Guided Test Generation for Coverage Criteria

Rahul Pandita, Tao Xie
Department of Computer Science
North Carolina State University, Raleigh, NC
Email:{rpandit, txie}@ncsu.edu

Nikolai Tillmann, Jonathan de Halleux
Microsoft Research
One Microsoft Way, Redmond, WA
Email: {nikolait, jhalleux}@microsoft.com

***Abstract*—Test coverage criteria including boundary-value and logical coverage such as Modified Condition/Decision Coverage (MC/DC) have been increasingly used in safety-critical or mission-critical domains, complementing those more popularly used structural coverage criteria such as block or branch coverage. However, existing automated test-generation approaches often target at block or branch coverage for test generation and selection, and therefore do not support testing against boundary-value coverage or logical coverage. To address this issue, we propose a general approach that uses instrumentation to guide existing test-generation approaches to generate test inputs that achieve boundary-value and logical coverage for the program under test. Our preliminary evaluation shows that our approach effectively helps an approach based on Dynamic Symbolic Execution (DSE) to improve boundary-value and logical coverage of generated test inputs. The evaluation results show 30.5% maximum (23% average) increase in boundary-value coverage and 26% maximum (21.5% average) increase in logical coverage of the subject programs under test using our approach over without using our approach. In addition, our approach improves the fault-detection capability of generated test inputs by 12.5% maximum (11% average) compared to the test inputs generated without using our approach.**

I. Introduction

Software maintenance is an important and expensive phase in the software development life cycle. Among various activities performed during the maintenance phase, software testing is an important activity to ensure high software quality during maintenance. An existing test suite with high fault-detection capability helps ensure that changes performed during maintenance do not introduce any unwanted side effects. Thus, it is quite important to have test suites with high fault-detection capability during software maintenance. However, manual generation of test suites is often labor intensive. Furthermore, manually written test suites may be insufficient in achieving high structural coverage, thereby incapable of detecting faults that lie in not-covered code, or manually writing test suites to achieve high structural coverage may be very expensive.

Automated test generation has been proposed to reduce human effort in testing. Existing test-generation approaches [9], [19], [18] often use block or branch coverage (among the existing structural coverage criteria) as a primary criterion for test generation. Achieving high block or branch coverage with a passing test suite certainly increases the confidence on the quality of the program under test. However, block or branch coverage itself cannot be argued as a sole criterion for effective testing. Safety-critical or mission-critical domains such as aerospace mandate the satisfaction of other structural coverage criteria [1] that complement the block or branch coverage criteria. *Boundary Value Coverage* (BVC) [15] and *Logical Coverage* (LC) [1] are two such coverage criteria. However, existing test-generation approaches (which use block or branch coverage criteria for test generation and selection) often generate a test suite that achieves low BVC and LC.

In particular, BVC aims at exercising the boundary values of predicates in the branching statements from the program under test, similar to *Boundary Value Analysis* (BVA) [4], [13]. BVA is a popular black-box testing technique that executes the program under test not only within the input range, but also with the boundary conditions (i.e., values from where the range begins and ends). The key insight of using BVA is that many faults reside at the boundary values of inputs in contrast to values that lie within the input range [15]. Since only two values are required to achieve block or branch coverage involving a predicate in a branching statement, existing test-generation approaches fail to achieve complete BVC because achieving high or complete BVC often involves more than two values to be strategically chosen.

In contrast to BVC, LC targets at testing logical expressions that are commonly used in the program under test. LC criteria involve instantiating clauses in a logical expression with concrete truth values. A number of coverage criteria such as Modified Condition/Decision Coverage [1] have been proposed to achieve logical coverage. However, similar to the case of BVC, these criteria require strategic combination of truth values for clauses in the predicate condition, not always resulting in execution of a new block or branch in the program under test. Therefore, existing test-generation approaches fail to strategically choose input values to achieve high or complete LC.

We propose a general solution to address the aforementioned shortcomings of the existing test-generation approaches. In particular, we use code instrumentation to assist an existing test-generation approach that targets at block or branch coverage in achieving BVC and LC. An alternative solution is to modify the internal test-generation mechanisms of existing test-generation approaches. We chose the first solution over the second, since the first solution is generic and can be used in conjunction with any of the existing approaches that target at block or branch coverage. Since these existing approaches by nature are good at achieving high block or branch coverage, our solution first transforms the problem of achieving BVC and LC to the problem of achieving coverage of new blocks or branches that are instrumented into the program, and then applies an existing approach to cover these new blocks or branches. Our solution guides existing approaches to achieve the BVC and LC without any modification of these approaches themselves.

In particular, to transform the problem of achieving BVC and LC to the problem of achieving block or branch coverage, we instrument the program under test with new conditional statements that contain constraints that need to be satisfied to achieve BVC and LC. The purpose of instrumenting the program under test with these statements is to introduce new blocks or branches in the program under test. These statements (when encountered by an existing test-generation approach) result in generation of new concrete input values that achieve high BVC and LC. When we apply an existing test-generation approach on the instrumented program, the output is an extended test suite capable of exercising BVC and LC. By guiding an existing test-generation approach (i.e., by adding instrumented code in the program under test), we achieve high BVC and LC as compared to the coverage achieved by the test-generation approach without guidance.

To implement our approach, we have used Pex [22], an existing state-of-the-art structural testing tool from Microsoft Research that employs Dynamic Symbolic Execution (DSE) [9], [19], as our test-generation tool. DSE has recently emerged as an effective approach to generate a high-covering test suite. A DSE-based approach generates concrete test inputs by symbolically executing the Code Under Test (CUT) in parallel to actual execution with concrete values. This mixed execution collects symbolic constraints on inputs obtained from the predicates in branching statements during execution. The conjunction of these symbolic constraints is called a path condition. In order to maximize structural coverage, a DSE-based approach iteratively explores new paths in the CUT and thereby systematically increases the block or branch coverage of the CUT.

This paper makes the following major contributions:

- A general approach to achieve high BVC and LC via an existing test-generation approach that uses block or branch coverage as the primary criterion for test generation. To the best of our knowledge, our approach is the first one that uses code instrumentation to directly and effectively guide an existing test-generation approach to achieve BVC and LC.
- A tool implementation of our approach using Common Compiler Infrastructure (CCI) [7], a set of components (libraries) that provide functionalities that are commonly used in compilers and other programming tools. Our open source tool is released at http://pexase.codeplex.com/.
- An evaluation of our approach with three benchmarks and two open-source projects. The evaluation results show 30.5% maximum (23% average) increase in boundary-value coverage and 26% maximum (21.5% average) logical coverage of the subject programs under test using our approach over without using our approach. In addition, our approach improves the fault-detection capability of generated test inputs by 12.5% maximum (11% average) compared to the test inputs generated without using our approach.

The rest of the paper is organized as follows. Section II presents background on Pex as well as logical coverage. Section III presents our formal problem definition. Section IV presents illustrative examples of our approach. Section V presents our approach and tool architecture. Section VI presents the evaluation of our approach. Section VII discusses related work. Finally, Section VIII concludes.

II. Background

We next present background on LC followed by background on Pex.

A. Logical Coverage

Logical Coverage (LC) is measured based on *Test Requirements* (TR) (e.g., specific elements of software artifacts) that must be satisfied or covered. In the case of LC, these elements are the clauses and predicates present in the CUT.

First, we define clauses and predicates to describe the LC criteria. "*Let P be a set of predicates and C be a set of clauses in the predicates in P. For each predicate $p \in P$, let C_p be the set of clauses in p, that is $C_p = \{c | c \in p\}$. C is the union of the clauses in each predicate in P, that is $C = \bigcup_{p \in P} C_p$*" [1]. Among various logical criteria, enforcement of Modified Condition/Decision Coverage (MC/DC) by US Federal Aviation Administration (FAA) on safety critical avionics software makes MC/DC a good candidate for test-generation criteria of LC. Based on these definitions, we next provide the definition of Correlated Active Clause Coverage (CACC), also known as the masking MC/DC criterion for a logical expression (the main focus of our work):

- **Correlated Active Clause Coverage (CACC) [1]:** "For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j , $j \neq i$ so that c_i determines p. TR has two requirements for each c_i: c_i evaluates to `true` and c_i evaluates to `false`. The values chosen for the minor clauses c_j must cause p to be `true` for one value of the major clause c_i and `false` for the other, that is, it is required that p(c_i =`true`) $\neq$ p(c_i =`false`)."

A major clause [1] is the focus of testing among the clauses present in the predicate. Since MC/DC is not biased towards any of the clauses, the definition mentions *"each major clause"*. The rest of the clauses present in the predicate other than the major clause are termed as minor clauses [1].

B. Pex

Pex is an automated DSE-based testing tool developed at Microsoft Research. Pex was used internally at Microsoft Research to test core components of the .NET Architecture and found some serious defects [22].

For the implementation of our approach, we use Pex [22] as an example state-of-the-art DSE-based test-generation tool for generating test inputs. DSE is a variation of the conventional static symbolic execution [14]. DSE executes the

CUT starting with arbitrary input values while performing symbolic execution side by side to collect constraints on inputs obtained from the predicates of branching conditions along the execution. The collected constraints are modified in certain ways and then are fed to a constraint solver, which produces the variation of the previous input values to ensure that a different path is exercised in the CUT in future executions. The variation of inputs ensures that every feasible path is executed eventually over the iterations of variations. An important part of the iterative DSE algorithm in Pex is the choice of program inputs produced during iterations. These choices lead to the enumeration of all feasible execution paths in the CUT. The DSE algorithm in Pex is designed to explore all feasible paths in the CUT. It works well to achieve high block or branch coverage; however, the algorithm naturally does not aim to achieve BVC and LC.

III. Problem Definition

Our approach deals with the problem of achieving high coverage on a given CUT (P), in the context of a given coverage criterion C, using a tool that aims at achieving high coverage on the same CUT, however, with respect to a different coverage criterion C'. Our approach aims at transforming the problem of achieving high coverage on P in context of C by transforming P to P' such that the problem of achieving high coverage in P with respect to C is transformed into the problem of achieving high coverage in P' with respect to C'.

We first introduce terms that are used in our formal definition.

- **Coverage function** (Cov) is defined as the function that returns the percentage of coverage achieved by a test suite (T) over a given CUT (P) with respect to a given coverage criteria (C), i.e., $Cov(C, P, T)$.
- **Cost function (CT)** is defined as the function that calculates cost involved in generating a test suite (T) over a given CUT (P) for achieving 100% coverage for a given coverage criterion (C), i.e., $CT(C, P, T)$.

We next formally present the problem definition with respect to the input, output, and properties that need to be satisfied.

For a given CUT P, a tool α that generates test inputs to achieve high coverage on P with respect to a coverage criterion C' and a target coverage criterion C, where $C \neq C'$, the problem is to transform P to P', where $P' == P + \Delta P$ and ΔP is the added instrumentation code to P such that the following requirements are met.

R1: $\forall I : P(I) == P'(I)$, where I is a set of all legal inputs to P, and $P(I)$ and $P'(I)$ are the outputs of P and P' given inputs I, respectively.

The requirement states that the added instrumentation code ΔP should not have any side effect on P. ΔP has side effects if the execution of P' produces a different output from the one produced by the execution of P when executed with the same input.

R2: Given

$\exists T_1 : ((P' \vdash T_1) \& Cov(C', P', T_1) == 100\%)$

where test suite T_1 is generated for P' represented as $(P' \vdash T_1)$, since test-generation tool α uses program P to generate test suite T

then

$(T_1 \neq \Phi) \Rightarrow (Cov(C, P, T_1) = 100\%)$

```
01: public void SampleBVC(int x){
02:  if (x >= 10) { ...... }
03:  else { ...... }
04: }
```

Fig. 1. Example of CUT for BVC

```
TestBVC 0: SampleBVC(1);
TestBVC 1: SampleBVC(1073741824);
```

Fig. 2. Test inputs generated by Pex for `SampleBVC`

The requirement states that if there exists a test suite T_1, generated by α, that achieves 100% coverage on P' with respect to C, then coverage of T_1 on P with respect to C is 100%.

R3: $\forall \Delta P : (\exists T : (P' \vdash T) \& Cov(C', P', T) == 100\%)$ choose ΔP, such that $CT(C', P', T)$ is minimum.

IV. Example

We next explain how our approach guides a DSE-based approach to achieve high BVC and LC using illustrative C# examples shown in Figures 1 and 5. In particular, we explain how our code instrumentation helps in guiding a DSE-based approach such as Pex, to generate target test inputs that can achieve high BVC and LC.

To illustrate our approach, we compare the test inputs generated by Pex before and after the instrumentation of the CUT. Often, achieving BVC and LC requires inputs to be generated for exploring the same path more than once in the CUT. However, since Pex uses block coverage as the primary coverage criterion for test generation and selection, Pex fails to achieve high BVC and LC.

A. Boundary Value Coverage (BVC)

We use the method `SampleBVC` shown in Figure 1 to illustrate guided test generation to achieve BVC. The method accepts an integer variable `x` as a parameter and based upon the satisfaction of the condition `x>=10` (Line 2), the `true` or `false` branch is executed. We observe that the input space for the variable `x` can be partitioned into two ranges, one less than 10 and the other greater than or equal to 10. When we apply Pex to generate test inputs for the `SampleBVC` method, Pex generates test inputs as shown in Figure 2. Pex generates two concrete inputs each being a representative element of each partitioned input range. Although generated inputs achieve complete block coverage, these test inputs clearly do not achieve complete BVC (i.e., requiring that the values of `x` are on the boundary of the partitioned ranges, here being 9 and 10).

To guide a DSE-based approach such as Pex in achieving high BVC, we instrument the CUT to add new conditional statements as shown in Figure 3. These statements introduce

```
01: public void SampleBVC(int x) {
02:  if(x == 10) { }
03:  else if(x > 10) { }
04:  else if(x == (10 - 1)) { }
05:  else if(x < (10 - 1)) { }

06:  if (x >= 10) { ...... }
07:  else {...... }
08: }
```

Fig. 3. Instrumented `SampleBVC` to achieve BVC

```
01: TestBVC0: SampleBVC(1);
02: TestBVC1: SampleBVC(9);
03: TestBVC2: SampleBVC(10);
04: TestBVC3: SampleBVC(1073741824);
```

Fig. 4. Test inputs generated by Pex for BVC-instrumented `SampleBVC`

```
01: public void SampleLC(bool x,bool y,bool z) {
02:   if (x && (y || z) { ...... }
03:    else { ...... }
04: }
```

Fig. 5. Example of CUT for LC

```
01: TestLC0: SampleLC(false, false, false);
02: TestLC1: SampleLC(true, false, false);
03: TestLC2: SampleLC(true, true, false);
```

Fig. 6. Test inputs generated by Pex for `SampleLC`

new branching statements in the CUT guarded by the constraints present in the predicates of the conditional statements. Since the true branches of the introduced branching statements are empty, it can be observed that the addition of these blocks do not have any side effects on the CUT as far as the output of the program is concerned, thus satisfying R1 in the problem definition (Section III).

When we apply Pex to generate test inputs for the instrumented `SampleBVC` method, Pex generates test inputs as shown in Figure 4. The two conditional statements in the instrumented code, `x == 10` (Line 2) and `x > 10` (Line 3) enforce the boundary conditions on the range `x >= 10`. Similarly, the next two conditional statements `x == (10-1)` (Line 4) and `x < (10-1)` (Line 5) enforce the boundary conditions on the range `x <= (10-1)`. These statements guide Pex to generate two additional test input values as specified in the newly added conditional statements. From the test inputs generated by Pex for the instrumented code, it can be observed that the test inputs not only achieve complete block coverage but also achieve complete BVC of the original uninstrumented CUT.

Consider a scenario where a programmer accidentally typed `x > 10` instead of `x >= 10` in the `SampleBVC` method (Line 2) in Figure 1. The test inputs generated by Pex without the guidance of our approach cannot detect the fault. The primary reason is that the inputs generated by Pex do not check the boundary conditions that help detect the fault. However, test inputs generated by Pex with the guidance of our approach can detect the fault.

B. Logical Coverage (LC)

We use the method `SampleLC` shown in Figure 5 to illustrate guided test generation to achieve LC such as MC/DC. The method accepts three `Boolean` values as parameters and based on the satisfaction of the logical condition `(x && (y || z)` (Line 2), the `true` or `false` branch is executed. The test inputs generated by Pex for `SampleLC` are shown in Figure 6. The test inputs generated by Pex for the `SampleLC` method achieve complete block coverage. However, since `z` is never assigned a `true` value and thus MC/DC is not achieved, thereby highlighting the need of guidance to existing test-generation approaches to achieve MC/DC.

```
01: if((true && (y || z)!=(false && (y || z)){
02:   if(x) { }
03:   else if(!x) { }
04: }
05: if(((x && (true || z)!=((x&& (false || z)){
06:   if(y) { }
07:   else if(!y) { }
08: }
09: if(((x && (y || true)!=((x && (y || false)){
10:   if(z) { }
11:   else if(!z) { }
12: }
```

Fig. 7. Instrumentation code to introduce dummy blocks to achieve MC/DC

```
01: TestLC0: SampleLC(false, false, false);
02: TestLC1: SampleLC(false, false, true);
03: TestLC2: SampleLC(true, false, false);
04: TestLC3: SampleLC(true, true, false);'
05: TestLC4: SampleLC(true, false, true);
```

Fig. 8. Test inputs generated by Pex for MC/DC-instrumented `SampleLC`

To achieve MC/DC, we instrument the `SampleLC` method with the code shown in Figure 7. Statement `if((true && (y || z) != (false && (y || z))` (Line 1) in Figure 7 ensures that the variable `x` is the major clause. The subsequent inner conditional statements (Lines 2-3) ensure that when `x` is the major clause, then `x` is instantiated with both `true` and `false` values for different generated test inputs. The further subsequent conditional statements (Lines 5-11) impose similar constraints on `y` and `z`.

Figure 8 shows the test inputs generated by Pex, when applied over the `SampleLC` method instrumented with the code snippet shown in Figure 7. We observe that the number of generated test inputs is more than the required number of test inputs to achieve MC/DC. These additional test inputs are generated since they were generated early in the test generation and covered not-covered branches at that time in the CUT, although they are not required to satisfy the MC/DC criterion. For example, the input shown in Line 1 of Figure 8 does not satisfy any of the constraints in the instrumented code, but its execution still covers new branches, i.e., the `false` branch of every conditional statement (Lines 1, 5, and 9) in Figure 7.

V. Approach

The core idea of our approach is to transform the problem of achieving a target coverage criterion over a CUT into the problem of achieving block coverage, which is the coverage criterion of the chosen test-generation approach, such as Pex. Note that existing test-generation approaches often use block or branch coverage as the coverage criterion for test generation.

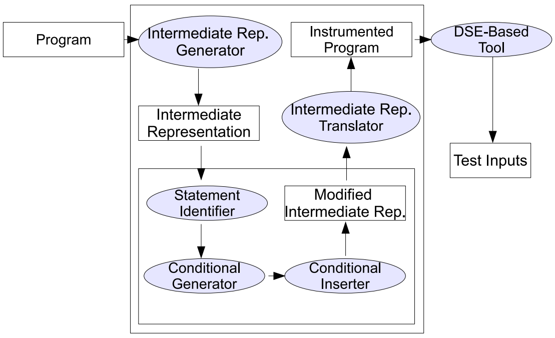

Fig. 9. Overview of our approach

Figure 9 shows a high-level overview of our approach. Our approach accepts a program (source or assembly) under test and constructs an intermediate representation of the program. Our approach then extracts statements that qualify for the target coverage criteria from the generated intermediate representation. For example, our implementation targets at BVC and LC. Therefore, our implementation extracts the predicates of the branching conditions from the generated intermediate representation. Our approach processes these extracted statements based on the requirements of the target coverage criteria to generate conditional statements, placing them strategically into the intermediate representation to produce a modified intermediate representation. Our approach uses the modified intermediate representation to synthesize a new program containing new blocks. Our approach next applies a DSE-based approach on the newly synthesized program.

Our implementation of the approach to achieve BVC and LC transforms the problem of achieving BVC and LC into a problem of achieving block coverage on the instrumented program under test. We next explain each component of the approach using the illustrative examples (for BVC and LC) shown in Figures 2 and 6.

A. Overview of our Approach

- **Intermediate Representation Generator.** The purpose of the intermediate representation generator component is to transform the CUT to an intermediate representation. This representation is required to facilitate effective manipulation of the CUT.
- **Statement Identifier.** We next traverse the generated intermediate representation to identify statements that qualify to be considered for the target coverage criteria. For example, in our implementation for achieving BVC and LC, we identify conditional statements as qualifying statements. The primary reason is that the BVC and LC criteria involve testing the variables present in constituent clauses of the predicates of conditional statements at boundary conditions and testing clauses at controlled assignment of truth values, respectively. In our implementation, this component identifies the predicates in the intermediate representation to be processed to generate conditional statements that are inserted in the intermediate representation.

  For example, when BVC is the target coverage criterion, the predicate `(x >= 10)` in the branching statement `if(x >= 10)` is identified as a qualified statement, while traversing the statements in the `sampleBVC` method of Figure 2. Similarly, when LC is the target coverage criterion, the conditional statement `(x || (y && z)` is identified as a qualified statement for the `sampleLC` method in Figure 5.
- **Conditional Generator.** This component generates conditional statements to be placed in the intermediate representation based on qualified statements for the target coverage criterion and test requirements of the target coverage criterion. In our implementation for BVC and LC, we insert conditional statements based on the clauses extracted from predicates identified by the statement identifier component.
- **Conditional Statement Inserter.** This component places the conditional statements generated by the conditional generator component into the corresponding methods in the intermediate representation. For example, the instrumented CUT generated by adding conditional statements for the methods in Figures 1 and 5 are shown in Figures 3 and 7, respectively.
- **Intermediate representation Translator.** The addition of conditional statements modifies the intermediate representation. This component then transforms the new intermediate representation back into the program (source or assembly) that can be used by the DSE-based approach. This transformation is the inverse of the transformation performed in the intermediate representation generator component.

We next present the conditional generator component in detail in terms of implementation details with BVC and LC as target coverage criteria.

B. Generation of Conditional Statements

The conditional generator component generates conditional statements based on the test requirements of the target coverage criterion and the qualified statements that are identified by the statement identifier component. In our implementation, we focus on BVC and LC as two target coverage criteria. Therefore, the conditional generator component extracts clauses from the predicates identified in the intermediate representation by the statement identifier component. The conditional generator component then evaluates these clauses to be considered for BVC, LC, or both, in order to generate conditional statements.

1) Generation of Conditional Statements for BVC: Table I shows the rules for generating conditional statements for BVC derived from the test requirements of BVC and qualified statements. Column 1 shows the clauses that are extracted from identified conditional statements in the CUT. Columns 2 and 3 represent the corresponding `true` and `false` conditions that need to be satisfied to achieve complete BVC, respectively, based on the test requirements of BVC. The constant `k` is the smallest distance between two consecutive values in data types of `x` and `y`. For example, if `x` and `y` are integers then the value of `k` is 1, which is the smallest distance between two consecutive numbers in the integer data type. The predicate identified for the `sampleBVC` method in Figure 1 is `(x>=10)`. This predicate matches with the last row in Table I. Therefore, the generated conditions are `(x==10)`, `(x>10)`, `(x==10-1)`, `(x<10 - 1)`, `k` being 1 for the integer data type. These introduced blocks neither redefine any existing variables (i.e., changing values associated with local variables) nor result in any new path that changes the execution order of blocks in the original CUT. These properties ensure that the introduced blocks have no side effects as far as the output of the CUT is concerned. Thus these statements satisfy the first requirement (R1) as stated in the problem definition (Section III).

Furthermore, we use the `else if` construct to add constraints. Addition of the `else if` construct reduces the number of additional paths introduced in the CUT, thereby preventing exponential growth of independent paths (feasible or infeasible). Mutually exclusive nature of inputs required to achieve BVC facilitate the application of this strategy. Restriction of the number of introduced additional paths (feasible or infeasible) is important, since increase in the number of paths can lead to path explosion, thereby increasing the cost of test generation using DSE. Furthermore, reducing the additional number of paths introduced in the CUT comes as a requirement (R3) stated in our problem definition (Section III).

TABLE I
BVC CONDITIONAL STATEMENTS FOR CLAUSES.

Condition	True Conditional Statements	False Conditional Statements
$(x == y)$	$(x == y)$	$(x == (y + k))$, $(x == (y - k))$
$(x! = y)$	$(x == (y + k))$, $(x == (y - k))$	$(x == y)$
$(x < y)$	$(x == (y - k))$, $(x < (y - k))$	$(x == y)$, $(x > y)$
$(x <= y)$	$(x == y)$, $(x < y)$	$(x == (y + k))$, $(x > (y + k))$
$(x > y)$	$(x == (y + k))$, $(x > (y + k))$	$(x == y)$, $(x < y)$
$(x >= y)$	$(x == y)$, $(x > y)$	$(x == (y - k))$, $(x < (y - k))$

Algorithm 1 MC/DC Instrumentation Generator

```
Input: Predicate $P$
Output: $List[\,]\ Instrumentation$
 1: $List[\,]\ Instrumentation = \phi$
 2: for each clause $c$ in $P$ do
 3:     $Predicate\ X = P$
 4:     $Predicate\ Y = P$
 5:     for each occurrence $c_x$ of $c$ in $X$ do
 6:         replace $c_x$ with $true$
 7:     end for
 8:     for each occurrence $c_y$ of $c$ in $Y$ do
 9:         replace $c_y$ with $false$
10:     end for
11:     $Condition\ Z = \phi$
12:     $Z.predicate = generateInequality(X, Y)$
13:     $Condition\ ptrue = \phi$
14:     $ptrue.predicate = c$
15:     $Condition\ pfalse = \phi$
16:     $pfalse.predicate = generateNegation(c)$
17:     $ptrue.falsebranch = pfalse$
18:     $Z.truebranch = ptrue$
19:     $Instrumentation.add(Z)$
20: end for
21: return $Instrumentation$
```

2) Generation of Conditional Statements for LC: For the MC/DC coverage, our approach adds conditional statements in the CUT to guide a DSE-based approach to generate a restricted set of inputs from all possible inputs required to achieve MC/DC. In order to achieve MC/DC, we use CACC (described in Section II) as the base criterion for code instrumentation. We use Algorithm 1 to generate the conditional statements for MC/DC. The algorithm accepts a predicate as input and generates a list of conditional statements as output. Lines 5-18 create conditions that force each clause to be a major clause in the predicate. Recall that a major clause should determine the predicate for the given instantiation of other clauses, i.e., if the major clause is `true`, then the predicate evaluates to one of the truth values. Similarly, the predicate evaluates to the other truth value when the major clause is `false` for a given instantiation of minor clauses. The `generateInequality` method in Line 12 accepts two predicates and returns a new predicate with an inequality operator between input predicates. Lines 13-16 ensure that when conditions for a major clause are achieved, then the inputs are generated to cause `true` and `false` values for that clause, respectively. The `generateNegation` method in Line 16 accepts a clause as input and returns a new predicate that is negation of the input clause. The iteration (Line 2) through all the clauses ensures that the process is applied to every clause in the predicate, thereby ensuring MC/DC. The conditional statements for the predicate identified for the `sampleLogical` method in the CUT (shown in Figure 7) are shown in Figure 7.

These instrumented statements guide the DSE-based approach to generate inputs for the variables `x`, `y`, and `z` so that the MC/DC criterion is achieved for the identified predicate. The first conditional statement (Line 1 in Figure 7) ensures that `x` is a major clause for the identified predicate in the CUT, i.e., `(x||y)&&z`. The sub conditions (Lines 2 and 3 in Figure 7) ensure that when instantiations for `y` and `z` are achieved with `x` as a major clause, the DSE-based approach generates different values to cause `true` and `false` instantiations of `x` ensuring that masking MC/DC (CACC) is achieved with respect to `x`. The subsequent conditional statements (Lines 5-12 in Figure 7) ensure that the masking MC/DC is achieved with respect to `y` and `z`. The `if, else` structure cannot be used to restrict the number of additional paths introduced in the method, since there might be (and often is) overlap in the instantiations of the participant clauses.

TABLE II
STATISTICS OF SUBJECTS.

Project	LOC	#Methods
NextDates	157	3
Triangle	72	23
StringDefs	1619	72
TCAS	187	9
CaramelEngine	2023	147

VI. Evaluation

We conducted two evaluations to demonstrate the effectiveness of our approach in achieving BVC and LC (MC/DC). In our evaluations, we used benchmarks that were previously used in evaluating related approaches [3], [21]. We address the following research questions in our evaluations.

- RQ1: What is the percentage increase in the BVC and LC by the test inputs generated by Pex with the assistance of our approach compared to the test inputs generated without the assistance of our approach?
- RQ2: What is the percentage of additional mutants that are weakly killed[1] [17], [8] by the test inputs generated by Pex with the assistance of our approach compared to the test inputs generated without the assistance of our approach? This percentage reflects the increase in the fault-detection capability of the generated test inputs [2].

A. Subject Applications

We used two benchmarks and three open-source applications for evaluating our approach. Two of the benchmarks were written in Java and used in evaluating a related approach [3]. The first benchmark is the triangle classification program (`Triangle` [3]). This program accepts three integer inputs representing lengths of the sides of a triangle and classifies the triangle as invalid, equilateral, isosceles, or scalene, and further into acute, right, or obtuse. The second benchmark `NextDate` [3] accepts three integer inputs as a day, month, and year of a date. `NextDate` validates the inputs and returns the date of the next day. We converted these benchmarks to C# code using Java2CSharpTranslator [12]. We also applied our approach on a C# version of Traffic Collision Avoidance System (TCAS) [21], which was previously used by Hutchins et al. [11] to investigate the effectiveness of dataflow and control-flow based test adequacy criteria for fault detection. TCAS is a basic implementation of an important safety system designed to avoid air collision of aircrafts. Furthermore, we also applied our approach on two open source projects StringDefs [20] and Caramel Engine [6]. StringDefs is an open source C# library for providing abstracted string-processing routines comparable to those of scripting languages such as Perl and Python. Caramel Engine is an open source C# implementation of a logic engine aimed for a wide variety of games. Table II shows the number of the lines of code and the number of methods associated with each subject. The lines of code do not include user-entered blank lines or comments.

B. Evaluation Setup

Although our ultimate goal for RQ1 is to measure the increase in the coverage with respect to BVC and LC, there are currently no specific tools publicly available to measure BVC and LC. The instrumentation code that we add to the CUT essentially transforms the problem of achieving BVC and LC to the problem of achieving block coverage by introducing new blocks in the code. To measure the achieved coverage, we insert `PexGoal.Reached(...)` statements within the introduced blocks. A `PexGoal.Reached(...)` statement is uniquely identified by the identifier name provided as the argument to the method call in the statement. During exploration, whenever such a statement is encountered by Pex, Pex remembers the identifier name of that statement as an achieved goal and outputs the identifier name in coverage reports. We measure the number of additional covered blocks by counting the number of `PexGoal.Reached(...)` statements (that have been introduced by our approach) executed by the generated test inputs with and without the assistance of our approach. The gain in the number of additional `PexGoal.Reached(...)` statements executed by the test inputs generated with the assistance of our approach (over the test inputs generated without the assistance of our approach) reflects the gain in the BVC and LC using our approach.

To investigate RQ1, we measure the number of `PexGoal.Reached(...)` statements inserted into the CUT (denoted by n). We also measure the number of `PexGoal.Reached(...)` statements executed by the test inputs generated by Pex without the assistance of our approach (denoted by n_1). We next measure the number of `PexGoal.Reached(...)` executed by the test inputs generated by Pex with the assistance of our approach (denoted by n_2). We define Improvement Factor (IF_1) as the percentage increase in BVC and LC as $\frac{n_2 - n_1}{n}$. The value of IF_1 indicates the increase in BVC and LC achieved by our approach.

For RQ2, we carry out mutation testing to investigate the fault-detection capability of generated test inputs. We mutate a method under test by modifying one operator at a time in a single line of code in the original method to generate a faulty method (referred to as mutant). In particular, we use relational mutation operators for our evaluation. We replace each relational operator ($>$, $>=$, $<$, $<=$, $==$, and $!=$) with every other relational operator. We also replace boolean operators with each other. For example, we replace $\&\&$ with $\|$ and vice versa. The Statement-Weak (ST-WEAK) mutation testing recommended by Offutt and Stephen [17] is used as the basis for weak mutation testing. In ST-WEAK, we compare the program states after the first execution of the mutated statement. A mutant is considered as killed when there is a difference between the program state reached by executing the mutated statement and the program state reached by executing the original statement. We detect differences in the program states by comparing the output (e.g., the value of the left-hand-side variable of an assignment statement or execution of

[1] Mutants, program state is infected after the execution of the mutated line of code.

a different branch in a conditional statement) of the mutated statement with the output of the original statement. If there is a difference between the two outputs, we consider the mutant as killed. We execute the test inputs generated by Pex with and without the assistance of our approach on the generated mutants. We measure the relative gain in the success of the test inputs generated with the assistance of our approach to kill mutants (causing differences in the program states) over the test inputs generated without the assistance of our approach. This gain reflects increase in the fault-detection capability of our approach [2].

To investigate RQ2, we measure the total number of mutants generated for each subject (denoted by n), the number of mutants killed by the test inputs generated without the assistance of our approach (denoted by u), the number of mutants killed by the test inputs generated with the assistance of our approach (denoted by v), the number of additional mutants killed (that were not killed by the test inputs generated without the assistance of our approach) by the test inputs generated with the assistance of our approach (denoted by k). We also measure the number of equivalent mutants that have the same behavior as the original program (denoted by s) among the mutants that were not killed by either of the two sets of test inputs. We next measure the Improvement Factor (IF_2) of our approach as $\frac{k}{n-s}$. The values for IF_2 indicate additional fault-detection capability of our approach. We also present another improvement factor IF_3 that indicates the effectiveness of the test inputs in killing mutants that were not killed previously. We calculate IF_3 as $\frac{k}{n-u-s}$.

Our current prototype implementation for MC/DC code instrumentation faces issues with conditional statements containing multiple $\|$. We manually instrumented the programs to generate desired instrumentation effect for a few conditional statements of this type. We also did not include 19 methods from the StringDefs application and 8 methods from the CaramelEngine application in our evaluation, since our prototype implementation currently cannot deal with the `foreach` construct used in these methods.

C. Results

RQ1: Coverage. We next describe our empirical results for addressing RQ1. Tables III and IV show our empirical results for BVC and LC, respectively. Column "Project" shows the name of the program. Column "n" shows the total number of mutants created for each program under test. Column "n" in Tables III and IV shows the number of `PexGoal.Reached()` statements (i.e., new blocks) introduced in the original program by the instrumentation process for achieving BVC and LC, respectively. Column "n_1" shows the number of `PexGoal.Reached()` statements covered by test inputs generated without assistance of our approach. Column "n_2" shows the number of `PexGoal.Reached()` statements covered by test inputs generated with the assistance of our approach. Column "a" shows the number ($n_2 - n_1$) of additional `PexGoal.Reached()` statements that are covered by our approach. Column "IF_1" shows the percentage increase in the BVC (Table III) and LC (Table IV) for programs under test.

TABLE III
IMPROVEMENT FACTOR IN ACHIEVING BVC.

Project	n	n_1	n_2	a	IF_1
NextDates	42	30 (71.5%)	39 (93%)	9	21.5%
Triangle	48	33 (69%)	41 (85.5%)	8	16%
StringDefs	212	123 (58%)	149 (70%)	26	12%
TCAS	24	10 (41.5%)	16 (66.5%)	6	25%
CaramelEngine	331	197 (59.5%)	298 (90%)	113	30.5%
Total	657	393 (63.5%)	543 (82.5%)	162	23%

TABLE IV
IMPROVEMENT FACTOR IN ACHIEVING LC (MASKING MC/DC).

Project	n	n_1	n_2	a	IF_1
NextDates	16	13 (81%)	16 (100%)	3	19%
Triangle	24	19 (79%)	24 (100%)	5	21%
StringDefs	166	103 (62%)	146 (88%)	43	26%
TCAS	48	29 (60.5%)	40 (83.5%)	11	23%
CaramelEngine	72	64 (89%)	72 (100%)	8	11%
Total	326	228 (70%)	298 (91.5%)	70	21.5%

Our approach instruments the program under test based on the conditional statements in the program. Since such statements vary from program to program, the increase in the number of blocks (due to our code instrumentation) also varies accordingly. As shown in Column "n", for our subject applications, CaramelEngine had a maximum increase of 331 blocks for BVC instrumentation and StringDefs had a maximum increase of 166 blocks for LC instrumentation.

The results in Tables III and IV show that test inputs generated without the assistance of our approach do not cover all the new blocks added in the programs via code instrumentation. Furthermore, although test inputs generated with the assistance of our approach cover a higher number of blocks (i.e., "n_2" > "n_1" in the Tables III and IV) introduced in a program, they did not cover all newly introduced blocks (i.e., "n_2" < "n" in the Tables III and IV). One of the reasons is that new blocks introduced in a program are infeasible to cover. Figure 10 shows an illustrative example of such a block from the Triangle program.

The triangle program accepts three non-zero and non-negative integer inputs representing lengths of the sides of a triangle and classifies the triangle as invalid, equilateral, isosceles or scalene, and further into acute, right, or obtuse. The program accepts the inputs and stores the largest of the three input in variable a and the remaining input arguments in b and c. Lines 4-8 of Figure 10 check whether a triangle is a right triangle or not. Lines 1-3 are the new lines added for BVC instrumentation. However, as mentioned in the description, there is a conditional check for ensuring that `a`'s value is not smaller than either `b` or `c`. Due to this conditional check, our instrumented code in Line 3 is infeasible. Furthermore, the check for non-zero variable values makes the condition in Line 2 infeasible. This example illustrates one of the reasons why Pex, even with the assistance of our approach, could not cover all newly introduced blocks.

RQ2: Mutation Testing. Table V shows our evaluation

TABLE V
IMPROVEMENT FACTOR IN FAULT-DETECTION CAPABILITY.

Project	#Mutants	Killed w/o	Killed w	Same	# Add. Mutants	IF_2	IF_3
	n	Assist. $(n-u)$	Assist. (v)	Behavior (s)	Killed. (k)		
NextDates	111	90	101	9	11	11%	92%
Triangle	80	51	54	26	3	5.5%	100%
StringDefs	616	465	533	67	68	12.5%	81%
TCAS	42	34	36	6	2	5.5%	100%
CaramelEngine	583	489	544	26	55	10%	81%
Total	1432	1129	1268	134	139	11%	82%

```
if((a*a) == ((b * b) + (c * c)))
else if((a*a) == ((b * b) + (c * c)) + 1)
else if((a*a) == ((b * b) + (c * c)) - 1)
if ((a * a) == ((b * b) + (c * c)))
{
 // right triangle
  triangle = triangle + "right triangle.";
}
```

Fig. 10. An illustrative example that shows infeasible blocks introduced by our instrumentation.

```
01: bool Own_Above_Threat() {
02:    return (Other_Tracked_Alt < Own_Tracked_Alt);
03: }
```

Fig. 11. An illustrative example that shows increase in fault detection capability.

results illustrating the usefulness of achieving BVC and LC. Column "Project" shows the name of the program. Column "n" shows the total number of mutants created for each program under test. Column "u" shows the number of mutants killed by the test inputs generated by Pex without the assistance of our approach. Column "v" shows the number of mutants killed by test inputs generated by Pex with the assistance of our approach. For example, with the assistance of our approach, 11 additional mutants are killed in the program NextDate shown in column "k". Column "s" shows the number of mutants with the same behavior. Column "IF_2" shows the improvement factor IF_2 percentage and column "IF_3" shows improvement factor IF_3 percentage. Our results show that our approach can guide an existing DSE-based approach to generate test inputs that achieve higher fault-detection capability (reflected by"IF_2" and "IF_3") in contrast to test inputs generated by the DSE-based tool without the assistance of our approach.

Figure 11 shows the `Own_Above_Threat` method from the TCAS program to illustrate the increase in the fault-detection capability achieved by our approach. One of the mutants for the code shown in Figure 11 is produced by changing `Other_Tracked_Alt < Own_Tracked_Alt` in Line 2 to `Other_Tracked_Alt != Own_Tracked_Alt`. The mutant is not killed by the test inputs generated by Pex without the assistance of our approach, since Pex does not generate test inputs that exercise boundary value for `Other_Tracked_Alt`, i.e., when `Othern_Tracked_Alt == Own_Tracked_Alt`. However, with the assistance of our approach, Pex generates test inputs to cause `Other_Tracked_Alt == Own_Tracked_Alt`, thereby killing the mutant.

D. Summary

In summary, the results show that our approach can effectively guide Pex to generate test inputs that achieve BVC and LC, represented by increase in the values of Column "n_2" in comparison to values in Column "n_1" for each subject application in Tables III and IV. Evaluation results show that our approach can assist Pex in achieving a maximum increase of 30.5% (23% average) in BVC and 26% maximum increase (21.5% average) in LC of the subject applications. In addition, our approach effectively improves the fault-detection capability of generated test inputs reflected by postive values of Column "k" in Table V, representing additional mutants killed by Pex with assistance from our approach in comparison to the mutants killed by Pex without the assistance from our approach. In our evaluation, there is a maximum gain of 12.5% (11% average) in the fault-detection capability of test inputs generated by Pex with assistance of our approach compared to the generated test inputs without the assistance of our approach.

E. Threats to validity

Threats to external validity primarily include the degree to which the subject applications used in our evaluations are representative of true practice. To minimize the threat, we used benchmarks that were used by previous related approaches [3], [11]. Furthermore, we also applied our approach on two real-world open source applications, StringDefs [20] and CaramelEngine [6]. The threat can be further reduced by evaluating our approach on more subjects. Threats to internal validity include the correctness of our implementation in adding the instrumentation. To reduce the threat, we manually inspected a significant sample of the generated instrumentation code in the evaluations.

VII. RELATED WORK

Automated test generation for achieving high structural coverage is one of the most widely investigated subjects in the area of software testing. Many approaches automate the process of test generation for achieving high structural coverage. There exist approaches [19], [22] that automatically generate test inputs based on DSE. Furthermore, random testing approaches [18] generate test inputs randomly. However, none of these approaches target at achieving BVC and LC criteria, which are the major focus of our approach.

Our approach also builds on previous work [15], [10], [1], that presents a formal definition of BVC and LC criteria used for automatic generation of test inputs. Hoffman et al. [10] proposed an approach for automatic generation of test inputs that satisfy BVC. However, their approach requires testers to provide inputs in the form of specifications of test focus (aspects of programs that are of interest in context of testing), syntax and semantics of templates that specify tests. Furthermore, their approach does not consider structure of the program under test. In contrast, our approach leverages the structure of the program under test to assist a DSE-based approach to generate test inputs that achieve high BVC and LC.

Awedikian et al. [3] proposed automatic MC/DC-based test generation. They use Evolutionary Testing (ET) techniques to generate test inputs to achieve MC/DC. They use a fitness function that is tailored for MC/DC. However, their approach faces the problem of local maxima[1] in achieving MC/DC. In contrast, ET [16] is shown to be highly effective for generating test inputs that achieve high branch coverage. Thus, our approach used in conjunction with a ET-based test-generation tool may be less likely to face issue of local maxima. Furthermore, their approach is restricted to ET techniques for MC/DC-based test generation; in contrast, our approach is general and can be used in conjunction with any existing automatic test-generation approaches that use block or branch coverage as one of the criteria for test generation.

Recent improvement in DSE search strategies [5], [23], guide DSE to effectively achieve high structural coverage of a program under test. However, these techniques do not specifically target to achieve BVC and LC. In contrast, our approach guides an existing DSE-based tool to generate test inputs that achieve high BVC and LC.

VIII. Conclusion

BVC and LC are proposed to complement the block or branch coverage to increase the confidence on the quality of the program under test. Safety-critical or mission-critical domains such as aerospace have mandated the satisfaction of these criteria. However, existing test-generation approaches (which use block or branch coverage criteria for test-generation and selection) do not support effective test generation against BVC and LC. In this paper, we presented a general approach to guide an existing test-generation approach that uses the block or branch coverage criterion for test generation and selection to generate test inputs that achieve high BVC and LC.

Our evaluations on five subject programs show 30.5% maximum (23% average) increase in BVC and 26% maximum (21.5% average) increase in LC of the subject programs under test using our approach over without using our approach. In addition, our approach improves the fault-detection capability of generated test inputs by 12.5% maximum (11% average) compared to the test inputs generated without using our approach.

Currently, our approach does not consider the satisfiability of the constraints used to add empty blocks, thereby increasing the cost of test generation by existing test-generation approaches. In future work, we plan to filter out the constraints that cannot be satisfied to achieve BVC and LC. Furthermore, we plan to extend our approach to achieve complex coverage criteria such as data flow coverage.

Acknowledgments. This work is supported in part by NSF grants CNS-0720641, CCF-0725190, CCF-0845272, CNS-0958235, CCF-0915400, an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

References

[1] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expression. In *Proc. ISSRE*, pages 99–107, 2003.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.

[3] Z. Awedikian, K. Ayari, and G. Antoniol. MC/DC automatic test input data generation. In *Proc. GECCO*, pages 1657–1664, 2009.

[4] B. Beizer and J. Wiley. Black box testing: Techniques for functional testing of software and systems. *IEEE Software*, 13(5):98, 1996.

[5] J. Burnim and K.Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446, 2008.

[6] Caramel Engine. http://caramelengine.codeplex.com/.

[7] Common Compiler Infrastructure, 2009. http://cciast.codeplex.com/.

[8] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. In *IEEE Comput., vol. 11, No. 4*, pages 34–41, 1978.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 75–84, 2005.

[10] D. Hoffman, P. Strooper, and L. White. Boundary values and automated component testing. *Software Testing, Verification and Reliability*, 9(1):3–26, 1999.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. ICSE*, pages 191–200, 1994.

[12] Java 2 CSharp Translator for Eclipse. http://sourceforge.net/projects/j2cstranslator/.

[13] B. Jeng and E. J. Weyuker. A simplified domain-testing strategy. *ACM Trans. Softw. Eng. Methodol.*, 3(3):254–270, 1994.

[14] J. C. King. Symbolic execution and program testing. In *Commun. ACM, 19(7)*, pages 385–394, 1976.

[15] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. In *Proc. ISSRE*, pages 139–150, 2004.

[16] P. McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[17] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Tran. Software Eng., vol. 20, No. 5*, pages 337–344, 1994.

[18] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.

[19] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[20] StringDefs, 2010. http://stringdefs.codeplex.com/.

[21] Traffic Collision Avoidance System. http://sir.unl.edu/portal/bios/tcas.html.

[22] N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[23] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, pages 359–368, 2009.

[1] A local maxima of an ET-based problem is a solution optimal within a neighboring set of solutions, in contrast to a global maximum, which is the optimal solution among all possible solutions.