A SAT-based Method for Basis Path Testing using KodKod

Insang Chung

Department of Computer Engineering, Hansung University, 116 Samseongyoro-16gil, Seongbukgu, Seoul, 136-792, South Korea.

Orcid ID: 0000-0001-8781-0126

Abstract

A large portion of automated testing techniques has been developed for the statement or branch test coverage criteria. Among structural criteria, however, research on basis path testing has received little attention. The basis path testing requires a set of basis paths to be executed during testing. However, traditional testing methods for basis path generation consider only the control flow of a program and furthermore ignore loop. As a consequence, infeasible basis paths can be generated. In this paper, we present a goal-oriented testing method based on SAT (Boolean SATisfiability) for basis path testing. Specifically, the proposed method makes advantage of KodKod which is an efficient constraint solver for first order logic with relations. In the proposed method, KodKod is used to formulate a program and the basis path testing criterion to generate feasible basis paths of optimal lengths. Our method can generate test inputs on which basis paths are executed. We present a case study with some empirical results to show that the proposed method can generate feasible basis paths of optimal lengths effectively.

Keywords: Goal-oriented testing, Basis path testing, Feasible program path, SAT, KodKod

INTRODUCTION

Numerous testing coverage criteria make use of structural information of source codes. They are called white-box testing or structural testing. Each coverage criterion has its own advantages and disadvantages. Among those testing criteria, basis path testing is well known because it introduces a new notion of basis paths [1]. Any program path can be represented by a linear combination of the set of basis paths, while no path in the basis set can be defined as a linear combination of the others in the set. Program paths making up the basis set are meant to be linearly independent. This is analogous to a set of basis vectors which can represent every vector in the vector space as a linear combination of this set. Basis path testing requires basis paths to be tested. This test requirement is based on the assumption that if basis paths could be proved to be fault-free, then any path expressed in terms of these basis paths is correct.

Another interesting property of basis path testing is the number of linearly independent paths to be tested. The number of basis paths is equal to the cyclomatic complexity of a program under consideration [1]. The cyclomatic complexity of a program can be calculated from the number of nodes and edges when the program is represented as a control flow graph. It provides an upper limit of program paths which we should test. Basis path testing subsumes the statement and branch coverage criteria by testing relatively a few number of program paths [2].

However, basis path testing has a problem that needs to be resolved in order to be applicable in practice. Some of program paths making up the basis set could be infeasible, meaning that they cannot be ever executed [3]. This is due to the way in which existing algorithms produce basis paths [4, 5]. Those algorithms consider decisions to be independent. In practice, it is not true. A decision can have a direct impact on another decision. For example, a decision taken at a node of the control flow graph can prohibit another decision from having certain outcome. It would make a path traversing those two decisions infeasible [6].

Furthermore, algorithms for basis path generation produce a basis path as a form of multiple program points, i.e., branches to be traversed rather than a complete program path [4, 5]. As a result, we should find a feasible program path traversing the selected branches while satisfying constraints among those branches. However, the number of control flow paths that go through the selected program points increases exponentially with respect to the number of program decisions or even can be infinite when loops are unbounded. It would be very difficult to determine the infeasibility of a given basis path, especially when only a few program paths covering the selected program points are feasible. Even when a feasible program path satisfying the basis path testing criterion exists, it is always possible to select a path which happens to be infeasible. In that case, we would have a false positive result indicating that the given basis path is infeasible. Unless all possible paths covering a given basis path are explored, we could not say anything about the infeasibility of the basis path. This characteristic of basis path testing requires the goal-oriented approach which does not demand to specify specific complete paths.
In this paper, we present a goal-oriented test approach based on SAT(Boolean SATisfiability) to tackling those problems mentioned above. Specifically, we transform a program into a constraint system by using KodKod [7]. KodKod which is an efficient constraint solver for first order logic with relations and provides a clean Java programming interface that has been designed to efficiently construct, manipulate, and solve constraints.

Furthermore, the proposed method can generate a feasible program path going through multiple program points. This is essential for basis path testing because a basis path is usually represented as a sequence of multiple program points rather a specific single program point. However, most existing goal-oriented approaches focus on satisfaction of the statement or branch coverage which requires a single program point to be specified for test requirements. Previous works on program testing show that KodKod allows us to effectively handle any kind of constraints required to specify various test coverage criteria [8, 9].

Even though SAT is a decision problem of determining if there is a value assignment that causes a given Boolean formula to evaluate to true, our work is also about an optimization problem where the objective function is to minimize the lengths of program paths. We try to find a solution by solving a series of decision problems stated as follows: “is there a feasible program path going through a basis path for a given scope?” If such a path exists, the path will have optimal length. We iteratively determine the existence of the path with varying scope until SAT solving computation cannot be completed within a reasonable time.

The rest of the paper is organized as follows. In Section 2, we present the notion of Program Assertion Graph (PAG) where serves as a basis to transform a program into a constraint system for SAT solving. Section 3 presents how to formulate a program and test coverage criteria in KodKod to treat test generation problem into a SAT problem. Section 4 presents a case study with some experimental results. Section 5 presents related work. Section 6 concludes the paper.

**PROGRM ASSERTION GRAPH**

Generation of the constraint system begins with construction of a Program Assertion Graph, shortly PAG. A PAG of program P is a directed graph (N, E, s, e, Σ) where 1) N is a set of nodes (N=∪Nk=1n−1Ni), 2) E is a binary relation on N (a subset of N×N), referred to as a set of edges, 3) s and e are, respectively, unique start and unique end nodes and s=N0, e=Ne−1, and 4) Σ is a set of atoms, referred to as location instances (Σ=∪|Ni|n−1Ni). Each node corresponds to a program point. Whenever execution reaches a node Ni, a new program location instance nk is created and it is included in the corresponding node, i.e., nk ∈ Ni. The cardinality of each node Ni represents the number of visits of program point Ni during execution. The following properties should hold between nodes and location instances:

- Σ is a totally ordered set such that the first element with respect to that order is included in s and the last element is included in e.
- Location instances cover Σ, i.e., Σ=∪n−1i=0Ni
- Nodes Ni, Nj for all i, j∈0..n−1 are pairwise disjoint, i.e., ∀Ni, Nj∈N, Ni≠Nj ⇒ Ni ∩ Nj = ∅.

We will call the cardinality of Σ, denoted by |Σ| a path scope. The path scope plays an important role in the size of state space in which we seek a solution. As the path scope increases, the search space becomes larger.

An edge indicates a possible execution flow between the corresponding program points. Each edge is associated with a first order predicate formula which corresponds to a statement or a condition executed when the edge is traversed. For each edge (Ni, Nj), the associated predicate says what should happen at Nj if execution starts at Ni. We will call an edge (Ni, Nj) a branch if Nj has multiple outgoing edges. Each branch in PAG is labeled by a predicate, referred to as a branch predicate, describing the conditions under which the branch will be traversed. For each branch b, paired(b) denotes the paired branch, i.e., the alternative branch of b. For each conditional statement con, b is executed when con evaluates to true (or false) if and only if paired(b) is executed when con evaluates to false (or true).

In particular, a PAG has a special edge (loop) which connects the end state to itself. The loop is introduced for the circumstance where a solution of length shorter than the specified path scope is found. Figure 1(b) shows a PAG of the example program in Figure 1(a) where P(s1) is a predicate formula corresponding to statement s1. Two adjacent edges can be combined into one if the statements associated with them are independent. For instance, the edges labeled with the assignments “i++; x++” are combined into one edge because they do not affect each other. We will show how to transform a statement into a predicate in Section 3.

A path in a PAG is a sequence <Nk1, Nk2,...,Nkr> of nodes, such that Nk1=s, Nkr=e, and for all 1≤i≤r−1, (Nki, Nki+1)∈E. A path <Nk1, Nk2,...,Nkr> is feasible if there exists a sequence <nq1, nq2,...,nqr> of program location instances such that for all 1≤i≤r, nqi∈Nki and for all 1≤i≤r−1, nqi immediately precedes nqi+1 under the total ordering.

We base generation of a constraint system of a procedure on the PAG. We now give some basic KodKod formulations on which the constraint system is built. Firstly, we define nodes as relations as shown in Figure 2. The unary relation LocationSet represents the entire set of nodes of the PAG. The array N is the Relation type and each element of N, i.e., N[i] is used to denote a node. Unary relations s and e
represent the entry and exit nodes, respectively. We also need
to define the total ordering \( \text{lord} \) among program location
instances in \( \text{LocationSet} \). The \texttt{totalorder} method
returns a formula, namely \( \text{ordformula} \), that imposes a total
ordering over the location instances in \( \text{LocationSet} \) and the
first and last elements are given by \( s \) and \( e \).

Figure 1: Example program and its corresponding PGA

The location instances are named “1” suffixed with an integer
\( i \) for \( 0 \leq i \leq \text{pathscope}-1 \) as shown in the \texttt{bounds}
method shown in Figure 3. The limits on the values each relation
can contain are specified separately through the \( \text{Bounds} \) object. A
problem’s universe is given as a \( \text{Universe} \) object constructed
from a user-provided collection of objects. Each \( \text{Universe} \)
object provides a \( \text{TupleFactory} \) for creating constants,
represented by \( \text{TupleSet} \) objects, from atoms drawn from that
\( \text{Universe} \). The \texttt{bound} method accepts an integer \texttt{pathscope}
as one of its input. It is an upper bound on the length of paths
to be explored. If we could not find a feasible path satisfying a
specified test coverage criterion, we try to seek a solution in
larger search space by increasing the value of \texttt{pathscope}.

The \texttt{buildrules} method in Figure 2 returns a formula \texttt{rules}
that enforce the property of partitioning the set of location
instances into \( N[i] \)s for \( 0 \leq i \leq \text{pathscope}-1 \). These formulae
form essential parts of the constraint system for SAT solving.

Figure 2. Formulation of nodes and location instances in KodKod

Figure 3. The \texttt{bounds} method that constructs location instances
TRANSFORMATION OF A PROGRAM TO A CONSTRAINT SYSTEM FOR SAT SOLVING

In this paper, we restrict our analysis to integer variables, arrays, and a structured subset of C programming language constructs including sequencing, if-then-else, and while or for loop statements. The treatment of pointers, especially heap-directed pointers which reference dynamically allocated objects usually involving complex structures is not presented. Based on our earlier work [10], we anticipate that the proposed method can be extended to pointers without much trouble. Let us start with variables.

Variables

We need to treat each program variable as a logical variable [10]. It is required that each variable has a unique static definition point. The property is ensured by doing:

- every assignment to a variable \( v \) generates a new variable \( v_i \) where \( i \) is a unique number
- just after the assignment to \( v \), \( v_i \) becomes the current name (the last version or the current instance) of \( v \), and
- every subsequent use of \( v \) is replaced by its current name \( v_i \).

For example, the sequence of code \(<x=10; x=x+3>\) is converted into \(<x_1=10; x_2=x_1+3>\). Note that the program variable \( x \) is replaced by two newly created variables \( x_1 \) and \( x_2 \). Introduction of those variables allows us to treat assignments as equalities. As a result, the first assignment can be treated as the equality to assert that \( x_1 = 10 \) and the second assignment as the equality to assert that the value of \( x_2 \) is equal to the result of adding 3 to the value of \( x_1 \).

Based on the observation, we treat each variable as a function mapping each program location to a value. Suppose that we encounter an assignment at program point \( N_1 \) that defines a variable \( v \) during execution. Then, a new location instance, namely \( l_{vk} \), is created and is associated with a value, namely \( w \), that the variable \( v \) will be assigned at \( N_1 \). The assignment causes \((l_{vk}, w) \in v\). The newly created location instance \( l_{vk} \) plays a role as an index of a new definition for the variable \( v \).

We can refer to the new definition in KodKod as \( \text{join}(v) \). For program variables \( x, i, n \) in the program in Figure 1(a), the code fragment in Figure 4 declares three binary relations and asserts that these relations are functions from LocationSet to integers.

```java
private final Relation x = Relation.binary("x");
private final Relation i = Relation.binary("i");
final Formula xVar = x.function(LocationSet, Expression.INTS);
final Formula iVar = i.function(LocationSet, Expression.INTS);
final Formula nVar = n.function(LocationSet, Expression.INTS);
```

Figure 4. Relations for variables

These sets are defined in the \textit{bounds} method in the same way that LocationSet is defined. For instance, suppose that we have an array, named \textit{arr}, of integer type where the size of \textit{arr} is given \textit{id}x and \textit{u} is the upper bound on the values of array elements. The code fragment in Figure 5 shows transformation of the array into KodKod.

Array

We formulate arrays as ternary relations involving three unary relations: LocationSet, Index, and Values where LocationSet is the set of location instances, Index is the set of values that the index of an array can take, and Values is the set of values of the array type.

Given a program location in terms of a specific location instance, this array formulation gets us a relation from indexes to values. That is, given a program location and an index, we can get some value

Assignment and Decision

An assignment is translated into a function with two parameters of Expression type that returns a formula corresponding to the statement. These parameters will take location instances \( N_1, N_2 \) such that \( N_1 \) immediately precedes \( N_2 \). For instance, the assignment “\( i++ \)” in Figure 1(a) generates the predicate shown in Figure 6.

The formula \( f_1 \) asserts that the value of variable \( x \) at location \( N_2 \) is equal to the value of incrementing the value of \( x \) at location \( N_1 \). The formulae \( f_2 \) and \( f_3 \) state that the assignment should not change the other variables, say \( i \) and \( n \). The conjunction of these formulae will be returned to become a part of the whole constraint system.

```java
private final Relation arr = Relation.ternary("array");
final Formula arrayrule = l.join(i.join(arr)).lone().forAll(l.oneOf(Index) and(l.oneOf(LocationSet));
public final Bounds bounds(int pathscope, int idx, int u, ...) {
    final TupleSet ibound = frange((tupl(\{\text{Integer}.valueOf(0), \text{Tuple}.Integer.valueOf(idx-1)})
    final TupleSet Values = frange((tupl(\text{Integer}.valueOf(0)), \text{Tuple}.Integer.valueOf(u-1)));
    b.boundExactly(idx, ibound);
    b.bound(arr, b.upperBound(LocationSet).product(b.upperBound(Index));
    b.product(Values));
}
```

Figure 5. Array formulation in KodKod
A decision generates two formulae. One formula is for the case where the outcome of the decision is \texttt{true} and the other is for the case where the outcome of the decision is \texttt{false}. For instance, the formulae corresponding to the decision \texttt{"x==3"} in Figure 1(a) are generated through the two functions \texttt{branch3} and \texttt{branch4} shown in Figure 7.

Both of the functions \texttt{branch3} and \texttt{branch4} use the function \texttt{nothing} that returns a formula asserting that all program variables do not change due to execution of the decision. In fact, the function \texttt{nothing} corresponds to an operation \texttt{NOP} that does nothing, so has no effect. Figure 8 shows the \texttt{nothing} predicate.

The next thing to consider is the alternative control flow associated with the conditional statement. If the conditional statement is of the form \texttt{"if con s1 else s2"}, then Figure 11 shows the corresponding part of the PAG where \texttt{P} is the formula for the case where the outcome of \texttt{con} is \texttt{true}, \texttt{Q} for the case where the outcome of \texttt{con} is \texttt{false}, and \texttt{P1}, \texttt{P2} are the formulae for the \texttt{then} and the \texttt{else} parts of the conditional.

The conditional statement is translated into KodKod as shown in Figure 12. The formulae denote a relation between the decision and statements \texttt{s1} and \texttt{s2}.

Translation of the loop control flow is not different from the case of the alternative control flow. Figure 13 shows the PGA corresponding to the loop statement \texttt{"while (con) s1"}. The edge \texttt{(13, 11)} causes a cycle that can possibly visit the control point \texttt{N(k0)} many times as long as the condition \texttt{con} is true.
B. Basis path testing criterion

Basis path testing is first introduced by McCabe [1]. Basis path testing subsumes the statement and branch coverage criterion. Basis path testing requires execution of independent paths of a procedure. Interestingly, the number of independent paths is equal to the cyclomatic complexity computed with the formula $|E| - |N| + 2$ where $E$ is the set of edges and $N$ is the set of nodes in the control flow graph. Unlike the statement and branch coverage criteria demanding a specification of a single program point, basis path testing requires multiple program points, especially branches to be executed while meeting certain constraints.

For instance, there are three basis paths in the example program of Figure 1(a) because its cyclomatic complexity is 3, i.e., $9-8+2=3$. Let us generate those three basis paths using the baseline method [5]. Firstly, we need to select a baseline path starting at the entry node. If we take the path $<s, N1, N2, N3, N1, N4, N5, N6, e>$ as the baseline, then we find the following basis paths:

$$P1=<s, N1, N2, N3, N1, N4, N5, N6, e> \text{ baseline}$$
$$P2=<s, N1, N2, N3, N1, N4, N6, e>$$
$$P3=<s, N1, N4, N5, N6, e>$$

There are two decision nodes $N1, N4$ in the baseline because if a decision is passed again for the loop, it is not thought of as a new decision. Note that the second $N1$ is not considered and its value need not change to generate a new basis path because it is iterated again for the loop.

Let us generate a test input on which the baseline $P1$ is executed. We can consider two approaches for the generation of test input. One is the path-oriented approach which requires specifying the full path. However, it can cause a severe problem. For instance, let us insert the assignment "$x=1$" before the loop in the program in Figure 1(a). The PGA does not change because the assignment "$i=1; x=1" is not considered and $x$ can be associated with one edge ($s, N1$). The program after the insertion of the assignment makes the baseline $P1$ infeasible because the baseline method does not take the loop into account when generating basis paths. In order to address the problem, we interpret each generated basis path as a test requirement rather than a complete program path to be traversed. For instance, we consider the basis path $P1$ as a test requirement stating that:

- the branch ($N1, N2$) should be executed prior to the branch ($N4, N5$),
- the branch ($N1, N2$) should be executed prior to its paired branch ($N1, N4$), and
- the branch ($N4, N5$) should be executed prior to its paired branch ($N4, N6$)
We have to remind that multiple occurrences of a decision due to a loop should not be considered a separate decision. Consequently, for each branch \( b \) in the test requirements, we have an implicit requirement that its alternative branch, \( \text{paired}(b) \) should not be executed prior to \( b \). From now on, we will denote the test requirement as \(<(N1, N2), (N4, N5)>\) where \( \text{paired}((N1, N2))=(N1, N4) \) and \( \text{paired}((N4, N5))=(N4, N6) \). Then, we just need to satisfy those requirements irrespective of the number of iterations.

Multiple iterations of the loop can reveal a feasible path for the baseline. Consequently, for each branch to a loop should not be considered a separate decision. Rather than real paths to be executed as mentioned above, for a feasible path satisfying those requirements is found. The other approach is to specify only branches extracted from a program. This is based on the interpretation that a basis path set states test requirements going through the specified branches. This is a time-consuming task. Definitely, the process is a time-consuming task.

The other approach is to specify only branches extracted from a selected basis path. Our goal is to generate a feasible path going through the specified branches. This is based on the interpretation that a basis path set states test requirements rather than real paths to be executed as mentioned above. For instance, we have the following test requirements corresponding to each basis path set:

- TR1 = \(<(N1, N2), (N4, N5)>\) for P1
- TR2 = \(<(N1, N2), (N4, N6)>\) for P2
- TR3 = \(<(N1, N4), (N4, N5)>\) for P3

These considerations result in the function \( \text{edge} \) in Figure 17 which returns the formula in KodKod for edge \((N[i], N[j])\) having \((N[i], N[k])\) as its paired edge. The statement labeled with (A) gets you the set of the program points executed prior to the program point \( N[i] \). The statement labeled with (B) brings about the formula asserting that for the edge \((N[i], N[j])\), its paired branch \((N[i], N[k])\) should not be executed prior to the target edge \((N[i], N[j])\).

```java
public Formula edge(int i, int j, int k) {
    Variable I1 = Variable.unary("location 1");
    Expression I2 = I1.join(lord);
    Variable I3 = Variable.unary("location 3");
    Formula f0 = I1.in(N[i]).and(l2(in(N[j])));
    Expression prevs = I1.join(lord).transpose().closure(); -A
    Formula f1 = I3.join(lord).intersect(N[k]).not().forAll(I3.isOneOf(N[i]).int
    Formula f = (f0.and(f1)).forSome(I1.isOneOf(LocSet.difference(e)));
    return f;
}
```

**Figure 17. Translation of basis path coverage criterion in KodKod**

Solving the resultant constraint system generates the feasible path that goes through the branches in TR1: \(<s,N1,N2,N3,N1,N2,N3,N1,N4,N5,N6,e>\). The solver also finds a test input \( x=3, i=19, n=3 \) on which the path is executed. Note that only the variable \( n \) determines a path to be executed in the changed procedure. The path on the test input traverses the loop twice and has an optimal length. This is a significant benefit using a SAT solver for our purpose. Table 1 shows the result of applying our method to the program in Figure 1 after insertion of the assignment “\( x=1 \)”. Note that we also find the path of optimal length.

<table>
<thead>
<tr>
<th>Test Requirement</th>
<th>Path</th>
<th>Test Input</th>
<th>Path Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR1</td>
<td>(&lt;s,N1,N2,N3,N1,N2,N3,N1,N4,N5,N6,e&gt;)</td>
<td>(x=3,i=19,n=3)</td>
<td>11</td>
</tr>
<tr>
<td>TR2</td>
<td>(&lt;s,N1,N2,N3,N1,N4,N6,e&gt;)</td>
<td>(x=3,i=19,n=2)</td>
<td>7</td>
</tr>
<tr>
<td>TR3</td>
<td>infeasible</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Handling infeasible paths

Each path can be represented as an edge vector where the elements of the vector are the number of times each edge appears along the path. As an example, for the paths P1, P2, P3, making up the basis path set in the example program in Figure 1(a), the corresponding edge vectors are shown in Table 2. Figure 18 shows the control flow graph of program 1(a) where the number at each edge specifies the corresponding position of the edge vector.

<table>
<thead>
<tr>
<th>Path</th>
<th>Test Input</th>
<th>Path Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>P2</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>P3</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Then, we have the following important characteristics:

- Any path of a program can be represented by a linear combination of its basis path set
- Each path in the basis path set cannot be represented by a linear combination of the other paths in the basis set. That is, the paths of the basis path are linearly independent with each other.

Let us take an example. Consider the path \( P4=<s,N1,N2,N3,N1,N2,N3,N1,N4,N6,e>\). The corresponding edge vector is \(<1,2,2,1,0,1,0,1,1>\). We can represent the edge vector as a linear combination of the edge vectors corresponding to the basis paths as \( P4=V1+V2-V3 \).

However, for instance, the path P1 cannot be represented by a linear combination of the paths P2 and P3. The second characteristic states that any basis path set is the maximal set of linearly independent paths. However, these notions of basis path testing do not take the feasibility of the paths into account.
For instance, consider Figure 19 which shows an example program and the control flow graph with the edges annotated with the positions of the edge vector [4]. Let us generate the basis path set by the baseline method. Then we have the basis path set and its corresponding edge vectors shown in Table 3. Even though there exists three basis paths, two of the basis paths, \( Q_2 \) and \( Q_3 \), are infeasible because there is no value of variable \( x \) that will execute the body in the first conditional, but not execute the body in the second and vice versa. However, even in this case, the program can possibly have another feasible path. When some of the basis paths are found to be infeasible, the set of the remaining feasible basis paths does not make up the maximal set of linearly independent paths. We need to check if another feasible basis path exists.

### Table 2: Edge vectors

<table>
<thead>
<tr>
<th>Path ID</th>
<th>Path</th>
<th>Edge Vector ID</th>
<th>Edge Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>(&lt;s,N1,N2,N3,N1,N4, N5,N6,e&gt;)</td>
<td>V1</td>
<td>(&lt;1,1,1,1,1,0,1,1,1&gt;)</td>
</tr>
<tr>
<td>P2</td>
<td>(&lt;s,N1,N2,N3,N1,N4, N6,e&gt;)</td>
<td>V2</td>
<td>(&lt;1,1,1,1,0,1,0,1&gt;)</td>
</tr>
<tr>
<td>P3</td>
<td>(&lt;s,N1,N4,N5,N6,e&gt;)</td>
<td>V3</td>
<td>(&lt;1,0,0,0,1,1,0,1,1&gt;)</td>
</tr>
</tbody>
</table>

### Table 3. Basis path and the corresponding edge vectors

<table>
<thead>
<tr>
<th>Path ID</th>
<th>Path</th>
<th>Edge Vector ID</th>
<th>Edge Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>(&lt;s, N1, N2, N3, e&gt;)</td>
<td>E1</td>
<td>(&lt;1,0,1,1,0,1&gt;)</td>
</tr>
<tr>
<td>Q2</td>
<td>(&lt;s, N1, N2, e&gt;)</td>
<td>E2</td>
<td>(&lt;1,0,1,0,1,0&gt;)</td>
</tr>
<tr>
<td>Q3</td>
<td>(&lt;s, N2, N3, e&gt;)</td>
<td>E3</td>
<td>(&lt;0,1,0,1,0,1&gt;)</td>
</tr>
</tbody>
</table>

Figure 20 shows the proposed procedure that deals with the case where an infeasible basis path is found. Let us take an example to illustrate the procedure. In Table 3, both of the paths \( Q_2 \) and \( Q_3 \) are infeasible. Thus, we have \( BP=\{Q_1\} \) and \( BC=\{<(s,N2),(N2, e)>\} \).

The set \( BC \) contains the branch combination \( bc \) where all the decisions are \( false \). The branch combination \( bc \) is an unique combination of outcomes of the decisions not considered in the basis path set. It is also linearly independent with \( BP \) because it cannot be represented by a linear combination of \( Q_1 \). Then, we try to generate a feasible basis path satisfying the predicate corresponding to \( bc \). In the example, we can yield the feasible path \(<s, N2, e>\), which will be added to \( BP \). The procedure stops because \( BC \) becomes empty.

**Figure 19: An example program and its control flow graph**

**Figure 20. A procedure of handling an infeasible basis Path**

Let us apply the procedure to TR3 in Table 1. The procedure starts with \( BC=\{<(N1,N4), (N4, N6)>\} \), which is linearly independent with other paths in the basis path set. The SAT-solver gets you a feasible path \(<s, N1, N4, N6, e>\). As a result, we have three feasible basis paths, which is equal to the complexity number of the program. In this case, the procedure stops because the number of the basis paths reaches the cyclomatic number of the program.
CASE STUDY

This section presents a case study where we have used a program in C language adapted from [12]. The program and its PGA are in Figure 21. The program is supposed to determine minimum and maximum values for selected elements of array Arr. It is simple, but has moderate complexity with an array, a loop, and two conditional statements. The baseline method generates 4 basis paths without considering the feasibility of the paths:

P1=<s, N1, N2, N3, N4, N5, N6, N1, e> baseline
P2=<s, N1, N2, N3, N4, N6, N1, e>
P3=<s, N1, N2, N4, N5, N6, N1, e>
P4=<s, N1, N6, e>

From the generated basis set of paths, we have the following test requirements:

TR1=<s, N1, N2, N3, N4, N5>
TR2=<s, N1, N2, N3, N4, N6>
TR3=<s, N1, N2, N4, N5, N6>
TR4=<s, N1, e>

Table 4 shows some results of solving those test requirements with the SAT solver, SAT4J when the size of the array Arr is set to five.

![Figure 21. An example program and the corresponding program assertion graph](image)

**Table 4:** Application results of the proposed method to the example program

<table>
<thead>
<tr>
<th>Test Requirement</th>
<th>Path ID</th>
<th>Path</th>
<th>Path Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR1</td>
<td>P1</td>
<td>infeasible</td>
<td>NA</td>
</tr>
<tr>
<td>TR2</td>
<td>P2</td>
<td>&lt;s, N1, N2, N3, N4, N6, N1, e&gt;</td>
<td>7</td>
</tr>
<tr>
<td>TR3</td>
<td>P3</td>
<td>&lt;s, N1, N2, N4, N5, N6, N1, e&gt;</td>
<td>7</td>
</tr>
<tr>
<td>TR4</td>
<td>P4</td>
<td>&lt;s, N1, N6, e&gt;</td>
<td>3</td>
</tr>
</tbody>
</table>

Three of the four basis paths turn out to be feasible. We could not find a feasible path satisfying the test requirement TR1. The reason why the baseline method can possibly take an infeasible method is because all decisions are implicitly assumed to be independent [6]. However, this is not a realistic assumption. The two conditionals “a[i]<max” and “a[i]<min” of the example program affects each other. If one is true, then the other could not become true.

As a result, we are supposed to apply the procedure in Figure 20 to find another feasible basis path. Table 5 shows test inputs that our method generates for each feasible basis paths P2, P3, P4. Even though Table 5 does not include the result of the application of the procedure to the test requirement TR1 for the sake of simplicity, we could find a feasible path <s, N1, N2, N4, N6, N1, e> which is supposed to be executed when the values of the variables low, step are provided such that Arr[low] is equal to Arr[low+step]. The basis path would be added to the feasible basis path set. All of these test inputs force execution of paths of optimal lengths, while satisfying the corresponding test requirements.

We have conducted an experimental study to investigate the effectiveness of our method in terms of solving time. Definitely, the solving time depends on the size of the state space to be explored. In the example program, the array size and integer values of variables determine the size of the state space. The experiment has varied the array size from 5 to 20 by 5 and used integer domains dom1=[0..24], dom2=[0..49], and dom3=[0..99]. As a result, we have the largest search space when the array size is 20 and the integer domain is [0..99]. Its size is 100^20, which is huge.

The experimental results are plotted in Figure 22. The x-axis gives the array size and the y-axis gives the solving time elapsed by the proposed method to find a solution. For each array size and integer domain, we have repeated the experiment five times and calculated the average time for each test requirement.

All the plots in Figure 22 shows that the solving time increases as the array size increases. However, the plots have different slopes. The plot for dom1 appears to fit an exponential model while the other two plots for dom2 and dom3 are nearly linear.

The plot for dom1 shows an abrupt rise in solving time as the array size gets closer to 20. Though nearly linear, the plots between dom2 and dom3 are notably different. The difference in solving time between the plots for dom2 and dom3 become

<table>
<thead>
<tr>
<th>Path ID</th>
<th>arr</th>
<th>low</th>
<th>step</th>
<th>high</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>{7,3,4,7}</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>{6,7,3,1,5}</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P4</td>
<td>don’t care</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>don’t care</td>
<td>don’t care</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Requirement</th>
<th>Path ID</th>
<th>Path</th>
<th>Path Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR1</td>
<td>P1</td>
<td>infeasible</td>
<td>NA</td>
</tr>
<tr>
<td>TR2</td>
<td>P2</td>
<td>&lt;s, N1, N2, N3, N4, N6, N1, e&gt;</td>
<td>7</td>
</tr>
<tr>
<td>TR3</td>
<td>P3</td>
<td>&lt;s, N1, N2, N4, N5, N6, N1, e&gt;</td>
<td>7</td>
</tr>
<tr>
<td>TR4</td>
<td>P4</td>
<td>&lt;s, N1, N6, e&gt;</td>
<td>3</td>
</tr>
</tbody>
</table>

Three of the four basis paths turn out to be feasible. We could not find a feasible path satisfying the test requirement TR1. The reason why the baseline method can possibly take an infeasible method is because all decisions are implicitly assumed to be independent [6]. However, this is not a realistic assumption. The two conditionals “a[i]<max” and “a[i]<min” of the example program affects each other. If one is true, then the other could not become true.

As a result, we are supposed to apply the procedure in Figure 20 to find another feasible basis path. Table 5 shows test inputs that our method generates for each feasible basis paths P2, P3, P4. Even though Table 5 does not include the result of the application of the procedure to the test requirement TR1 for the sake of simplicity, we could find a feasible path <s, N1, N2, N4, N6, N1, e> which is supposed to be executed when the values of the variables low, step are provided such that Arr[low] is equal to Arr[low+step]. The basis path would be added to the feasible basis path set. All of these test inputs force execution of paths of optimal lengths, while satisfying the corresponding test requirements.

We have conducted an experimental study to investigate the effectiveness of our method in terms of solving time. Definitely, the solving time depends on the size of the state space to be explored. In the example program, the array size and integer values of variables determine the size of the state space. The experiment has varied the array size from 5 to 20 by 5 and used integer domains dom1=[0..24], dom2=[0..49], and dom3=[0..99]. As a result, we have the largest search space when the array size is 20 and the integer domain is [0..99]. Its size is 100^20, which is huge.

The experimental results are plotted in Figure 22. The x-axis gives the array size and the y-axis gives the solving time elapsed by the proposed method to find a solution. For each array size and integer domain, we have repeated the experiment five times and calculated the average time for each test requirement.

All the plots in Figure 22 shows that the solving time increases as the array size increases. However, the plots have different slopes. The plot for dom1 appears to fit an exponential model while the other two plots for dom2 and dom3 are nearly linear.

The plot for dom1 shows an abrupt rise in solving time as the array size gets closer to 20. Though nearly linear, the plots between dom2 and dom3 are notably different. The difference in solving time between the plots for dom2 and dom3 become...
larger as the array size increases.

The experimental results indicate that as the state space becomes smaller, the solving time can be reduced more dramatically. For instance, let us consider Table 6 which shows average solving times and their ratios for TR3 with respect to various array sizes and integer domains.

The column ratio1 lists the ratios between solving times for dom1 and dom3 and the column ratio2 lists the ratios between solving times for dom1 and dom2. Remind that dom3 is four times smaller than dom1 and is two times smaller than dom2. The largest ratio is about 68 (≈ 726365/10644). It occurs when the array size is 20 and the integer domain is reduced from dom1 to dom3. When compared to ratio2, it shows the significant reduction in solving time.

We can gain further insight from this study. The state space should be reduced without losing optimal solutions. For instance, consider TR1 which is unsatisfiable. TR1 is unsatisfiable not only on dom1 but also on smaller domains dom2 and dom3. It indicates that if we draw some results on small domains, then the same results will be possibly drawn from larger domains.

![Figure 22. Experimental results](image)

**Table 6:** Ratios between Solving Times for TR3

<table>
<thead>
<tr>
<th>array size</th>
<th>dom1</th>
<th>dom2</th>
<th>dom3</th>
<th>ratio1</th>
<th>ratio2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>85302</td>
<td>8644</td>
<td>1706</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>161087</td>
<td>22710</td>
<td>6234</td>
<td>26</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>295824</td>
<td>73449</td>
<td>9172</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>726365</td>
<td>94749</td>
<td>10644</td>
<td>68</td>
<td>8</td>
</tr>
</tbody>
</table>

In facts, our method relies on the small scope hypothesis in traditional software testing [13, 14]. The small scope hypothesis states that we can detect a high proportion of errors by testing a program for all test inputs within some small scope. This suggests that we can save a lot of testing efforts to test a program exhaustively for some bounded small scope instead of deliberately selecting test inputs from a larger one. Thus, it is important to find a way in which the state space can be reduced while preserving optimal solutions.

**RELATED WORK**

A large portion of test techniques can be divided into two approaches: path-oriented approach and goal-oriented approach [15]. The path-oriented approach demands an entire path going through a target and tries to find test input on which the given path is executed. It often uses symbolic execution. Symbolic execution derives a system of algebraic constraints from a given path in terms of symbols representing any input values that can traverse the path [16]. Solutions to the system of algebraic constraints become test data that cause the execution of the path.

Yan and Zhang proposed a path-oriented method for basis path testing [3]. Their work was motivated by existing basis path testing methods. Poole’s method and the baseline method are well known methods used for generating basis paths from a given program [4, 5]. The methods build the control flow graph of a program to find its basis paths. As a result, basis paths frequently contain infeasible paths. Based on the observation, the method proposed by Yan and Zhang continues to generate paths until the number of feasible basis paths reaches the cyclomatic complexity of the program. However, their method suffers from the same problems as the path-oriented approach. When testing a program containing loops, the number of paths to be explored can be possibly unbounded. Unfortunately, a large portion of the paths is infeasible. Clearly, this wastes a lot of search effort. Even if their method can find desired number of feasible paths, furthermore, some of the paths are redundant. For instance, the method could continuously generate paths of the form <s, (N1, N2, N3, N4, N5, N6) +, N1, e> where ‘+’ denotes multiple occurrences of the sequence within parentheses for the example program in Figure 21. Irrespective of how many times the loop iterates, all paths of the form are infeasible. Generation of those paths causes a large number of attempts before the search procedure terminates and a lot of effort can be wasted. Furthermore, the method would continue the search process until the number of generated feasible paths reaches the cyclomatic number of the program. It could find an additional (feasible) path satisfying TR2, TR3, or TR4 rather than TR1. If the method generates a path of the form <s, (N1, N2, N3, N4, N6) +, N1, e>, the path would be taken into account as an element of the basis path set even if it satisfies TR2.

The goal-oriented approach aims to produce inputs which execute the given target without a specific path. Many goal-oriented techniques can be categorized as dynamic because they require the execution of the program. Dynamic test data generation methods rely on distance function to discriminate between candidate tests in terms of the cost required to
achieve the test goal. One typical example of the goal-oriented approach is the chaining method presented by Ferguson and Korel [17]. The chaining method takes use of data flow dependency to identify statements affecting the branch predicate at which the branch function cannot guide the search and attempts to find inputs that cause those statements to be executed. Because the path taken is determined on the fly, it is very difficult to predict which path will be taken in advance.

Some of the goal-oriented approach us constraint solving techniques for test generation. Gotlieb et al presented a goal-oriented testing method which transforms a program into a constraint system to take advantage of constraint solvers [18]. Unlike the proposed method, however, the method is tightly coupled with the statement coverage or branch coverage criteria.

We can frequently encounter software testing techniques using evolutionary algorithms. Ghiduk proposed an interesting approach to basis path testing using a genetic algorithm, in which the length of each chromosome varies from iteration to iteration according to the change in the length of the path [19]. The approach focused on generation of the basis paths. However, Ghiduk did not describe how to discover test inputs to traverse them.

CONCLUDING REMARKS

We presented a SAT-based test generation method that follows the goal-oriented testing approach for basis path testing. A large proportion of existing test data generation methods has focused on the statement or branch test coverage criteria which require traversal of a selected program point (statement or branch). Unlike the statement or branch test coverage criteria, basis path testing requires multiple program points to be traversed to satisfy test requirements rather than a single program point while satisfying certain constraints among those program points. SAT solving techniques allow us to effectively specify constraints in test requirements. We discussed how to transform test requirements for various test criteria including basis path testing into constraints for SAT solving. One notable benefit of our SAT-based testing is to generate feasible basis paths of optimal lengths. The detection of the infeasibility of a given basis path can also be done as early as possible with the SAT solver. We also presented a procedure for the case where a basis path is infeasible. The procedure can possibly generate another feasible path which is linearly independent with the existing set of basis paths. The proposed method enables us to save a lot of testing efforts by avoiding or reducing trials of generating redundant basis paths.

ACKNOWLEDGEMENTS

This research was financially supported by Hansung University.

REFERENCES

