

Test Generation via Dynamic Symbolic Execution for Mutation Testing

Lingming Zhang*, Tao Xie[†], Lu Zhang*, Nikolai Tillmann[‡], Jonathan de Halleux[‡], Hong Mei*

*Key Laboratory of High Confidence Software Technologies, Ministry of Education

Institute of Software, Peking University, Beijing, 100871, P. R. China

Email: {zhanglm07, zhanglu, meih}@sei.pku.edu.cn

[†]Department of Computer Science, North Carolina State University, Raleigh, NC27695

Email: xie@csc.ncsu.edu

[‡]Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA

Email: {nikolait, jhalleux}@microsoft.com

Abstract—Mutation testing has been used to assess and improve the quality of test inputs. Generating test inputs to achieve high mutant-killing ratios is important in mutation testing. However, existing test-generation techniques do not provide effective support for killing mutants in mutation testing. In this paper, we propose a general test-generation approach, called PexMutator, for mutation testing using Dynamic Symbolic Execution (DSE), a recent effective test-generation technique. Based on a set of transformation rules, PexMutator transforms a program under test to an instrumented meta-program that contains mutant-killing constraints. Then PexMutator uses DSE to generate test inputs for the meta-program. The mutant-killing constraints introduced via instrumentation guide DSE to generate test inputs to kill mutants automatically. We have implemented our approach as an extension for Pex, an automatic structural testing tool developed at Microsoft Research. Our preliminary experimental study shows that our approach is able to strongly kill more than 80% of all the mutants for the five studied subjects. In addition, PexMutator is able to outperform Pex, a state-of-the-art test-generation tool, in term of strong mutant killing while achieving the same code coverage.

I. INTRODUCTION

Regression testing is often time consuming, and accounts for approximately one half of the cost in software maintenance [27], [36]. In regression testing, generating high-quality test inputs for regression test suites is crucial. Although manually written test inputs are valuable, they are often insufficient in guarding against regression faults. Therefore, automatic test generation has been used to generate test inputs to complement manually written test inputs.

In the literature, various automatic techniques have been proposed for test generation: some techniques generate test inputs randomly [6], [7], [13]; some techniques use code coverage as the test criterion and generate test inputs to satisfy the test criterion [5], [26], [29], [30]; some techniques use mutation testing as the test criterion and generate test inputs to kill mutants [10], [19], [22]. Note that using mutant killing to guide automatic test generation has been shown to be intractable [10]. Therefore, existing techniques use the concept of weak mutation testing [17] to guide test generation, and thus improve the probability of mutant killing (detailed information about weak mutation testing is shown in Section II-A).

Among various test criteria aimed by automatic test-generation techniques, mutation testing has been shown to be an effective indicator for the quality of test inputs [2]. In addition, mutation testing has been shown to be superior to common code coverage in evaluating the effectiveness of test inputs [12]. However, existing mutation-testing techniques face various challenges in practice. One of the main challenges is automatic test generation for mutant killing. Although there are techniques [10], [19], [22] aiming to facilitate the process of test generation towards mutant killing, they have two main limitations. First, these techniques construct a whole constraint system for each weak mutant killing¹, making it costly to generate test inputs for a large number of mutants. Second, these techniques are based on solving statically constructed constraint systems, not being able to handle programs with complex data structures, non-linear arithmetic, or array indexing with non-constant expressions. These limitations cause the existing mutant-killing-based test-generation techniques not to be applicable for real-world programs, and not to be widely used in practice.

In this paper, we propose a general test-generation approach for mutant killing using dynamic symbolic execution (DSE), a recent effective test-generation technique, and implement it as a tool called *PexMutator*. In a typical application, PexMutator first transforms the original program under test into a *meta-program*, which contains all the weak-mutant-killing constraints inserted via instrumentation. More specifically, each constraint instrumented into the meta-program is wrapped as a conditional statement, whose execution takes the true or false branch according to whether the constraint is satisfied or not. Then, PexMutator uses Pex, a state-of-the-art DSE engine, to generate test inputs for the meta-program. The DSE engine generates test inputs to try to cover all the branches in the meta-program. Test inputs that cover the true branches of instrumented conditional statements would satisfy the weak-mutant-killing constraints, and thus weakly kill the corresponding mutants. Therefore, the introduced weak-mutant-

¹Existing techniques conjunct one weak-mutant-killing constraint with the constraint system of the original program to form a whole constraint system, and then solve the whole system to weakly kill a mutant.

killing constraints in the meta-program guide the DSE engine to satisfy them and thus generate test inputs to kill the mutants automatically.

This paper makes the following main contributions:

- We propose a general approach that automatically generates test inputs to kill mutants via DSE. Our approach generates a meta-program containing all the weak-mutant-killing constraints, instead of the costly traditional way of conjuncting one weak-mutant-killing constraint with the original program to form a constraint system one at a time. Furthermore, our approach generates test inputs via DSE, which is more effective than previous test-generation techniques for mutant killing in dealing with complex programs.
- We implement the proposed technique in a tool called PexMutator as an extension for Pex [30], an automated structural testing tool for .NET developed at Microsoft Research. PexMutator has been released as open source in our Pex Extensions project webpage².
- We conduct a preliminary experimental study and show that our approach is able to strongly kill more than 80% of all the mutants for studied subjects, and sometimes even achieves a 100% killing ratio for non-equivalent mutants. Furthermore, the experimental study also shows that PexMutator is able to strongly kill more mutants than Pex, a state-of-the-art test-generation technique, while achieving the same code coverage.

The rest of this paper is organized as follows. Section II introduces background of mutation testing and dynamic symbolic execution. Section III illustrates our approach with examples. Section IV presents the detailed implementation and the current status of PexMutator. Section V reports our experimental study. Section VI presents related work. Section VII concludes with future work.

II. BACKGROUND

A. Mutation Testing

Mutation testing [9], [15] is an intensively studied technique for assessing and improving the quality of test inputs. In mutation testing, the original program under test is mutated to various faulty versions (known as *mutants*). Each mutant includes one fault automatically seeded based on a set of *mutation operators*, each of which is a rule for mutating a statement of the original program into a faulty statement. The statement in which mutation takes place is called a *mutation point*. Since its first proposal, mutation testing has been a popular topic in software testing. Some researchers investigated techniques to overcome challenges of using mutation testing to assess the effectiveness of existing test suites [1], [4], [14], [16], [21], [28], [33]–[35]. Some researchers used mutation testing to automatically generate faulty versions to construct subjects for software-testing experimentation [2], [3], [11], [20], [32]. Some researchers investigated automatic techniques for generating test inputs to kill mutants [10], [19], [22].

We next illustrate one key notion in mutation testing: *mutant killing*. In mutation testing, all the mutants are executed against a set of test inputs. The mutants whose executions produce different results (i.e., different final states) from those of the original program are denoted as *killed* mutants. There might be mutants that do not change the program’s overall semantics, and thus cannot be killed by any test inputs. These mutants are called *equivalent* mutants. The quality of test inputs is reflected by the number of mutants killed by the test inputs: the more mutants killed, the more effective the test inputs are. In order to kill a mutant, a test input should satisfy the following three criteria [10], [24] (the program and its test suite are denoted as P and T , respectively, and a mutant of P on statement S is denoted as M):

Reachability: Mutant M is the same with program P except the mutated statement S . Therefore, if S is not executed by a test input t ($t \in T$), the execution on M will produce the same result as that of P . That is, for any t ($t \in T$), if S is not reached by t , t is assured not to kill M .

Necessity: For a test input t ($t \in T$) to kill mutant M , t must cause different internal states on P and M immediately after executing S . Otherwise, since all the other parts of M and P are exactly the same, there will not be any different states between P and M during the execution, and their final results will be the same as well.

Sufficiency: For a test input t ($t \in T$) to kill mutant M , t must lead to different final states for M and P . That is, the different internal states caused by satisfying the necessity criterion must be propagated through the program’s execution to the final state and yield different results.

Due to the expensiveness of mutation testing, Howden et al. [17] proposed the concept of *weak mutation testing*. According to weak mutation testing, test inputs that satisfy the reachability and necessity criteria are denoted as satisfying the weak-mutation-testing criterion. As it has been shown to be intractable to automatically generate test inputs that definitely satisfy the sufficiency criterion [10], existing techniques mainly adopt the concept of weak mutation testing and use the reachability and necessity criteria to generate test inputs. That is to say, the generated test inputs do not guarantee to kill mutants, but improve the probability to kill mutants. In this paper, our approach also adopts the reachability and necessity criteria to generate test inputs. In summary, there are two types of mutant killing:

Definition 2.1: A test input t *weakly kills* a mutant M , iff t satisfies the reachability and necessity criteria in mutation testing.

Definition 2.2: A test input t *strongly kills* a mutant M , iff t satisfies the reachability, necessity, and sufficiency criteria in mutation testing.

B. Dynamic Symbolic Execution (DSE)

DSE is a technique used to automatically generate test inputs that achieve high code coverage. DSE executes the program under test for some given test inputs (e.g., ones generated randomly), and at the same time performs symbolic

²<http://pexase.codeplex.com/>

execution [8], [18] in parallel to collect symbolic constraints obtained from predicates in branch statements along the execution traces. The conjunction of all symbolic constraints along a path is called a *path condition*. In test generation, DSE is performed iteratively on the program under test to increase code coverage. Initially, DSE randomly chooses one test input from the input domain. Then, in each iteration, after running each test input, DSE collects the path condition of the execution trace, and uses a search strategy to flip a branching node in the path. Flipping a branching node in a path constructs a new path that shares the prefix to the node with the old path, but then deviates and takes a different path. Whether such a flipped path is feasible is checked by building a constraint system. If a constraint solver can determine that the constraint system is satisfiable within the available resources, DSE generates a new test input that will execute along the flipped path and achieve additional code coverage. In this way, DSE is able to generate a set of test inputs that achieve high code coverage. DSE has been intensively studied and various practical techniques and tools has been implemented (e.g., EXE [5], CUTE [29], and Pex [30], [31]).

III. FRAMEWORK OF PEXMUTATOR

PexMutator includes four main steps. First, PexMutator generates mutants for the program under test. Second, PexMutator generates corresponding weak-mutant-killing constraints for each mutant. Third, PexMutator inserts all the generated constraints to certain positions of the original program to form a meta-program. Finally, PexMutator uses a DSE engine to generate test inputs for the meta-program, so that the generated test inputs will be able to satisfy the weak-mutant-killing constraints and thus weakly kill mutants. Next we illustrate the detailed design of PexMutator.

A. Mutant Generation

Mutation-testing techniques generate mutants based on a set of mutation operators. Each mutation operator defines a rule for transforming the original program under test to mutated programs, each of which contains a seeded fault. Researchers have realized that a large set of mutation operators may cause to generate too many mutants. The large number of mutants may exhaust time or space resources without providing comparable benefits. Therefore, researchers start to find subsets of mutation operators that can achieve approximately the same effectiveness in indicating the quality of test inputs. Offutt et al. [23], [25] found that five mutation operators (i.e., ABS, AOR, ROR, LCR and UOI) are approximately as effective as all the 22 mutation operators of Mothra, a mutation-testing tool. These five mutation operators are denoted as *sufficient mutation operators*.

To generate test inputs efficiently, PexMutator uses the five sufficient mutation operators in generating mutants. The details of the five mutation operators are shown in Table I. In Table I, $op1$, $op2$ denote any expressions that appear in the program under test; α , β denote binary operators, e.g., they could be relational operators (denoted as *RO*), arithmetic operators

TABLE I
SUFFICIENT MUTATION OPERATORS USED BY PEXMUTATOR.

Mutation Operators	Mutation Rules
ABS (Absolute Value Insertion)	$op1 \rightarrow \forall \nu, \nu \in V, \nu$
AOR (Arithmetic Operator Replace)	$\forall \alpha, \alpha \in AO, op1 \alpha op2 \rightarrow \forall \beta, (\beta \in AO) \wedge (\beta! = \alpha), op1 \beta op2$
LCR (Logical Connector Replace)	$\forall \alpha, \alpha \in LC, op1 \alpha op2 \rightarrow \forall \beta, (\beta \in LC) \wedge (\beta! = \alpha), op1 \beta op2$
ROR (Relational Operator Replace)	$\forall \alpha, \alpha \in RO, op1 \alpha op2 \rightarrow \forall \beta, (\beta \in RO) \wedge (\beta! = \alpha), op1 \beta op2$
UOI (Unary Operator Insertion)	$op1 \rightarrow \forall \mu, \mu \in UO, \mu (op1)$

TABLE II
CONSTRAINT-GENERATION RULES.

Mutation Operators	Constraint-Generation Rules
ABS	$op1 \rightarrow \forall \nu, \nu \in V, op1 != \nu$
AOR	$\forall \alpha, \alpha \in AO, op1 \alpha op2 \rightarrow \forall \beta, (\beta \in AO) \wedge (\beta! = \alpha), (op1 \alpha op2) != (op1 \beta op2)$
LCR	$\forall \alpha, \alpha \in LC, op1 \alpha op2 \rightarrow \forall \beta, (\beta \in LC) \wedge (\beta! = \alpha), (op1 \alpha op2) != (op1 \beta op2)$
ROR	$\forall \alpha, \alpha \in RO, op1 \alpha op2 \rightarrow \forall \beta, (\beta \in RO) \wedge (\beta! = \alpha), (!(op1 \alpha op2) \wedge (op1 \beta op2)) \vee ((op1 \alpha op2) \wedge !(op1 \beta op2))$
UOI	$op1 \rightarrow \forall \mu, \mu \in UO, op1 != \mu (op1)$

(denoted as *AO*), or logical connectors (denoted as *LC*); μ denotes a unary operator (denoted as *UO*); ν denotes a absolute value inserted within domain V . For example, the AOR (which denotes Arithmetic Operator Replace) operator defines a rule for transforming an arithmetic operator into another one with the other part of the program unchanged. Assume that we have a program p with a statement s : $sum = op1 + op2$, the AOR operator generates four mutants for p with s mutated to $sum = op1 - op2$, $sum = op1 * op2$, $sum = op1 / op2$, and $sum = op1 \% op2$, respectively.

B. Mutant-Killing-Constraint Generation

After generating mutants for the program under test, PexMutator constructs corresponding mutant-killing constraints. In mutation-testing-based test generation, solving strong-mutant-killing constraints has been shown to be intractable [10]. Therefore, PexMutator constructs weak-mutant-killing constraints to guide test generation, i.e., PexMutator generates constraints satisfying which guarantees the generated test inputs to satisfy the reachability and necessity criteria, and weakly kill mutants.

For basic expressions, such as $op1 > op2$, in which $op1$ and $op2$ denote two operands for the relational operator $>$, PexMutator generates a constraint for each mutant of the expression. The rules for generating constraints for basic expressions are shown in Table II. In the table, the used symbols are defined in the same way with those of Table I. For mutants generated by the ABS operator, PexMutator generates constraints asserting that the inserted values are different from the values of the original expressions; for mutants generated by the AOR, LCR, and UOI operators, PexMutator generates constraints asserting that the mutated

expressions derive different values from the original ones; for mutants generated by the ROR operator, PexMutator generates constraints asserting that the mutated conditional expressions have opposite values with the original ones. For example, $op1 > op2$ has a corresponding mutated expression $op1 \geq op2$, then PexMutator generates the following constraints for the this mutant: $((op1 > op2) \wedge !(op1 \geq op2)) \vee (!(op1 > op2) \wedge (op1 \geq op2))$.

For complex expressions that contain sub-expressions, such as $(op1 > op2) \&\& (op3 > op4)$, PexMutator uses the Depth-First-Search algorithm to generate mutant-killing constraints for each mutation point. The pseudo code for the constraint-generation process is shown in Figure 1. As shown in the figure, method `GenCompConsts` generates constraints for expression exp , and returns $consts$ as the generation result. Line 1 initializes $consts$ as an empty set. Line 2 checks whether the input expression exp is in set `Terminals` (which denotes a set of expressions that have no sub-expressions and will not be mutated). If so, the algorithm returns to the upper recursive level. Line 5 invokes method `GenConsts` and generates mutant-killing constraints for exp at its outmost level (i.e., treating exp as a basic expression, and ignoring mutants in its sub-expression(s)). Line 6 decides whether exp is an unary operation. If so, PexMutator obtains its sub-expression and recursively generates mutant-killing constraints for its sub-expression. Line 10 decides whether exp is a binary operation. If so, PexMutator obtains both its right and left sub-expressions and recursively generates mutant-killing constraints for its sub-expressions. Finally, Line 16 returns the set of constraints $consts$ as the generation result. For example, when we use PexMutator to generate constraints for $(op1 > op2) \&\& (op3 > op4)$, PexMutator first treats $op1 > op2$ and $op3 > op4$ as terminal variables and generates constraints at the expression’s outmost level; then PexMutator recursively generates constraints for $op1 > op2$ and $op3 > op4$, respectively.

C. Mutant-Killing-Constraint Insertion

Before inserting the generated constraints into the program under test, PexMutator wraps the constraints up as executable statements. In addition, PexMutator should insert the generated constraints into proper positions of the original program under test. Otherwise, the added constraints may cause syntactic faults or derive different states from the corresponding mutation points. As the constraint insertions for mutants of conditional statements and non-conditional statements are different, here we next depict them separately.

1) *Constraint Insertion for Conditional Statements:* For a conditional statement with a corresponding set of generated constraints $consts$, PexMutator wraps each $const \in consts$ as a conditional statement: `if(const) log.write("Mutant Killed");`. The added log-writing statement has two benefits: first, the statement introduces a new branch, enabling the DSE technique to generate test inputs to cover the branch, thus satisfying the mutant-killing

Algorithm *GenCompConsts(exp)*

Input: A complex expression exp .

Output: A set of mutant-killing constraints for the input expression $consts$

Begin

```

1:  $consts \leftarrow \emptyset$ 
2: if  $exp \in Terminals$  then
3:   goto Line 16
4: end if
5:  $consts \leftarrow consts \cup GenConsts(exp)$ 
6: if  $exp \in UnaryOperations$  then
7:   Expression  $sExp \leftarrow exp.subOperand$ 
8:    $consts \leftarrow consts \cup GenCompConsts(sExp)$ 
9: end if
10: if  $exp \in BinaryOperations$  then
11:   Expression  $rExp \leftarrow exp.leftOperand$ 
12:   Expression  $lExp \leftarrow exp.rightOperand$ 
13:    $consts \leftarrow consts \cup GenCompConsts(rExp)$ 
14:    $consts \leftarrow consts \cup GenCompConsts(lExp)$ 
15: end if
16: return  $consts$ 
End

```

Fig. 1. Generating constraints for complex expressions

constraint; second, the statement records the corresponding weak mutant-killing information for further analysis.

The wrapped mutant-killing constraints for conditional statements should be inserted to positions near mutation points, so that they can reflect states of the mutation points. For conditional statements, there are three positions near a mutation point: the position before both the mutation point and the subsequent branch, the position between the mutation point and the subsequent branch, and the position after the mutation point and of the first place in the subsequent branch. For the convenient of illustration, we denote the three positions as “before”, “between”, and “after”, respectively. PexMutator inserts constraints to the “before” position, and the rule for inserting multiple constraints to the “before” position of a conditional statement is: `for(constSta in ConstStas) S \leftarrow {constSta;S}`, where S is the mutated statement and `ConstStas` is the corresponding set of mutant-killing statements (generated by wrapping mutant-killing constraints) of S . PexMutator does not insert constraints to the “between” and “after” positions, because inserting constraints into the “between” position causes syntactic faults and inserting constraints into the “after” position could cause the variables in the constraints to have different domains from the corresponding variables in the mutation points.

For example, assume that we have a program under test p depicted below:

```

read(a);
if (a >= 0)
{ do something; }

```

Further assume that we have a mutant of p , which mutates `if (a >= 0)` to `if (a > 0)`, and then the corresponding mutant-killing constraint generated for this mu-

position "before"	position "between"	position "after"
<pre>read (a); if (((a >= 0) && !(a > 0)) (!(a >= 0) && (a > 0))) { log.write("Mutant Killed"); } if (a >= 0) { do something; }</pre>	<pre>read (a); if (a >= 0) if (((a >= 0) && !(a > 0)) (!(a >= 0) && (a > 0))) { log.write("Mutant Killed"); } { do something; }</pre>	<pre>read (a); if (a >= 0) { if (((a >= 0) && !(a > 0)) (!(a >= 0) && (a > 0))) { log.write("Mutant Killed"); } do something; }</pre>

Fig. 2. Inserting constraints for a conditional statement

position "before"	position "after"
<pre>read (a); read (b); if (false (a+b) != (a-b)) { log.write("Mutant Killed!"); } b = a + b; return b;</pre>	<pre>read (a); read (b); b = a + b; if (false (a+b) != (a-b)) { log.write("Mutant Killed!"); } return b;</pre>

Fig. 3. Inserting constraints for a non-conditional statement

tants is $((a \geq 0) \wedge !(a > 0)) \vee (!(a \geq 0) \wedge (a > 0))$. Before inserting the constraint into the program under test, PexMutator first wraps the constraint up as bellow:

```
if(((a >= 0) && !(a > 0)) || (!(a >= 0) && (a > 0)))
{ log.write("Mutant Killed"); }
```

As shown in Figure 2, there are three possible positions for inserting the constraint into program p . It is obvious that inserting the constraint to the "between" position causes a syntactic fault (shown in the second column), while inserting the constraint to the "after" position causes variable a in the constraint to have a different domain with variable a at the mutation point (shown in the third column). Therefore, PexMutator inserts the constraint into the "before" position, as shown in the first column of Figure 2.

2) Constraint Insertion for Non-Conditional Statements:

For a non-conditional statement with a corresponding set of constraints $consts$, wrapping constraints is more complex, because mutated expressions may throw exceptions, causing the subsequent part of the generated meta-program not to be able to execute. For example, a constraint generated for statement $sum = op1 + op2$ is $((op1 + op2) \neq (op1 / op2))$. If we just wrap the constraint following the process of wrapping constraints for conditional statements, the wrapping result would be

```
if ((op1 + op2) != (op1 / op2))
{ log.write("Mutant Killed"); }
```

We can find that if $op2$ is assigned with 0, the subsequent part of the generated meta-program will never be executed. To avoid this problem, for a non-conditional statement with a set of constraints (which are generated according to mutants of the statement) $consts$, PexMutator separates

exception-triggering conditions from the constraints and wraps $const$ ($const \in consts$) as: `if(trigger || const) log.write("Mutant Killed");`, where `trigger` denotes the exception-triggering condition for $const$. Note that when there is no exception-triggering condition, `trigger` is assigned with `false`. The conditional expression `if(trigger || const)` does not execute `const` if the trigger is satisfied, preventing the meta-program from throwing exceptions. For example, the exception-triggering condition for constraint $((op1 + op2) \neq (op1 / op2))$ is $op2 == 0$. PexMutator generates the following conditional statement for the constraint:

```
if((op2 == 0) || ((op1 + op2) != (op1 / op2)))
{ log.write("Mutant Killed"); }
```

When $op2$ is 0, PexMutator detects that the mutant is weakly killed without throwing exceptions. The added log-writing statement has similar benefits with that of a conditional statement.

Similar with conditional statements, mutant-killing constraints for non-conditional statements should also be inserted to positions near the corresponding mutation points. For non-conditional statements, there are two positions near a mutation point: the position right before the mutation point, and the position right after the mutation point. For the convenient of illustration, we denote these two positions as "before" and "after", respectively. PexMutator inserts constraints to the "before" position, and the rule for inserting multiple constraints to the "before" position of a non-conditional statement is the same with that of a conditional statement: $for(constSta \text{ in } ConstStas) S \leftarrow \{constSta; S\}$, where S is the mutated statement and $ConstStas$ is the corresponding set of mutant-killing statements (generated by wrapping mutant-killing constraints) of S . PexMutator does not insert constraints to the "after" position because inserting constraints into the "after" position could cause variables in the constraints to have different values or states with the corresponding variables in the mutation points, since the mutation points may change the values or states of certain variables in the constraints (as illustrated by the example in Figure 3).

D. Test-Generation for the Meta-Program Using DSE

After transforming the original program under test to a meta-program containing all mutant-killing constraints, PexMutator uses the DSE engine to generate test inputs for the meta-program. The DSE engine tries to cover all the possible paths, making the generated test inputs satisfy all the possible mutant-killing constraints, thus weakly killing the mutants. Assume that we have a program under test as below:

```
void foo(int a, int b) {
  if(a > 10)
    return a-b;
  else
    return a;
}
```

There are two mutation points in statements `if(a > 10)` and `return a-b`. Normally, PexMutator generates mutant-killing constraints for every mutant and inserts them into the original

program under test to obtain a meta-program. Here to simplify the illustration, we use three mutants of each mutation point to generate the meta-program, the generated meta-program is shown below (the added conditional statements 1-3 are generated by mutant-killing constraints `const` 1-3 of the first mutation point, and the added conditional statements 4-6 are generated by mutant-killing constraints `const` 4-6 of the second mutation point):

```
void foo(int a) {
  if(((a>10) && !(a>=10)) || (!(a>10) && (a>=10))) //const1
    log.write("mutant a>=10 killed");
  if(((a>10) && !(a==10)) || (!(a>10) && (a==10))) //const2
    log.write("mutant a==10 killed");
  if(((a>10) && !(a!=10)) || (!(a>10) && (a!=10))) //const3
    log.write("mutant a!=10 killed");
  if(a > 10) {
    if(false || (a-b)!=(a+b)) //const4
      log.write("mutant a+b killed");
    if(false || (a-b)!(a*b)) //const5
      log.write("mutant a*b killed");
    if((b==0) || (a-b)!(a/b)) //const6
      log.write("mutant a/b killed");
    return a-b;
  } else
    return a;
}
```

After the meta-program is available, PexMutator uses the DSE engine to generate test inputs for the meta-program. As shown in Section II-B, the DSE engine generates test inputs to cover all the possible branches of the meta-program. The generated test inputs cover the true branches of the added conditional statements, thus satisfying mutant-killing constraints and weakly killing mutants. For example, the test inputs generated by the DSE engine for the preceding meta-program cover the true branches of the six instrumented conditional statements, satisfying `const` 1-6 and weakly killing the six mutants. As all the added mutant-killing constraints do not change the state of the original program, the generated meta-program has the same results with the original program if they are provided with the same inputs. Therefore, the generated test inputs for the meta-program can be directly used for the original program under test to kill mutants.

IV. IMPLEMENTATION

We have implemented PexMutator as a tool for generating test inputs for .NET applications. To test a program, PexMutator first analyzes the program and transforms the program to a meta-program based on the transformation rules proposed in Section III, and then uses the DSE engine to generate test inputs for the meta-program toward mutant killing. Next we briefly illustrate the implementation of PexMutator including two steps:

1) *Meta-program Generation*: Meta-program generation is the key part of PexMutator. Typically, there are two ways to generate a meta-program: generating at the source level and generating at the compiled-file level. Manipulating source code could be tedious and fault-prone. In addition, the generated meta-program is not immediately available for test generation. Therefore, PexMutator generate the meta-program at the compiled-file level directly. PexMutator is developed using the

*Common Compiler Infrastructure*³, which provides functionality for reading, writing, and manipulating *Microsoft Common Language Runtime* (CLR) assemblies using a hierarchical object model. In the implementation, PexMutator reads the assemblies of the original program under test and generates the corresponding meta-program in the form of assemblies. The code manipulation based on CCI makes the generated meta-program less fault-prone. In addition, the generated meta-program can be directly used for test generation.

2) *Test Generation via DSE*: As a state-of-the-art DSE engine, Pex [30] has been previously used internally at Microsoft to test core components of the .NET architecture and has found serious faults [30]. Moreover, Pex has been used increasingly widely in both industry and academia. Therefore, PexMutator adopts Pex as its test-generation engine. In testing a program, PexMutator first transforms the original program to a meta-program, and then uses the Pex engine to generate test inputs for the meta-program. The generated test inputs in the form of C# unit tests are able to kill mutants automatically.

Currently, PexMutation 1.1 has been released as open source in the Pex Extensions site⁴. In the near future, we plan to integrate PexMutator as a new test-generation option within Pex.

V. EXPERIMENTAL STUDY

In our study, we intend to investigate the following research questions:

- How does PexMutator perform in generating test inputs to kill mutants?
- How does PexMutator compare with Pex in term of code coverage and mutant killing?

The first research question is mainly concerned with the effectiveness and efficiency of our approach in practice. We further investigate the second research question for the following reasons: first, PexMutator uses Pex to generate test inputs, and therefore we want to evaluate the improvement of PexMutator over Pex in term of mutant killing; second, we want to compare the effectiveness of test inputs generated by PexMutator and a state-of-the-art test-generation technique (e.g., Pex). Here we evaluate the effectiveness of test inputs in terms of both code coverage and mutant killing.

A. Experimental Setup

We conducted our experiments on a PC with a 2.66GHz Intel Pentium4 CPU and 2GB memory running the Windows XP operating system. In the rest of this sub-section, we present the subject programs (Section V-A1), supporting tools (Section V-A2), and experimental procedure (Section V-A3) of our experimental study.

1) *Subjects: Data Structures and Algorithms* (DSA)⁵ is a widely used library implementing data structures and algorithms under the .NET framework. To answer the research

³<http://cciast.codeplex.com>

⁴<http://pexase.codeplex.com>

⁵<http://dsa.codeplex.com/>

TABLE III
SUBJECTS.

Subjects	# Methods	# Blocks	GLOC
Sets	1	13	39
Searching	2	46	98
Numbers	10	104	306
Strings	7	167	285
Sorting	12	267	472

questions, we conduct experimental study on all the algorithms in DSA. All the algorithms in DSA form five subjects. Subject `Sets` deals with a set-permutation algorithm. Subject `Searching` deals with search algorithms. Subject `Numbers` deals with number-calculation algorithms. Subject `Strings` deals with string-manipulation algorithms. Subject `Sorting` deals with various sorting algorithms (e.g., `QuickSort`, `BubbleSort`, `MergeSort`, `ShellSort` and `RadixSort`). These five subjects range from 13 blocks⁶ to 267 blocks. The details for the five subjects are shown in Table III, in which Column 1 lists the names of subjects; Columns 2-3 show the number of methods and blocks in each subject, respectively; and Column 4 lists gross lines of code (denoted as GLOC) in each subject.

2) *Supporting Tools*: As the evaluation of strong mutant killing requires the concrete execution of various mutants for each subject. Therefore, besides `PexMutator` and `Pex`, in this study, we also use *GenMutants* to generate various mutant versions to evaluate the performance of test inputs in term of strong mutant killing. *GenMutants* is a mutant-generation tool based on the five sufficient mutation operators for .NET applications. Currently, it has been released as open source in our project site⁷.

3) *Experimental Procedure*: Taking into account the efficiency issue, we compare `PexMutator` against `Pex` under the same time constraints. For each subject, we set all the other `Pex` exploration attributes to be large enough, leaving the `Timeout` attribute to be controlled. We set the `Timeout` attribute to be 1s, 5s, 10s, 20s, 40s for each subject. For each combination of subjects and time constraints, we follow the following procedure:

- First, we use `PexMutator` and `Pex` to generate test inputs for the combination of a subject and a time constraint separately. Each technique generates a corresponding set of test inputs.
- Second, we run the two sets of test inputs (generated by `PexMutator` and `Pex`, respectively) against the original subject under test to collect the code coverage achieved by each set of test inputs.
- Third, we run the two sets of test inputs against the meta-program generated by `PexMutator`, and record the number of executed mutants and weakly killed mutants for each set of test inputs.
- Finally, we run the two sets of test inputs against various mutant versions of the original subject (generated by

`GenMutants`) to evaluate the effectiveness in term of strong mutant killing.

B. Results and Analysis

In this section, we present and analyze the experimental results to answer our research questions. The detailed experimental results are shown in Table IV. In the table, Column 1 lists the studied subjects; Column 2 lists the number of weak-mutant-killing constraints generated in the meta-program of each subject (i.e., the number of mutants of each subject); Column 3 shows the time constraints; Columns 4-5 list the code coverage achieved by `Pex` and `PexMutator` on each combination of subjects and time constraints; Columns 6-7 list the number of mutants executed by test inputs generated by `Pex` and `PexMutator`; Columns 8-9 list the number of mutants weakly killed by test inputs generated by `Pex` and `PexMutator`; Columns 10-11 list the number of mutants strongly killed by `Pex` and `PexMutator`; and Columns 12-13 list the number of test inputs generated by `Pex` and `PexMutator`, respectively.

1) *RQ1: Performance of PexMutator*: In this research question, we are mainly concerned with the efficiency and effectiveness of `PexMutator` in term of mutant killing. As shown in Column 11 of Table IV, for all the combinations of subjects and time constraints, the test inputs generated by `PexMutator` successfully kill the majority of all the mutants.

For better analysis, we further present the ratios of mutants that are executed, weakly killed, and strongly killed by `PexMutator`. In Figure 4, the black area (*ERate*) denotes the ratio of executed mutants to all the mutants; the red area (*WRate*) denotes the ratio of weakly killed mutants to all the mutants; and the blue area (*SRate*) denotes the ratio of strongly killed mutants to all the mutants (note that these three areas are overlapped). First, we find that, on all the $5 * 5 = 25$ combinations, `PexMutator` is able to generate test inputs to execute 98.86%, weakly kill 87.05%, and strongly kill 80.00% of all the mutants on average. Second, on the combinations with not too tight time constraints (i.e., greater than 1s), `PexMutator` achieves relatively stable and better results: executing 100.00%, weakly killing 89.83%, and strongly killing 83.40% of all the mutants on average. Note that there are a number of equivalent mutants for every subject and if we exclude these equivalent mutants, `PexMutator` could achieve even higher strong-mutant-killing ratios. For example, subject `Searching` has 9 equivalent mutants, and `PexMutator` is able to kill all its non-equivalent mutants (i.e., with a 100% killing ratio on non-equivalent mutants).

In summary, `PexMutator` is able to strongly kill more than 80% of all the mutants for the studied subjects automatically, and sometimes even achieves the 100% killing ratio on non-equivalent mutants.

2) *RQ2: Comparison of PexMutator against Pex*: First, we compare `PexMutator` against `Pex` in term of strong mutant-killing. As shown in Columns 10-11 of Table IV, the four combinations on which `Pex` outperforms `PexMutator` all have the time constraint 1s. We suspect the reason to be that the time constraint 1s is too tight for `PexMutator` to explore a

⁶A block denotes a sequential set of statements that only have one entrance and one exit.

⁷<http://pexase.codeplex.com/>

TABLE IV
EXPERIMENTAL RESULTS .

Subjects	#Consts	Time (s)	Code Coverage (%)		#Executed Mutants		#Weak-killed Mutants		#Strong-killed Mutants		#Test inputs	
			Pex	PexMutator	Pex	PexMutator	Pex	PexMutator	Pex	PexMutator	Pex	PexMutator
Sets	31	1	100.00	91.00	31	31	25	23	24	20	4	3
		5	100.00	100.00	31	31	25	28	24	25	4	5
		10	100.00	100.00	31	31	25	28	24	25	4	5
		20	100.00	100.00	31	31	25	28	24	25	4	5
		40	100.00	100.00	31	31	25	28	24	25	4	5
Searching	75	1	95.65	95.65	75	75	58	57	58	57	16	17
		5	95.65	95.65	75	75	66	66	66	66	19	23
		10	95.65	95.65	75	75	66	66	66	66	19	24
		20	95.65	95.65	75	75	66	66	66	66	19	24
		40	95.65	95.65	75	75	66	66	66	66	19	24
Numbers	288	1	95.19	94.23	261	261	229	225	206	204	45	56
		5	99.04	99.04	288	288	258	263	238	259	50	88
		10	99.04	99.04	288	288	258	263	238	259	50	89
		20	99.04	99.04	288	288	258	263	238	259	50	92
		40	99.04	99.04	288	288	258	263	238	259	50	93
Strings	343	1	92.22	86.23	310	293	262	244	203	196	51	51
		5	100.00	100.00	343	343	306	311	266	276	86	111
		10	100.00	100.00	343	343	309	311	266	276	96	119
		20	100.00	100.00	343	343	309	311	266	276	97	122
		40	100.00	100.00	343	343	309	311	266	276	100	123
Sorting	440	1	95.13	96.63	420	420	352	355	271	286	58	69
		5	98.88	99.63	440	440	376	389	314	336	72	98
		10	99.63	99.63	440	440	384	391	327	342	79	109
		20	99.63	99.63	440	440	384	391	327	342	80	130
		40	99.63	99.63	440	440	384	391	327	343	81	134

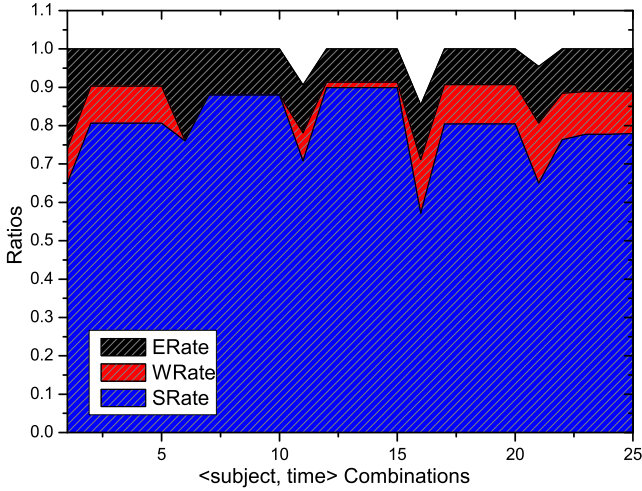


Fig. 4. The performance of PexMutator

complex meta-program. With time constraints greater than 1s, PexMutator is able to show stable performance and strongly kills more mutants than Pex on four of all the five subjects (i.e., except subject *Searching*). On the four subjects (excluding *Searching*), PexMutator is able to outperform Pex by 4.37% in strong mutant-killing ratios. Although the improvement does not seem substantial, we consider it a valuable improvement for the following reason: most mutants are killed as long as they are executed, so that Pex, a state-of-the-art code-coverage-based engine, is able to kill most of them, leaving only a few hard-to-kill mutants. The equal effectiveness of PexMutator and Pex on *Searching* demonstrates this point: all the 9 mutants of *Searching* that cannot be killed by Pex

are equivalent mutants, making it impossible for PexMutator to make any improvement. To further demonstrate this point, we perform a finer-grained analysis on subject *Strings* and find that the mutants killed by Pex are averagely killed 9.20 times by test inputs generated by PexMutator, while the 10 mutants further killed by PexMutator are killed only 1.93 times averagely. Therefore, PexMutator further kills mutants that are hard to kill, and the improvement made by PexMutator is valuable.

Second, the mechanism of PexMutator is to introduce additional mutant-killing constraints to the original program under test to form a meta-program, and the numerous introduced constraints may make the DSE engine to focus on generating test inputs for mutant-intensive parts (which contain more mutants than normal parts) of the program while ignoring the overall code coverage. Therefore, we further compare PexMutator against Pex in term of code coverage. As shown in Columns 4-5 of Table IV, among all the 25 combinations of subjects and time constraints, PexMutator is able to achieve as high code coverage as Pex in 22 combinations. The only 3 combinations on which PexMutator does not perform well in term of code coverage are three subjects with the time constraint 1s (i.e., $\langle \text{Sets}, 1s \rangle$, $\langle \text{Numbers}, 1s \rangle$, and $\langle \text{Strings}, 1s \rangle$). We can conclude that PexMutator may struggle in mutant-intensive areas and compromise overall coverage with very tight time limits (e.g., with the time constraint 1s). Yet, as long as the time constraint is not very tight, PexMutator is able to achieve competitive overall code coverage compared with Pex.

In summary, PexMutator is able to outperform Pex sig-

nificantly in term of strong mutant killing, while holding competitive overall coverage. Also, we find that although the code-coverage-based technique (e.g., Pex) is able to achieve relatively high mutant-killing ratios, it still cannot kill some hard-to-kill mutants.

C. Threats to Validity

Threats to internal validity are concerned with uncontrolled factors that may also be responsible for the results. In our study, the main threat to internal validity is the possible faults in the implementation of our technique and result analysis. To reduce this threat, we implement our approach based on the Microsoft Common Compiler Infrastructure (CCI) and state-of-the-art DSE engine, Pex. Furthermore, we reviewed all the code that we produced to assure its correctness.

Threats to external validity are concerned with whether the findings in our study are generalizable for other situations. In our study, the main threat to external validity is the subjects used in our study. To reduce this threat, we conducted experiments on all the algorithms in the widely used .NET library DSA. However, they still may not be representative for other programs.

Threats to construct validity are concerned with whether the measurement in our study reflects real-world situations. In our study, the main threat to construct validity is the way we measure the effectiveness of generated test inputs. To reduce this threat, we use widely used criteria (i.e., code coverage and number of mutants strongly killed) to evaluate the effectiveness of test inputs.

VI. RELATED WORK

A. Mutation Testing

Mutation testing is a fault-based testing approach, which has been shown to be an effective indicator for the quality of test inputs [2]. In addition, mutation testing has been shown to be superior to commonly code coverage measurements in evaluating the effectiveness of test inputs [12]. The first major work on mutation testing was first proposed by Hamlet [15] and DeMillo et al. [9]. Since the first proposal, mutation testing has been intensively studied. Some researchers investigated techniques to overcome the challenges of using mutation testing to measure the effectiveness of existing test suites (e.g., Acree et al. [1], Budd et al. [4], Wong and Mathur [33], [34], Mresa et al. [21], Hierons and Harman [16], Zeller et al. [14], [28], and Zhang et al. [35]). Some researchers used mutation testing to automatically produce faulty versions to construct subjects for software-testing experimentation (e.g., Briand et al. [3], Mayer et al. [20], Tuya et al. [32], Andrews et al. [2], and Do et al. [11]). There are also researchers using mutation testing as the test criterion and investigating automatic techniques for generating test inputs to satisfy the criterion (e.g., Offutt et al. [10], [22] and Liu et al. [19]).

B. Automatic Test Generation

Automatic test generation can be depicted as automatically generating test inputs that satisfy certain test criteria. Various

automatic techniques have been proposed. Chen et al. [6], Godefroid et al. [13], and Ciupa et al. [7] proposed random or adaptive random techniques to generate test inputs, and their techniques are based on random algorithms. Ramamoorthy et al. [26], Clarke et al. [8], and King et al. [18] proposed test generation techniques using code coverage as the test criterion, and their techniques are based on symbolic execution, which generates test inputs by solving symbolic constraints statically extracted from code. Sen et al. [29], Cadar et al. [5], and Tillmann et al. [30] improved previous techniques based on symbolic execution and developed techniques based on dynamic symbolic execution (DSE), which combines concrete execution and symbolic execution to generate test inputs to achieve standard coverage. Offutt et al. [10], [22] and Liu et al. [19] used mutant killing as the test criterion and their techniques are based on solving statically constructed constraints to kill mutants.

In this paper, we focus on automatically generating high-quality test inputs. As mutant killing has been shown to be an effective indicator for effectiveness of test inputs [12], our PexMutator automatically generates test inputs based on mutant killing. Although there are techniques [10], [19], [22] aiming to facilitate the process of test generation towards mutant killing, these techniques face challenges when being applied on real-world applications. To support effective and efficient mutation killing, we propose our general PexMutator approach: first, PexMutator inserts all the mutant-killing constraints into the original program under test to form a meta-program, instead of the costly traditional way of conjoining one mutant-killing constraint with the original program one at a time; second, PexMutator generates test inputs using a state-of-the-art DSE engine, Pex, which is more effective than previous test-generation techniques used for mutant killing. The efficient meta-program generation and the state-of-the-art Pex engine enable PexMutator to conduct test generation for real-world programs, which most existing test-generation tools for mutation testing cannot handle.

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose a general approach that automatically generates test inputs to kill mutants via dynamic symbolic execution (DSE). Although there are techniques that generate test inputs to kill mutants, our approach is more efficient and effective, making our approach applicable to real-world programs for mutant killing, which most existing techniques cannot handle. We have implemented the proposed approach in a tool called PexMutator as an extension for Pex [30], an automated structural testing tool for .NET developed at Microsoft Research. Our preliminary experimental study shows that PexMutator is able to strongly kill more than 80% of all the mutants for the studied subjects. In addition, PexMutator outperforms Pex in term of strong mutant-killing while holding the same code coverage.

In future work, we plan to address three main issues. First, we plan to integrate PexMutator with Pex, providing PexMutator as a test-generation option within Pex. Second, the

current version of PexMutator introduces as many constraints as mutants of the program under test into the corresponding meta-program, making it expensive for the DSE engine to generate test inputs for the tremendous number of branches. We plan to investigate techniques that generate constraints that enable to weakly kill multiple mutants, thus reducing the number of total generated constraints while keeping the same effectiveness. Finally, we plan to extend our experimental study to more and larger real-world .NET applications to further evaluate the performance of PexMutator.

REFERENCES

- [1] A. Acree, T. Budd, R. DeMillo, R. Lipton, and F. Sayward. Mutation analysis. *Georgia Inst. Technol., Atlanta, Tech. Rep. GIT-ICS-79/08*, 21:220–232, 1979.
- [2] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, pages 402–411. ACM New York, NY, USA, 2005.
- [3] L. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. 2004.
- [4] T. Budd, R. DeMillo, R. Lipton, and F. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 233. ACM, 1980.
- [5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.
- [6] T. Chen, R. Merkel, P. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Proceedings of the International Conference on Quality Software*, pages 79–86, 2004.
- [7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *Proceedings of the International Conference on Software Engineering*, pages 71–80. ACM New York, NY, USA, 2008.
- [8] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, pages 215–222, 1976.
- [9] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
- [10] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.
- [11] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, pages 733–752, 2006.
- [12] P. Frankl, S. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *The Journal of Systems & Software*, 38(3):235–253, 1997.
- [13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.
- [14] B. Grun, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops-Volume 00*, pages 192–199. Citeseer, 2009.
- [15] R. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, pages 279–290, 1977.
- [16] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.
- [17] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, pages 371–379, 1982.
- [18] J. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.
- [19] M. Liu, Y. Gao, J. Shan, J. Liu, L. Zhang, and J. Sun. An approach to test data generation for killing multiple mutants. In *Proceedings of the International Conference on Software Maintenance, 2006. ICSM'06*, pages 113–122, 2006.
- [20] J. Mayer and C. Schneckeburger. An empirical analysis and comparison of random testing techniques. In *Proceedings of the International Symposium on Empirical Software Engineering*, page 114. ACM, 2006.
- [21] E. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.
- [22] A. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software-Practice and Experience*, 29(2):167–194, 1999.
- [23] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):118, 1996.
- [24] A. Offutt, J. Pan, and I. PRC. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, pages 165–192, 1997.
- [25] A. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the International Conference on Software Engineering*, pages 100–107. IEEE Computer Society Press Los Alamitos, CA, USA, 1993.
- [26] C. Ramamoorthy, S. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, pages 293–300, 1976.
- [27] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, 1999.
- [28] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 69–80. ACM, 2009.
- [29] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of ESEC/FSE*, pages 263–272. ACM New York, NY, USA, 2005.
- [30] N. Tillmann and J. de Halleux. Pex—white box test generation for .net. *Tests and Proofs*, pages 134–153, 2008.
- [31] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. of ESEC/FSE*, pages 253–262, 2005.
- [32] J. Tuya, M. Suarez-Cabal, and C. la Riva. Mutating database queries. *Information and Software Technology*, 49(4):398–417, 2007.
- [33] W. Wong. *On mutation and data flow*. PhD thesis, Citeseer, 1993.
- [34] W. Wong and A. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.
- [35] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is Operator-Based Mutant Selection Superior to Random Mutant Selection? In *Proceedings of the International Conference on Software Engineering*, page to appear, 2010.
- [36] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing JUnit test cases in absence of coverage information. In *Proceedings of the International Conference on Software Maintenance*, pages 19–28, 2009.