites.

C. Methodology And Variables

Unfortunately, there is no standard methodology for evaluating mutation testing tools. Our experimental methodology is centered on evaluating efficiency and effectiveness of jMINT. We compare jMINT with the closest comparable tool, μJava, a publicly available mutation system for Java that supports both method-level mutants and class-level mutants [38], [42]. A key

TABLE II: Comparing jMINT and μJava by the ratio of the total numbers of generated mutants, $\frac{\mu Javamutants}{jMINTmutants}$, the coupling coefficient, memory consumption and elapsed time.

AUT Name	Mut Ratio	Coup ling	Max. Memory		Max. Elap Time	
			jMINT	μJava	jMINT	μJava
MONGO	0.93	1,242	600M	115M	59s	50s
NANOXML	6.5	164	112M	133M	23s	10s
SCRIBE	2.19	162	112M	17M	27s	18s
JCON	1.61	52	132M	83M	16s	4s
SHTUTXML	18.87	16	51M	80M	12s	3s

for understanding the effectiveness of a mutation integration testing is in a measure of integratedness of components within the AUT. Suppose that all classes in the AUT are independent of one another, i.e., each class offers methods that service clients without using any methods of any other classes. In this extreme case, classes are not coupled at all, their interactions are nonexistent and there are no integration paths. If testers wrote some integration tests, which are redundant in this extreme case, μJava generates many mutants, some of which will be killed, while jMINT will generate no mutants, since these classes are not integrated in the AUT and there will be no integration points and no state transfer paths.

Conversely, suppose that all classes in the AUT are integrated, i.e., for all input values there exist an integration path that includes all classes. In this case, jMINT is likely to generate as many mutants as μJava, since modifying every instruction will result in the error state that will propagate across a class boundary. A key takeaway here is the kill/alive mutant ratios are not useful measures to estimate the effectiveness of mutation testing tools. In general, correlating the number of generated mutants by jMINT with the level of class coupling [4] shows if an AUT is a good candidate for integration testing.

Our experiments consist of applying jMINT and μJava to the subject applications to generate mutants. Then, we run integration test suites on each mutant to determine which mutants are killed by tests from these test suites. Our goal was also to compare jMINT with Bacterio [39], a mutation integration tool for Java; however, we were not able to carry out experiments with Bacterio due to its instability.

The independent variables include the mutation tools (i.e., jMINT, μJava), the set of subject applications, and the set of mutation operators. The dependent variables include the number of mutants generated, mutant killing ratio that is measured as the ratio of killed mutants to the total number of generated mutants, elapsed time required to generate the mutants and the memory requirements for the mutation tools.

D. Weak And Strong Mutation

We evaluate jMINT using separate experiments with weak (i.e., when we check that the erroneous state propagates between components) and strong (i.e., when a check is triggered by assertions in a test against an oracle to kill the mutant). We show results for the strong mutation in Table V and for the weak mutation in Table IV.

Even though our subject applications come with manually created integration tests with oracles, it is unclear how com-

TABLE III: Comparison of mutants generated by jMINT and muJava. The first column shows mutation operators and the cell values state the number of mutants/the number of strong killed mutants/the number of weak killed mutants. Mutation operator EAM changes the name of the accessor method into some other syntactically compatible method; IOD deletes the declaration of some overriding method; IPC deletes the call to the constructor `super`; JID deletes explicit initializations of class members; JTD deletes uses of the keyword `this`; JTI inserts the keyword `this`; OMD deletes overloading method declarations; OMR replaces the body of an overloading method with the body of some other method; PNC replaces the name of the instantiated class in the operator `new` with the name of some other class; PRV changes operands in assignment statements.

	SCRIBE		JCON		MONGO		SHTUTXML		NANOXML	
	jMint	μJava	jMint	μJava	jMint	μJava	jMint	μJava	jMint	μJava
EAM	183/2/6	316/33/14	29/16/16	46/8/19	232/83/134	108/53/39	0/0/0	140/72/0	42/23/20	433/151/127
IOD	0/0/0	0/0/0	0/0/0	0/0/0	7/1/1	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
IPC	0/0/0	2/0/0	0/0/0	5/0/1	8/6/5	9/5/3	0/0/0	0/0/0	0/0/0	1/0/1
JID	4/0/1	0/0/0	7/6/0	0/0/0	11/4/1	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
JTD	0/0/0	15/10/3	0/0/0	0/0/0	0/0/0	11/9/10	1/1/1	1/0/0	24/0/17	25/16/17
JTI	0/0/0	37/23/10	0/0/0	0/0/0	3/3/0	77/29/36	1/0/1	6/4/6	0/0/0	88/53/68
OMR	6/2/4	3/1/1	0/0/0	5/1/1	68/29/8	95/23/64	0/0/0	0/0/0	9/6/3	55/45/47
PNC	0/0/0	0/0/0	0/0/0	0/0/0	1/0/1	1/0/0	0/0/0	0/0/0	0/0/0	0/0/0
PRV	4/4/4	59/50/15	0/0/0	2/0/0	58/20/41	61/40/26	6/4/4	4/2/0	29/5/0	76/64/70
Total	197/8/15	432/117/43	36/22/16	58/9/21	388/146/191	362/159/178	8/5/6	151/78/6	104/34/40	678/329/330

prehensive these tests are. Since programmers created them, these tests could miss some valuable oracles and assertions. Moreover, these tests can be biased toward some components of the AUTs, thus giving us a skewed representation of the effectiveness of jMINT. On the other hand, weak mutation testing is free from this bias, since it is checked that erroneous state propagates across integrated components without needing any oracles and which is needed to answer RQ3. In the context of jMINT, the effectiveness of mutants is evaluated using both strong and weak mutation, which is based on whether the instructions containing the mutants were executed and modified the state that eventually would propagate to an integration point, thus giving us one more comparison point.

E. Threats to Validity

One threat to validity is that we generate additional integration tests using only one tool, namely, FUSION. Ideally, we need more tools to generate integration tests that contain meaningful oracles, unfortunately, little research is done in this area that resulted in tools that can be used in our evaluation.

The other threat to validity is that our approach is designed for object-oriented programs – the results may be different for procedural applications, especially those written in languages with pointers (e.g., C). It was not our goal to extend jMINT for other languages and its fundamental direction in using dataflow analysis to guide mutant generation should be generalized to other languages without significant obstacles.

A different threat to validity is a small number of subject applications that we used to evaluate jMINT, and it definitely restricts our conclusion on how generalizable the results are that we obtained in this paper. In choosing applications we were constrained by the limitations of FUSION and having sufficient test suites. We countered this threat by choosing subject applications that are popular and representative of other applications, and it is likely that choosing additional applications will not affect our results drastically.

Finally, our analysis of killed mutants can contain mistakes. The results of FUSION tests had to be analyzed manually, since not all tests that raised exceptions could be considered as ones that kill mutants. Moreover, more thorough examination is required for equivalent mutants. Our evaluation depends on the quality of the tests and it is an independent variable that is very difficult to control. To counter this threat to validity, more experimental studies should be carried out with more elaborate test suites, which is a subject of future work.

V. RESULTS

In this section, we report the results of the experiment and state how they address our RQs. The results of comparison between jMINT and μJava are shown in Table II, where the column `Gen Mut Ratio` shows $\frac{\texttt{\# of generated } \mu\texttt{Java mutants}}{\texttt{\# of generated jMINT mutants}}$, the coupling coefficient among components in each application [4] and the remaining four columns compare memory and elapsed times. A comparison between all generated mutants with the breakdown by mutation operators is shown in Table III.

The numbers of state transfer paths and the values of the coupling coefficients are strongly correlated with the numbers of generated integration mutants. We computed the value of the Pearson correlation coefficient between the numbers of integration mutants generated by jMINT and the values of the coupling coefficient, it is ≈ 0.93 thus showing a very strong correlation. It is not surprising, since the coupling among components is associated with a larger number of state transfer paths. MONGO and NANOXML have many more long state transfer paths whose lengths are measured in dozens of instructions. Yet, it is somewhat paradoxical that the mutant generation elapsed time for MONGO using jMINT is almost the same as the time it takes for μJava (i.e., 59 seconds vs 50 seconds) while it takes almost five time as much memory for jMINT. Our explanation is that once dataflow analysis is done and integration paths are identified, it takes less time to generate fewer mutants when compared to unrestricted application of mutation operators using μJava. Generating mutants involves nontrivial amount of I/O traffic (saving mutants to a persistent media) and in that it takes more resources. It is shown in Table II that the elapsed time is small in the absolute time measurements for both jMINT

and μJava. Together these results allow us to answer RQ2 positively, since **it is practical to use jMINT in that it does not require significantly more resources to generate mutants when compared to μJava**.

To answer RQ1 let us examine Table IV and Table V. These tables have an identical structure and the former table gives a breakdown for the experiment with weak mutation while the latter shows the results of the experiments with strong mutation. These tables shows the number of integration tests (i.e., generated by FUSION for weak mutation and manually created tests for strong mutation), the percentage of statement test coverage that is achieved with these tests, and the numbers of generated mutants by corresponding mutation generation tools (i.e., jMINT and μJava). For each subject applications the tables give the numbers of killed mutants and mutation killing ratios. SCRIBE contains very few long state transfer paths, and subsequently there are fewer opportunities for jMINT to generate more integration mutants.

Consider the AUT SHTUTXML for which μJava generates 151 mutant, out of which only six mutants are killed with both strong and weak mutation testing. In contrast, jMINT generates only eight mutants, out of which six and five are killed for strong and weak mutation correspondingly. Similar data is observed with other applications with the exception of MONGO. When we studied this result in depth we determined that the implementation of μJava has a defect, since it misses a number of EAM mutants. We reported this problem to the support of μJava. This result suggests that while jMINT generates fewer mutants, they are effective for targeting integration tests. It is shown in Table II that the reduction in the number of the generated mutants for jMINT is up to 19 times. In that we positively answer RQ1, since **jMINT generate fewer mutants that strongly target integration tests than μJava**.

Table III gives us some insight into the breakdown of killed mutants by mutation operators. Some mutation operators (e.g., IOD, PNC) resulted in mutants that were not killed by integration test suites, while others (e.g., EAM and OMR) demonstrated a high mutant killing power. However, we also noticed a problem that points to some limitation of jMINT.

jMINT is somewhat oblivious to the mutation operators JTD and JTI, while μJava uses them a lot to its advantage. Our explanation is the following. If the keyword `this` is used mostly to reference a variable in a method, e.g., `void m( int x) {this.x = x;}`, then μJava is biased against jMINT. Given the limitation of the dataflow analysis, jMINT has smaller chances that it recognizes a state transfer path that can be affected by resetting the value of one of the object fields in a method invocation that may occur outside this path. However, the distribution of unkilled mutants tells us that there is no obvious bias in jMINT in the way it generates mutants. Yet, the highest strong mutant killing ratio, 62.5% suggests that there is plenty of room for improvement – existing integration test suites are not sufficient to detect and kill all generation mutants. In addition, all mutants that were generated using jMINT resulted in integration errors. These results suggest that we can positively answer RQ3, since **jMINT is effective in generating mutants that result in integration errors**.

We uncovered a bug in the implementation of μJava when we noticed that jMINT sometimes generated more mutants (i.e., the entry for the AUT MONGO in Table IV and Table V. We investigated the source code of μJava and determined that among other things, μJava does not generate mutants for synchronized methods and methods of inner classes. In general, mutants generated by jMINT should be a subset of the mutants generated by μJava.

VI. Related Work

Related work to jMINT consists of two sections: research on integration testing and research for using static analysis for mutation testing. We believe that our work is the first at the intersection of these two important areas.

The idea of using static analysis to guide mutant generation has been used successfully in a number of different approaches. One of the first approaches to use program analysis for mutant generation is data flow driven mutation testing for FORTRAN programs [20]. Regression mutation testing speeds up mutation testing for evolving systems by incrementally calculating mutation testing results for the new program version based on the results from the old program version using a static analysis to check which results can be safely reused [57]. Semantic mutation testing mutates the semantics of the language to represent possible misunderstandings of the description language and thus capture a different class of faults than jMINT [10]. In a recent approach, a table of mutants is derived by control flow analysis of a disassembled binary and mutants are generated using dynamic translation [5]. Unlike jMINT, these approaches do not direct mutant generation to address integration points by using static dataflow analysis.

Symbolic execution, dynamically obtained invariants, dynamic program slicing for effective fault localization, static dataflow approaches, and evolutionary testing have been used to automate the test input generation according to different mutation testing criteria [26], [30], [34], [41], [43], [44], [54], [58]. Using mutation for integration testing was explored in interface mutation approaches that work by inducing simple changes to the interface between modules [12], [13], [18], [19]. Related to jMINT is an approach that uses control flow information along with data flow information in mutation testing for detection of bugs [3]. An approach is proposed to predict and rank error-prone connections in object-oriented systems for integration testing [2]. An integration mutation testing approach uses an idea of mutating Java library items that are heavily used in commercial software [7]. While these works provided a base for jMINT's theory and implementation, they fall short of building a fault model for integration testing and using program analyses to create fewer and more effective mutants that target integration tests.

Since a goal of jMINT is to generate fewer mutants that are more effective for integration testing, jMINT is related to multiple approaches and techniques that exist to improve the efficiency and effectiveness of generating mutant suites

TABLE IV: Comparison of killed mutants for subject applications using weak mutation testing.

AUT Name	# of Integration Tests	% Stmt Test Coverage	μJava			jMINT		
			# mutants	# Killed	% Killed	# mutants	# Killed	% Killed
MONGO	196	36.2	362	178	49.0	388	191	49.0
NANOXML	75	29.4	678	330	48.6	104	40	38.46
SCRIBE	28	11.2	432	43	10.04	197	15	7.6
JCON	24	37.2	58	21	36.0	36	16	44.0
SHTUTXML	6	2.0	151	6	3.9	8	6	75.0

TABLE V: Comparison of killed mutants for subject applications using strong mutation testing.

AUT Name	# of Integration Tests	% Stmt Test Coverage	μJava			jMINT		
			# mutants	# Killed	% Killed	# mutants	# Killed	% Killed
MONGO	599	10.6	362	159	43.9	388	146	37.6
NANOXML	116	61.7	678	329	48.5	104	34	32.69
SCRIBE	100	28.1	432	117	27.0	197	8	4.0
JCON	32	81	58	9	15.5	36	22	62.11
SHTUTXML	43	59.2	151	78	51.6	8	5	62.5

while reducing their sizes and making them more efficient. Selective mutation, for example, reduces the number of mutants by applying only a subset of mutation operators [21], by prioritizing and reducing tests to more quickly determine the sets of killed and non-killed mutants [56], by reducing the number of mutation operators and reducing the sections of the code where they are applied [23], by identifying hierarchies among mutants [32], by using probabilistic sampling methods to prioritize operators whose mutants are likely to remain unexposed by the existing test suites [51] and by using the mutation score to select test cases while preserving the quality of the suite and reducing the number of generated mutants [46]. A main difference between jMINT and these broad-spectrum efficiency improving approaches for mutation testing lies in our use of dataflow analysis to select integration paths to apply mutation operators.

VII. Conclusion

We created *Java Mutation Integration Testing (jMINT)* to generate mutants that specifically target integration tests. We formulated a fault model for integration bugs. jMINT generates mutants by applying mutation operators to dataflow paths through which components exchange data. We evaluate jMINT on five open-source applications and compare it with μJava, a publicly available mutation tool for Java. Our evaluation shows that even though jMINT takes approximately five times more memory in the worse case, it leads to reduction of the number of generated mutants by up to 19 times with a strong power to determine inadequacies in integration test suites.

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[2] S. E. Bani Ta'An. *Towards Test Focus Selection for Integration Testing Using Software Metrics*. PhD thesis, Fargo, ND, USA, 2013. AAI3561106.

[3] M. B. Bashir and A. Nadeem. Control oriented mutation testing for detection of potential software bugs. FIT '12, pages 35–40, Washington, DC, USA, 2012. IEEE Computer Society.

[4] F. Beck and S. Diehl. On the congruence of modularity and code coupling. ESEC/FSE '11, pages 354–364, New York, NY, USA, 2011. ACM.

[5] M. Becker, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller. Binary mutation testing through dynamic translation. DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[6] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[7] J. M. Bieman, S. Ghosh, and R. T. Alexander. A technique for mutation of java objects. ASE '01, pages 337–, Washington, DC, USA, 2001. IEEE Computer Society.

[8] S.-k. Cheong, K.-h. Lee, and T.-w. Jeong. The analysis of integration test result for atm switching systems. In *Selected proceedings of the IFIP TC6 9th international workshop on Testing of communicating systems*, pages 83–89, London, UK, UK, 1996. Chapman & Hall, Ltd.

[9] C. Choppy. Formal specifications, prototyping and integration tests. In *Proceedings of the 1st European Software Engineering Conference*, pages 172–179, London, UK, 1987. Springer-Verlag.

[10] J. A. Clark, H. Dan, and R. M. Hierons. Semantic mutation testing. *Sci. Comput. Program.*, 78(4):345–363, Apr. 2013.

[11] P. J. Clarke. *A taxonomy of classes to support integration testing and the mapping of implementation-based testing techniques to classes*. PhD thesis, Clemson, SC, USA, 2003. AAI3098273.

[12] M. E. Delamaro and J. C. Maldonado. Interface mutation: Assessing testing quality at interprocedural level. SCCC '99, pages 78–, Washington, DC, USA, 1999. IEEE Computer Society.

[13] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Trans. Softw. Eng.*, 27(3):228–247, Mar. 2001.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[15] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. ICSE '02, pages 241–251, New York, NY, USA, 2002. ACM.

[16] M. Ellims, J. Bridges, and D. C. Ince. The economics of unit testing. *Empirical Softw. Engg.*, 11(1):5–31, Mar. 2006.

[17] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12(6):17–26, Nov. 1995.

[18] S. Ghosh and A. P. Mathur. Interface mutation to assess the adequacy of tests for components and systems. TOOLS '00, pages 37–, Washington, DC, USA, 2000. IEEE Computer Society.

[19] S. Ghosh and A. P. Mathur. Mutation testing for the new century. chapter Inferface Mutation (Abstract Only), pages 90–. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[20] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. ICSE '85, pages 313–319, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[21] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. ISSTA 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[22] B. J. M. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. ICSTW '09, pages 192–199, Washington, DC, USA, 2009. IEEE Computer Society.

[23] A. Haley and S. Zweben. Development and application of a white box approach to integration testing. *J. Syst. Softw.*, 4(4):309–315, Nov. 1984.

[24] M. Hamill and K. Goseva-Popstojanova. Common trends in software fault and failure data. *IEEE Trans. Softw. Eng.*, 35(4):484–496, July 2009.

[25] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[26] M. Harman, R. Hierons, and S. Danicic. Mutation testing for the new century. chapter The Relationship Between Program Dependence and Mutation Analysis, pages 5–13. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[27] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *SVTR*, 9:233–262, 1999.

[28] J. Hu, N. Li, and J. Offutt. An analysis of oo mutation operators. ICSTW '11, pages 334–341, Washington, DC, USA, 2011. IEEE Computer Society.

[29] C. Jones and O. Bonsignour. *The Economics of Software Quality*. Addison-Wesley Professional, Aug. 2011.

[30] S. Kakarla, S. Momotaz, and A. S. Namin. An evaluation of mutation and data-flow testing: A meta-analysis. ICSTW '11, pages 366–375, Washington, DC, USA, 2011. IEEE Computer Society.

[31] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, Jan. 1976.

[32] K. Kapoor and J. P. Bowen. Ordering mutants to minimise test effort in mutation testing. FATES'04, pages 195–209, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] S. Kiteley and J. Draper. Results of an investigation into software integration testing automation. Ada-Europe '00, pages 280–290, London, UK, UK, 2000. Springer-Verlag.

[34] X. Liu, G. Xu, X. Fu, and Y. Dong. Test data generation considering data dependence. FCST '10, pages 208–213, Washington, DC, USA, 2010. IEEE Computer Society.

[35] Y.-S. Ma, M. J. Harrold, and Y.-R. Kwon. Evaluation of mutation testing for object-oriented programs. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 869–872, New York, NY, USA, 2006. ACM.

[36] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

[37] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: An automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, June 2005.

[38] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. Mujava: A mutation system for java. ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM.

[39] P. R. Mateo, M. P. Usaola, and J. Offutt. Mutation at system and functional levels. ICSTW '10, pages 110–119, Washington, DC, USA, 2010. IEEE Computer Society.

[40] A. P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008.

[41] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Softw. Pract. Exper.*, 26(2):165–176, Feb. 1996.

[42] J. Offutt, Y.-S. Ma, and Y.-R. Kwon. An experimental mutation system for java. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, Sept. 2004.

[43] M. Papadakis and N. Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Control*, 19(4):691–723, Dec. 2011.

[44] M. Papadakis, N. Malevris, and M. Kallia. Towards automating the generation of mutation tests. AST '10, pages 111–118, New York, NY, USA, 2010. ACM.

[45] M. Pezze, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Proceedings of the 2013 IEEE ICST*, ICST '13, pages 11–20, Washington, DC, USA, 2013. IEEE Computer Society.

[46] M. Polo Usaola, P. Reales Mateo, and B. Pérez Lamancha. Reduction of test suites using mutation. FASE'12, pages 425–438, Berlin, Heidelberg, 2012. Springer-Verlag.

[47] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[48] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. ESEC/FSE '09, pages 297–298, New York, NY, USA, 2009. ACM.

[49] Y. Shin, Y. Choi, and W. J. Lee. Integration testing through reusing representative unit test cases for high-confidence medical software. *Comput. Biol. Med.*, 43(5):434–443, June 2013.

[50] B. H. Smith and L. Williams. An empirical evaluation of the mujava mutation operators. TAICPART-MUTATION '07, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.

[51] M. Sridharan and A. S. Namin. Prioritizing mutation operators based on importance sampling. ISSRE '10, pages 378–387, Washington, DC, USA, 2010. IEEE Computer Society.

[52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[53] A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, NIST: National Institute of Standards and Technology, Gaithersburg, MD, 1996. See http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm.

[54] F. Wedyan. *Testing with State Variable Data-flow Criteria for Aspect-oriented Programs*. PhD thesis, Fort Collins, CO, USA, 2011. AAI3468815.

[55] M. Welsh. What i wish systems researchers would work on. *http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html*, May 2013.

[56] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. ISSTA 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[57] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. ISSTA 2012, pages 331–341, New York, NY, USA, 2012. ACM.

[58] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.