Using Boolean Satisfiability Solving for Pairwise Test Generation from Cause-Effect Graphs: Comparison of Three Approaches

Insang Chung
Department of Computer Engineering, Hansung University, Seoul, Republic of Korea
insang@hansung.ac.kr

Abstract

Cause-effect graphs can be used for specifying safety critical systems including avionics control software that are often intended to satisfy Boolean expression. Using cause-effect graphs for requirements-based testing demands the ability of dealing with various constraints in cause-effect graphs. Due to its rapid advance, Boolean Satisfiability (SAT) solving seems to be a promising approach for constraint handling. In this paper, we present three approaches using SAT solving for pairwise test generation from cause-effect graphs. One is an ideal approach that tries to obtain a minimal set of tests. Another approach focuses on breaking the problem to smaller solvable problems to scale the applicability of SAT solving. The other makes full use of a partial instance that forms a part of a solution known priori. We compare the three approaches in terms of the number of generated pairwise tests and fault detection capability. Comparison results show that an approach using partial instances can generate less number of pairwise tests than the other two approaches without degrading fault detection capability.

Keywords: Cause-effect graph, Pairwise testing, SAT solving, Partial instances

1. Introduction

The focus of requirements-based testing is to design a necessary and sufficient set of tests from the requirement specifications to ensure that the design and code fully meet the requirements. The cause-effect graph formulates the requirements specification in terms of logical between inputs and outputs of a software system [1]. Recently, the use of Boolean satisfiability (SAT) solving for pairwise test generation from the cause-effect graph has been proposed [2, 3].

Pairwise testing is a combinatorial technique to reduce the total number of tests while still achieving the desired quality. Given a set of input parameters, a pairwise test set consists of tests which capture all possible combinations of pairs of input parameter values. Pairwise testing is based on the premise that most software faults can be captured by either single-value inputs or by an interaction between pairs of input values. Studies have shown pairwise testing to be a very practical and effective software testing criterion even though the size of test sets is dramatically reduced [4-7].

The use of SAT solving for pairwise test generation from the cause-effect graph has been shown to be effective in dealing with constraints on inputs and exploiting the structural information of the cause-effect graph [3]. In the previous study, we developed a tool named “CEGPairGen” that automates the whole process of generating pairwise tests from cause-effect graphs [2]. It produces a test generator written in Java KodKod that generates pairwise tests from a given cause-effect graph. Kodkod is an efficient SAT-based constraint solver for first order logic with relations [8]. It is designed as a plugin component that can be easily incorporated
into other tools and provides a clean Java programming interface that can efficiently construct, manipulate, and solve constraints.

In fact, SAT is a decision problem of determining if there is a value assignment that causes a given Boolean formula to evaluate to true. On the other hand, test generation is an optimization problem where the objective function is to minimize the number of tests. The test generation strategy employed in CEGPairGen tries to obtain a minimal test set by solving a series of decision problems stated as follows: “is there a test set of size \( l \) to cover a given set of pairs?.” If possible, the test set of that size will be optimal in number. The strategy iteratively determines the existence of a test set with varying size until SAT solving computation cannot be completed within a reasonable time.

This paper presents three SAT-based approaches to generating pairwise tests from the cause-effect graph developed by the author. The first one is called SIGA (Simple Incremental Generation Approach). It is an intuitive approach that continues to increment the test set size until a given set of pairs can be satisfied. The second one is called SPBA (Scope Partitioning Based Approach). It is often the case that if the test set size becomes close to a certain value, SAT solving cannot be completed within a reasonable time. Based on this observation, SPBA limits the test set size. Consequently, it leads to many small test sets which cover portions of a given set of pairs. These test sets may have duplicate tests which need to be eliminated when merging them to one test suite. The final one is called PSBA (Partial Solution Based Approach). It makes full use of tests generated in previous iterations to cover new pairs. To do this, PSBA encodes tests already generated as a partial instance. This enables the SAT solver to look for a solution within a more reduced search space.

In this paper, we extend the tool CEGPairGen to make use of partial instances to support PSBA. Previously, it supported only SIGA and SPBA where tests are iteratively generated by solving the conjunction of test predicates representing pairs to be covered over the specification model. As a result, we had to rediscover tests that would be already generated during previous iterations. If we are able to find a way to encode such tests as a partial instance, i.e., a part of a test set, we can expect that the efficiency of CEGPairGen would be improved dramatically in terms of the number of the generated test set.

The rest of the paper is organized as follows. In Section 2, we present a brief overview of the cause-effect graph. Section 3 describes three SAT-based approaches in details. Section 4 presents comparison results of three approaches. Section 5 concludes the paper and presents future extension of the work.

2. Cause-Effect Graphs

A cause-effect graph is originally developed for hardware testing, which is adapted to software testing. It specifies only the desired external behavior of a system by logically relating causes to effects outputs, to produce test cases [1]. A cause represents a distinct input condition that brings about an internal change in the system. An effect represents an output condition, a system transformation or a state resulting from a combination of causes. Basic symbols used in cause-effect graphs are shown in Figure 1.

Each node has the value 0 or 1. The identity relation states that C is equivalent to E. That is, if C is 1, E is 1 or we can say if C is 0, E is 0. The NOT relation states that if C is 1, E is 0 and vice-versa. The OR relation states that if C1 or C2 is 1, E is 1 else E is 0. Similarly, the AND relation states that if both C1, and C2 are 1, E is 1; else E is 0. The AND and OR relations are allowed to have any number of inputs.
Furthermore, a cause-effect graph can specify constraints among causes. Figure 2 shows the constraints expressed in a cause-effect graph.

There are various constraints among causes: E (Exclusive-or), O (One and only one), I (Inclusive-or), and R (Requires). The Exclusive-or constraint states that at most one of the causes C1, C2, and C3 can be 1, i.e. they cannot be simultaneously. The Inclusive-or (at least one) constraint states that at least one of the causes C1, C2 or C3 must be 1. That is, all cannot be 0 simultaneously. The One and only one constraint states that only one of the causes C1 or C2 can be 1. The Requires constraint states that if cause C1 is 1, then cause C2 must be 1. The E, I, and O constraints can be related with any number of causes. In contrast to these constraints on causes, there is one constraint on effects known as Masking (M). The masking constraint states that if effect E1 is 1 then effect E2 is 0.
3. Pairwise Test Generation from Cause-effect Graphs

3.1. Problem Formulation

We encode the desired test set by a two-dimensional grid. Each row in that grid corresponds to a test, while each column corresponds to a node of a given cause-effect graph. In this setting, our goal is to map each row and column pair to a binary value in that cell on the grid in such a way that all of given pairs are to be captured, while the size of the grid is kept as small as possible.

A test set can be formulated in terms of a ternary relation called $TS$ and unary relations, i.e., sets, including $Row$, $Col$, $Cell$, $Node$ and $bf$ as follows:

- $TS$: $Row \rightarrow Col \rightarrow Cell$
- $Row$: $\{1..m\}$ where $m$ is the test set size
- $Col$: $\{1..n\}$ where $n$ is the number of nodes in a cause-effect graph
- $Cell$: $\{0, 1\}$
- $Node$: the set of nodes in a cause-effect graph
- $bf$: $Node \rightarrow Col$ such that $bf(k_1) \neq bf(k_2)$ for $k_1 \neq k_2$

Our pairwise test generation aims at capturing each possible value pairs for all nodes of a given cause-effect graph. We formulate each pair to be covered as a formula as follows:

$$tp = bf(k_1) = m_1 \land bf(k_2) = m_2$$

where $1 \leq k_1, k_2 \leq n$ and $m_1, m_2 \in \{0, 1\}$. Hereafter, we will refer to $tp$ as a test predicate. The test predicate represents that the nodes indexed by $bf(k_1)$ and $bf(k_2)$ have the values $m_1$ and $m_2$, respectively. Even though we can produce test predicates according to various strategies, this paper considers test predicates consisting of all possible value pair for all nodes of the cause-effect graph. Notice that each node can have either 0 or 1 as its value.

Let the set of test predicates to be covered be $TP$. We say that a test set established by the $TS$ relation satisfies the pairwise coverage if all test predicates in $TP$ are satisfied by at least one of its rows. Our goal is to look for a set of assignments to cells on the smallest grid that can satisfy a given set of test predicates.
It is often the case that traditional pairwise testing does not consider relations among input and outputs. It just requires input parameters and values which each input parameter can take on in order to generate pairwise test sets. Such a black-box nature of pairwise testing may miss some important tests [9]. Unlike traditional pairwise testing, we have developed test generation strategies that take into account the constraints on causes and exploit the structural information of the cause-effect graph [2, 3].

Let us take an example. Suppose that the relations, bf(1) and bf(3) contain the column indexes indicating C2 and C4 on the two-dimensional grid that the TS relation establishes, respectively. Then, the Exclusive–or constraint on C2 and C4 of the example cause-effect graph in Figure 3 is represented as the following formula:

\[ \forall r \in \text{Row}: TS(r, \text{bf}(1)) \oplus TS(r, \text{bf}(3)) \]

where ‘\( \oplus \)’ denotes the exclusive-or operator. We also express structural relations in the cause-effect graph by Boolean formulae. Formulation of structural relations is analogous to that of constraints. The following formula represents the “AND” relation among nodes C1, C2, and I1 when bf(0), bf(1), and bf(4) indicate C0, C1, and I1, respectively:

\[ \forall r \in \text{Row}: TS(r, \text{bf}(4)) \leftrightarrow TS(r, \text{bf}(0)) \land TS(r, \text{bf}(1)) \]

Notice that Boolean formulas representing constraints and structural information must hold for all rows of the grid, i.e., for all tests. Hereafter, we will refer to the constraints and the structural relations of a given cause-effect graph expressed in the form of a conjunction of Boolean predicates as the specification model, denoted by M. In this setting, our goal is to find a smallest relation TS where for \( \forall \text{tp} \in \text{TP} \), the formula “\( \text{tp} \land M \)” should be satisfiable.

3.2. Approaches for Pairwise Test Generation

In this section, we present three approaches for pairwise test generation from cause-effect graphs. Before proceeding, we introduce a notion of consistent test predicates. We say that two test predicates \( \text{tp1} \) and \( \text{tp2} \) are consistent if and only if both \( \text{tp1} \) and \( \text{tp2} \) are satisfied by a test. This means that if a certain node happens to be simultaneously in both \( \text{tp1} \) and \( \text{tp2} \), the node must have the same value in those predicates. One of the benefits given by SAT-based test generation is that the SAT analyzer can choose the value of the node to build a test in such a way that “\( \text{tp1} \land \text{tp2} \land M \)” is satisfiable if they are consistent for the given M. Any human intervention is not required during determination of tests.

What happens if two test predicates \( \text{tp1} \) and \( \text{tp2} \) are not consistent with each other? In this case, one test is not enough to make the formula “\( \text{tp1} \land \text{tp2} \land M \)” satisfiable. We need to decompose it into two formulas “\( \text{tp1} \land M \)” and “\( \text{tp2} \land M \)” and deal with each formula separately. This indicates that we must need at least two tests.

Let us extend the notion of consistent test predicates to sets of test predicates. We say that a test predicate \( \text{tp1} \) is consistent with a set of test predicates \( \text{TP} \) if and only if a test predicate \( \text{tp2} \) that is consistent with \( \text{tp1} \) exists in \( \text{TP} \). In order to check if test predicates are consistent with each other, we compose test predicates into one extended test predicate by means of a conjunction of them and solve the composed test predicate by the SAT solver.

3.2.1. Simple Incremental Generation Approach

An intuitive approach to pairwise test generation is to conjunct all test predicates to be covered and solve it at once over the given specification model M. However, it may fail depending on the number of test predicates and the structural complexity of the given
cause-effect graph driven by the number of nodes and constraints over causes. To deal with such a scalability issue, the first approach called SIGA (Simple Incremental Generation Approach) limits the number of test predicates that will be considered at a time.

\[ TS: \text{test suite} \]
\[ TP: \text{a set of test predicates to be covered} \]
\[ cs: \text{current scope} \]
\[ etp, etp': \text{extended test predicate} \]

\[ TS=\emptyset; \text{cs}=1; \]
\[ \text{do} \{ \]
\[ \text{etp} = \text{true}; \]
\[ \text{select a test predicate } tp \text{ from } TP; \]
\[ \text{compose an extended test predicate } etp' \text{ by joining } tp \text{ with } etp; \]
\[ \text{if } (etp' \text{ cannot be solvable in scope } cs) \{ \]
\[ \text{if } (etp' \text{ cannot be solvable in scope } cs+1) \{ \]
\[ \text{mark } tp \text{ as inconsistent}; \]
\[ \text{remove } tp \text{ from } TP \text{ and } etp'; \]
\[ \} \text{ else } cs=cs+1; \]
\[ \} \text{ if } (tp \text{ is consistent}) \{ \]
\[ TS = \text{solve } etp'; \]
\[ etp=etp'; \]
\[ \} \text{ while } (TP\neq\emptyset); \]

Figure 4. Simple Incremental Generation Strategy

Figure 4 shows the procedure of SIGA. It starts with composing each test predicate with the specification model \( \mathcal{M} \) that serves as axioms that any test must satisfy. We say that a test is valid if it satisfies \( \mathcal{M} \). Some test predicates \( tp \) may be impossible to be deduced from \( \mathcal{M} \), i.e., \( \mathcal{M} \not\models tp \). Invalid tests could be generated if we do not consider \( \mathcal{M} \).

To cope with \( \mathcal{M} \) in test generation, we just initialize \( etp \) by \( \mathcal{M} \). Observe that each test predicate is being composed as a conjunction of as many uncovered test predicates as possible. If a newly added test predicate is not consistent with existing conjunction of test predicates, the resulting composed test predicate cannot be solved for the current scope. In that case, we add one additional test by incrementing the scope by one. This process is repeated until all test predicates are covered. Figure 5 shows the relation TS generated using SIGA whenever the scope increases by one.

Figure 5. The TS Relation Generated using SIGA
In the example cause-effect graph, we obtain a final test set consisting of six tests shown in Figure 5 which satisfy the pairwise coverage and constraints over its nodes. For comparison of SIGA to traditional pairwise test generation, let us generate using PICT [10] without considering constraints at all. Then, we have the following 5 tests: (0,1,1,1), (0,0,0,0), (1,0,1,0), (1,1,0,0), and (1,0,0,1) where we show only values causes take. Observe that the test (0,1,1,1) is not valid because there exists the “Require” relation between C1 and C3.

Furthermore, the traditional pairwise generation approach does not include certain interactions of nodes of the cause-effect graphs. For example, the pair (C2:1, E1:1) is not covered by the test set. This indicates that the situation where C2 affects E1 can be ignored. On the other hand, SIGA generates the tests (1,1,0,0) and (0,1,0,0) which cover the pair. We also ensure that all cases where every cause can affect every effect are covered in the test set. However, the case where E2=1 when C2=1 is absent. Since C2 and C4 are related with Exclusive-Or, C4=0 when C2=1, leading to E2=0.

3.2.2. Scope Partitioning based Approach

The second approach is called SPBA (Scope Partitioning Based Approach). This approach is very similar to SIGA. One most important difference from SIGA is that SPBA limits the scope range defining the search space where we are looking for tests while SIGA does not place any limit on scope.

\[
TS: \text{test suite} \\
TP: \text{a set of test predicates to be covered} \\
cs: \text{current scope} \\
etp, \text{etp }': \text{extended test predicate}
\]

\[
TS=\emptyset; \\
do \{ \\
cs=1; \ etp=\text{true}; \\
while (cs<s and TP\neq\emptyset) \{ \\
\quad \text{select a test predicate } tp \text{ from } TP; \\
\quad \text{compose an extended test predicate } etp' \text{ by joining } tp \text{ with } etp; \\
\quad \text{if } (etp' \text{ cannot be solvable in scope } cs) \{ \\
\quad \quad \text{mark } tp \text{ as inconsistent}; \\
\quad \quad \text{remove } tp \text{ from } TP \text{ and } etp'; \\
\quad \} \text{ else } cs=cs+1; \\
\} \text{ if } (tp \text{ is consistent }) \{ \\
\quad TS=\text{solve } etp'; \\
\quad etp=etp'; \\
\} \\
\} \text{ while } (TP\neq\emptyset); \\
\text{Remove redundant tests in } TS;
\]

**Figure 6. Scope Partitioning based Approach**

Figure 6 shows the procedure of SPBA. Basically, it is the “divide-and-compose” strategy. It divides the search space for tests in smaller scope ranges that the SAT solver can process more easily. As in SIGA, SPBA composes as many uncovered test predicates as possible. It also checks the inconsistency of a newly added test predicate by incrementing the scope by one. If the predefined scope limit is reached, both the composed test predicate and the current scope are initialized. This is different from SIGA that increments the range of the scope without any limits. This process is repeated until all test predicates are to be solved. Since our strategy break down the search space to several smaller ones, it may generate some duplicate tests. In the final stage, we remove duplicate
tests to get a final test suite with fewer tests.

The tool CEGTestGen supports SPBA [2]. It produces a test generator written in Java KodKod. The tool starts by accepting one input file representing a cause-effect graph and parameters involving incremental test generation such as scope and maximum time of duration. Execution of the test generator produces pairwise tests in such a way that for each pair of distinct nodes of the input cause-effect graph, every pair of possible values is used in at least one of the tests. Of course, the generated tests are guaranteed to satisfy constraints on causes as well as structural constraints in the cause-effect graph. Figure 7 shows the relation TS generated for the example cause-effect graph using SPBA when the scope limit is set to 2. We have the set consisting of 9 tests that is bigger than that obtained using SIGA.

\[
\begin{array}{cccccccccc}
& C_1 & C_2 & C_3 & C_4 & I_1 & I_2 & I_3 & I_4 & E_1 & E_2 \\
\text{scope=2} & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
& 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
& C_1 & C_2 & C_3 & C_4 & I_1 & I_2 & I_3 & I_4 & E_1 & E_2 \\
\text{scope=4} & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
& 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
& 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
& C_1 & C_2 & C_3 & C_4 & I_1 & I_2 & I_3 & I_4 & E_1 & E_2 \\
\text{scope=6} & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
& 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
& 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
& 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
& 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
& C_1 & C_2 & C_3 & C_4 & I_1 & I_2 & I_3 & I_4 & E_1 & E_2 \\
\text{scope=8} & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
& 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
& 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
& 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
& 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
& 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
& 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
& C_1 & C_2 & C_3 & C_4 & I_1 & I_2 & I_3 & I_4 & E_1 & E_2 \\
\text{scope=9} & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
& 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
& 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
& 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
& 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
& 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
& 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
& 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Figure 7. The TS Relation Generated Using SPBA

3.2.3. Partial Solution based Approach

A closer look at the procedure of SPBA reveals that we do not make use of tests generated during previous iterations to cover new test predicates. Obviously, this process leads to duplicate tests. Based on the observation, we present a new approach called PSBA (Partial Solution Based Approach) that makes advantage of tests generated during the process of solving the conjunction etp. Unlike SIGA and SPBA, it encodes the tests as the partial instance of the problem’s solution, i.e., the part of a final test suite. This is depicted in lines A, B of Figure 6, while the rest of the code is similar to SPBA in Figure
5. The size of the resulting SAT problem can be reduced because test generation is performed in the situation where some tests are directly provided instead of rediscovering them again.

Notice that every relational variable in KodKod must be bound within two relational constants. A relational constant specifies the set of tuples the relational variable may contain. This constant is an upper bound of the variable. The other is a lower bound consisting of tuples the variable must contain. That is, this lower bound is a partial instance. The SAT solver must look for a solution within the provided lower and upper bounds.

\[
\begin{align*}
TS: & \text{ test suite} \\
TP: & \text{ a set of test predicates to be covered} \\
cs: & \text{ current scope} \\
etp, etp': & \text{ extended test predicate}
\end{align*}
\]

\[
\begin{align*}
TS=\emptyset; \\
do \{ \\
\text{cs}=1; \text{etp}=M; \\
\text{while } (cs<s1 \text{ and } TP\neq\emptyset) \{ \\
\text{select a test predicate } tp \text{ from } TP; \\
\text{compose an extended test predicate } etp' \text{ by joining } tp \text{ with } etp; \\
\text{if } (etp' \text{ cannot be solvable in scope } cs) \{ \\
\text{if } (etp' \text{ cannot be solvable in scope } cs+1) \{ \\
\text{mark } tp \text{ as inconsistent}; \\
\text{remove } tp \text{ from } TP \text{ and } etp'; \\
\} \text{ else } cs=cs+1; \\
\} \text{ if } (tp \text{ is consistent }) \{ \\
\text{encode } TS \text{ as the partial instance } PI \text{ -----A} \\
\text{t = solve } etp' \text{ for } PI; \text{ ---------------B} \\
TS=TS \cup t; \\
etp=etp'; \\
\} \\
\} \text{ while } (TP\neq\emptyset);
\end{align*}
\]

**Figure 8. Partial Solution based Approach**

Line A in Figure 8 encodes the test suite obtained in the previous iteration as the partial instance. Encoding is actually done through the function “bounds” that is a part of the test generator in Java produced by CEGPairGen that has been extended to support PSBA for our study. Figure 9 shows the function bounds generated from the cause-effect graph in Figure 3.

We see that the relations mentioned before are bound by the bounds function. A problem’s universe is given as a Universe object constructed from a user-provided collection of objects. Each Universe object provides a TupleFactory for creating constants, represented by TupleSet objects, from atoms drawn from that Universe. The function accepts two integers as its input \( m, n \). They determine the test set size. The number of tests is specified by the Row relation. The Col relation contains integers 1 through \( n \) that index each row (i.e., each test) of the grid. The input parameter \( m \) specifies the column size of the grid. The column size is determined by the number of nodes in the give cause-effect graph. For a given cause-effect graph, each relation \( 
bf[i] \) indicates a node by assigning the same lower and upper bounds using the boundExactly method.

The only relation with different lower and upper bounds is “TS”. It establishes the desired test set by a two-dimensional grid where each row corresponds to a test and each column corresponds to a node of a given cause-effect graph. Its lower bound is given by tuples in the TupleSet “generated” which the previous iteration has generated. In this setting, the SAT solver binds the values of cells in the grid to satisfy a given extended test
predicate, such that the total number of tests is small (line B). Of course, the solution must contain tuples, i.e., tests generated in the previous iteration without the need to rediscover them.

```java
public final Bounds bounds(int m, int n) {
    int max;
    if (m>n) max = m;
    else max = n;
    final Set<Integer> atoms = new LinkedHashSet<Integer>(max);
    for (int i = 1; i <= max; i++) atoms.add(i);
    final Universe u = new Universe(atoms);
    final TupleFactory f = u.factory();
    final Bounds b = new Bounds(u);
    b.bound(Row, f.range(f.tuple(1), f.tuple(m)));
    b.boundExactly(Col, f.range(f.tuple(1), f.tuple(max)));
    b.boundExactly(Cell, f.setOf(1));
    for (int i=1; i<=n; i++) b.boundExactly(bf[i-1], f.setOf(i));
    final TupleSet generated = f.noneOf(3)
    for (Test t: tests) generated.add(f.tuple(t.getRowPosition(), t.getColPosition(), 1);
    b.bound(TS, givens, b.upperBound(Row).product(b.upperBound(Col).product(b.upperBound(Cell))));
    return b;
}
```

Figure 9. The Bounds Function

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope=2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>scope=4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>scope=6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 10. The TS Relation Generated using PSBA

Figure 10 shows the relation TS generated for the example cause-effect graph using PSBA when the scope limit is set to 2. We have the set consisting of 6 tests that happens to be the same as that obtained using SIGA while it is smaller than that obtained using SPBA. In PSBA, we continue to add a new test predicate until the specified scope limit reaches. If the SAT solving fails to find any solutions to the extended test predicate, the current scope value is incremented by one and tries to solve it in that incremented scope. This process continues until test predicates to be covered remain or the specified scope limit reaches. If the specified scope limit reaches and there still remain test predicates, we
reset the scope and the process is repeated for the remaining test predicates.

During the process, PSBA makes full use of previously generated tests to satisfy new test predicates. This is different from SPBA in that once SPBA resets the scope, tests generated during previous iterations do not play a role in covering new test predicates at all. As a consequence, SPBA leads to more test than PSBA does. In addition, SPBA may cause duplicate tests which need to be eliminated.

4. Experiments

This section presents comparison results of SIGA, SPBA, and PSBA. They are applied to the cause-effect graph of the DISPLAY command in [1]. The graph consists of 35 nodes. Among the nodes there are 18 causes and 7 effects. It has 17 “Requires” constraints and one “Exclusive-or” constraint. For the graph, we constructed test suites by using CEGPairGen that has been extended to support the three approaches. Firstly, we used the number of generated tests as a comparison measure. Table 1 shows the number of tests obtained by the three approaches with varying scope limit.

<table>
<thead>
<tr>
<th>Scope</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SPBA</td>
<td>120</td>
<td>105</td>
<td>130</td>
<td>99</td>
<td>96</td>
<td>75</td>
<td>74</td>
<td>58</td>
</tr>
<tr>
<td>PSBA</td>
<td>32</td>
<td>30</td>
<td>27</td>
<td>27</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Cells having as its value ‘N/A’ indicate that we stopped the SAT solving from running after more than 1 hour with no result at all. Note that SIGA does not generate any test sets that satisfy the pairwise coverage for all scopes to be considered. This is because it does not place any limits on the test set size. As a consequence, the search space may grow too large to find a desired test set within a reasonable time as the scope is increasing. This depends on the problem’s complexity derived by the number of test predicates and constraints of the cause-effect graph. Of course, SIGA may find an optimal test set for small problems such as the simple cause-effect graph shown in Figure 3. This observation indicates that SIGA is worth attempting for small problems. SPBA and PSBA may be used when SIGA fails to obtain a test set.

An obvious observation one can make from the table is that the test set size tends to decrease as the scope increases. This looks like a natural consequence because larger scopes indicate that there are less duplicate tests and more test predicates will be covered. Another important observation is that PSBA can significantly reduce the number of generated tests compared to SPBA. The number of tests obtained by PSBA for score 4 is less than half of tests obtained by SPBA for score 8. This is a rather surprising result because PSBA generates much less number of tests even for smaller scopes. This huge difference in the test set size comes from the difference on how to make use of tests generated during previous iterations. PSBA makes full use of previously generated tests in covering new test predicates while SPBA ignores them. Consequently, SPBA may create new tests to cover new pairs that would be covered by previously generated tests. We also observe the greater stability of PSBA in the test set size with varying scope.

We also investigated the effectiveness of test sets produced by SPBA and PSBA with CEGTester [11]. The tool CEGTester is based on the notion of fault-based testing [12-14]. Thus, certain types of faults that may be committed by programmers are firstly hypothesized, and then the testing methodology is applied to the hypothesized faults to show the types of faults that are revealed and those that are not revealed. CEGTester presumes faults which are directly related to cause-effect graphs.

CEGTester generates five fault types: Input Reference Fault (IRF), Input Negation Fault (INF), Expression Negation Fault (ENF), Operator Reference Fault (ORF), and Constraint Reference Fault (CRF). IRF is a fault type where a cause (input) is replaced by
0, 1, or another cause which exists in the cause-effect graph. For an INF type of fault, an occurrence of a cause is replaced by its negation. ENF is a fault where an occurrence of an intermediate node in the cause-effect graph is replaced by its negation. An ORF fault type means that an occurrence of a logical operator AND is replaced by the OR operator or vice versa. CRF concerns dependencies over causes. A CRF is generated by omitting a constraint or by replacing its constraint operator by another constraint operator except the Requires constraint. For example, the Exclusive-or constraint is omitted or is replaced by Inclusive-or constraint or One and only one constraint. Table 2 shows the number of generated faults according to the five types of faults for the graph using CEGTester.

Table 2. Number of Generated Faults

<table>
<thead>
<tr>
<th>Number of generated faults</th>
<th>IRF</th>
<th>INF</th>
<th>ENF</th>
<th>ORF</th>
<th>CRF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>342</td>
<td>25</td>
<td>18</td>
<td>14</td>
<td>196</td>
<td>595</td>
</tr>
</tbody>
</table>

Figure 11. Evaluation Results

The test sets obtained by SPBA and PSBA using CEGPairGen were evaluated with mutation score used in mutation analysis. Mutation score is defined as “Number of faults detected/Number of all faults generated” [15]. Figure 11 shows the mutation scores by fault classes for the generated test sets. We observe that the mutation scores of all of the test sets generated using the two approaches are above 90 for all types of fault classes. For IRF and CRF, SPBA is slightly better than PSBA. However, the difference is considered negligible. The differences between their average mutation scores for IRF and CRF were 0.06 and 0.02, respectively. This result is somewhat surprising because the size of the test set obtained by PSBA is much smaller than that obtained by SPBA. Furthermore, there is no difference between them when the scope limit is set to 1 even though the test set obtained by PSBA is four times smaller than that obtained by SPBA.
5. Concluding Remarks

We described the three SAT-based approaches that generate pairwise tests from a cause-effect graph: SIGA, SPBA, and PSBA. We evaluated them in terms of the number of generated tests and fault detection capability. Experimental results showed that PSBA yields much smaller test sets than the other two approaches without degrading fault detection capability. SIGA and SPBA generate tests by solving the conjunction of test predicates over the specification model. In particular, SPBA has to rediscover tests already generated in previous iterations. In contrast, PSBA encodes previously generated tests as the part of the problem’s solution. Thus, we can make full use of them to cover new pairs, while SPBA ignores them. However this comes with cost of rather long computation time. We plan to conduct more extensive evaluation of PSBA using more complex cause-effect graphs to investigate scalability issues. According to the evaluation results, a possible future direction of the work is to study how to shorten computation time.

Acknowledgements

This research was financially supported by Hansung University.

References

Author

Dr. Insang Chung is a Professor in Department of Computer Engineering, Hansung University, Seoul, S. Korea. He received his Bachelor of Engineering degree at Seoul National University, S. Korea in 1987. He also received MS and PhD in Computer Science from KAIST(Korea Advanced Institute of Science and Technology), S. Korea in 1989 and 1993, respectively. His research interests are in automated test data generation, formal techniques including model checking and testing process for automotive software. He has authored many refereed journals and conference papers about software testing.