http://dx.doi.org/10.5392/IJoC.2014.10.3.041

Modeling Pairwise Test Generation from Cause-Effect Graphs as a Boolean Satisfiability Problem

Insang Chung
Department of Computer Engineering
Hansung University, Seoul, 136-792, Korea

ABSTRACT

A cause-effect graph considers only the desired external behavior of a system by identifying input-output parameter relationships in the specification. When testing a software system with cause-effect graphs, it is important to derive a moderate number of tests while avoiding loss in fault detection ability. Pairwise testing is known to be effective in determining errors while considering only a small portion of the input space. In this paper, we present a new testing technique that generates pairwise tests from a cause-effect graph. We use a Boolean Satisbiability (SAT) solver to generate pairwise tests from a cause-effect graph. The Alloy language is used for encoding the cause-effect graphs and its SAT solver is applied to generate the pairwise tests. Using a SAT solver allows us to effectively manage constraints over the input parameters and facilitates the generation of pairwise tests, even in the situations where other techniques fail to satisfy full pairwise coverage.

Key words*: Cause-Effect Graph, Pairwise Testing, Alloy, SAT problem, Requirements-based Testing.*

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 technique formulates the requirements specification in terms of logical between inputs and outputs of a software system by using Boolean operators like AND, OR, and NOT for system modeling and test design [1]. One of the major challenges in testing a software system with cause-effect graphs is to reduce the total number of tests while still achieving the desired quality.

Pairwise testing is a combinatorial technique used to reduce the number of tests in situations where exhaustive testing is not feasible [2]. Given a set of input parameters, a pairwise test set consists of tests which capture all possible combinations of pairs of input parameter values. For example, consider a system of 30 input parameters where each parameter can be assigned one of 10 values. Exhaustive testing would require the execution of 10^{30} input combinations. On the other hand, there are a total of 100 pairwise tests which capture all possible pairs of input 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.

* *Corresponding author, Email: golsung@naver.com*
Manuscript received May. 21, 2014; revised Jul. 01, 2014; accepted Jul. 08, 2014

Traditional pairwise testing does not consider any relationships 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. The black-box nature of pairwise testing may miss some important tests [3]. For example, consider the following Boolean expressions with three Boolean input parameters (x, y, and z): F_0=*x or y*; F_1=F_0 *or z*. Then, the following test set of 4 tests captures all possible pairs of Boolean values for each pair of the Boolean variables x, y, and z: *(x:T, y:T, z:T), (x:T, y:F, z:F), (x:F, y:T, z:F)*, and *(x:F, y:F, z:T)*. Even when the operator 'or' is changed to 'and', the test set can also be a pairwise test set for the modified Boolean expressions because pairwise testing does not consider how inputs and outputs are related and what operators are used to connect them. Such ignorance in pairwise test generation can miss tests which would reveal certain faults such as ORF (Operator Reference Fault).

We formulate the problem of generating pairwise tests from a cause-effect graph as a SAT (SATisfiability) problem and make advantage of a SAT solver for test generation. The idea of using a SAT solver was introduced in [4]. However, it is developed for test generation from feature diagrams used in the context of software product lines. We adapt the strategy to generate pairwise tests from cause-effect graphs. Specifically, we transform cause effect graphs into Alloy models and then produce pairwise tests via the Alloy analyzer [5]. Furthermore, we enhance traditional pairwise testing to consider how inputs and outputs are related and what operators are used to connect them.

The rest of the paper is organized as follows. In Section 2, we present a brief overview of the cause-effect graph and give

some background on existing pairwise testing techniques. Section 3 presents our approach. Finally, Section 4 concludes the paper and presents some ideas for future extension of this work.

2. RELATED WORK

2.1 Cause-effect Graph

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 to produce test cases. 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 Fig. 1.

Each node has the value 0 or 1. The identity relation is denoted by Iden(C, E). It means that 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 NOT relation is denoted by NOT(C, E). The OR relation, denoted by OR (E, {C1, C2}), states that if C1 or C2 is 1, E is 1 else E is 0. Similarly, the AND relation, denoted by AND(E, {C1, C2}), 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.

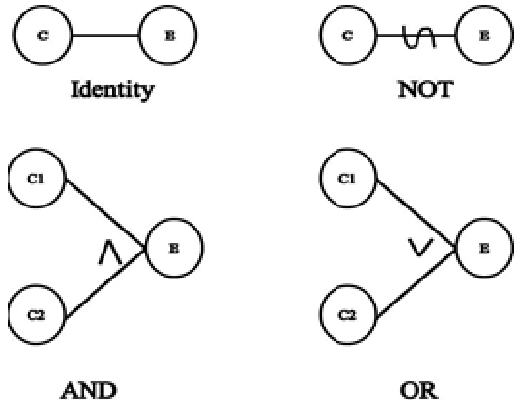

Fig. 1. Basic elements of cause-effect graphs

Furthermore, a cause-effect graph can specify constraints among causes. Fig. 2 shows the constraints expressed in cause-effect graphs: E(*Exclusive-or*), O(*One and only one*), I (*Inclusive-or*), and R (*Requires*). The *Exclusive-or* constraint, denoted by E(C1, C2, C3), states that at most one of the causes C1, C2, and C3 can be 1, i.e. they cannot be 0 simultaneously. The *Inclusive-or* (at least one) constraint, denoted by I(E1, E2, E3), 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, denoted by O(C1, C2), states that only one of the causes C1 or C2 can be 1.

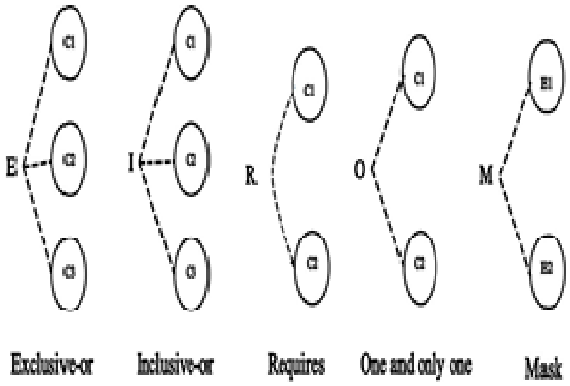

Fig. 2. Constraints of cause-effect graphs

The *Requires* constraint states that if cause C1 is 1, then cause C2 must be 1. The *Requires* constraint is also denoted by R(C1, C2). 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. The masking relation is also denoted by M(E1, E2)

2.2 Pairwise Test Generation

Pairwise testing is the most commonly used form of combinatorial testing which tests at least once all possible combinations for every pair of input parameters of software. As an example, consider software that takes four input parameters, say, x, y, z, and w. If each parameter can have three different values, then there will be 81 different pairs: (x_1, y_1), (x_1, y_2), …, (z_3, w_3). A test (x_1, y_2, z_3, w_2), for example, covers six of these 81 pairs: (x_1, y_2), (x_1, z_3), (x_1, w_2), (y_2, z_3), (y_2, w_2), and (z_3, w_2). In the example, a set of nine tests can capture all 81 pairs.

There is a large body of work on pairwise test generation. The *IPO* (*In-Parameter Order*) strategy builds a test set by repeating the two steps, so-called horizontal growth and vertical growth [6]. The AETG system uses a greedy algorithm which produces a certain number of candidate tests when a new test is needed and selects the one with the largest number of pairs that have not been covered yet [7]. Another interesting approach was presented in [8] which applied genetic algorithms to pairwise testing, and developed a testing tool called "PWiseGen".

Although many pairwise testing techniques have been proposed, little work has been done in the context of generating minimal pairwise tests from the cause-effect graph. In practice, pairwise test generation from the cause-effect graph needs to address the following issues:

- To formalize the cause-effect graph for pairwise test generation,
- To take into account the constraints on causes, and
- To exploit the structural information of the cause-effect graph.

The first issue has not been dealt with in existing pairwise testing techniques because they construct test sets without considering specific testing models. They assume that a testing model consists of input parameters with each parameter having several values [9].

Most pairwise testing techniques either ignore or incompletely address the problem of constraints related to the second issue. For example, meta-heuristic techniques which use genetic algorithms or simulated annealing totally ignore constraints [8], [10]. The AETG system considers only simple constraints of type "*Requires*" [7], [9], while IPO does not support any constraint handling mechanisms. Cohen et. al. investigated the impact of ignoring constraints during test generation and recognized handling constraints as a highly desirable feature of a testing method [11]. They also presented a framework to support combinatorial testing in the presence of constraints. It, however, requires that constraints should be modeled only in a canonical form of Boolean formulae as forbidden tuples which define pairs that cannot occur in a valid test.

The third issue was dealt with the *WBPairwise* algorithm [3]. Unlike traditional pairwise test strategies mentioned in the above, the *WBPairwise* further considers the logical relations between inputs and outputs for pairwise test generation. However, it does not provide any constraint handling techniques. For certain cases, furthermore, the *WBPairwise* algorithm does not generate tests that guarantee full pairwise coverage.

This paper proposes the use of a SAT solver to cope with the above issues. To this aim, we transform the cause-effect graph to the Alloy specification. Then, constraints on causes and any structural relations in the cause-effect graph are expressed as Boolean formulae. Each pair to be covered is also represented as a corresponding Boolean predicate. We use the Alloy analyzer to check the validity of each pair. For only valid pairs, we compose as many as possible pairs into an extended predicate. When composing an extended predicate, we ensure that the pairs do not contradict each other. Using this extended predicate, we derive a pairwise test that covers the pairs in the extended predicate and the newly created test is added to the test suite. This process is repeated until all valid pairs are covered by the test suite.

3. PAIRWISE TEST GENERATION WITH SAT

3.1 Overview of Alloy

Alloy is a formal language which was developed at MIT by Daniel Jackson and his team [5]. Alloy has been applied to modelling and analysis of systems in a wide range of application domains including security analysis [12].

It is supported by Alloy Analyzer, a tool, which allows fully automated analysis. Alloy can specify elements and constraints between them.

The first construct is *Signature*. A signature declares a set of elements and can possibly introduce fields which represent the relationships with other atoms. Constraints are defined by facts, predicates and functions. Facts are axioms that are intended to always hold. Predicates are parameterized constraints which can evaluate to true or false. We use the Alloy analyzer to find instances that satisfy all constraints and evaluate one predicate to true. The search space in which Alloy looks for solutions is limited by the scope which is the maximum number of instances for each signature.

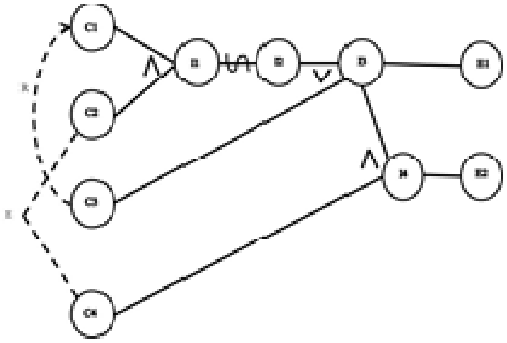

Fig. 3. An example cause-effect graph

3.2 Transformation rules

Firstly, a signature is generated for each node of a cause-effect graph. For example, there are 10 nodes in the cause-effect diagram, shown in Fig. 3. The transformation generates 10 signatures to represent these nodes as shown in Fig. 4.

```
one sig I2 {}
one sig I3 {}
one sig I4 {}
one sig C1 {}
one sig C2 {}
one sig C3 {}
one sig C4 {}
one sig E1 {}
one sig E2 {}
}
```

Fig. 4. Signatures corresponding to nodes of the cause-effect graph in Fig. 3

We also declare a signature Config which specifies the semantics of a cause-effect graph. A configuration is mapped to a set of nodes and is a basis from which a test can be generated. The semantics of a cause-effect graph is the set of all possible configurations that satisfy all constraints among nodes. The *Config* signature corresponding to the cause-effect graph in Fig. 3 is shown in Fig. 5.

The relations f1~f10 declared in the *Config* signature maps each configuration instance to at most one instance in the corresponding signature. For example, if c.f5 exists for a configuration instance c, it indicates that the cause C1 is true.

```
sig Config {
        f1: lone I1;
        f2: lone I2;
        f3: lone I3;
        f4: lone I4;
        f5: lone C1;
        f6: lone C2;
        f7: lone C3;
        f8: lone C4;
        f9: lone E1;
        f10: lone E2;
}
```

Fig. 5. A *Config* signature

Furthermore, the constraints among the nodes are transformed to Alloy facts because they must hold for all configurations. Table 1 shows the transformation rules for Alloy facts.

Table 1. Transformation rules for Alloy facts

Constraints	Alloy facts
Ident(N1, N2)	all c: Config | #c.f1=#c.f2
AND(N1, {N2,…,Nm})	all c: Config | #c.f2=1 and #c.f3=1 and … and #c.fm=1<=>c.f1=1
OR(N1, {N2,…,Nm})	All c:Config| #(c.f2+...+c.fm)>=1<=> c.f1=1
NOT(N1, N2)	all c: Config | #c.f1=0<=>#c.f2=1
E(N1, …, Nm)	all c: Config |#(c.f2+...+c.fm)<=1
I(N1, …, Nm)	all c: Config |#(c.f2+...+c.fm)>=1
O(N1, …, Nm)	all c: Config |#(c.f2+...+c.fm)=1
R(N1, N2)	all c: Config |#c.f1=1=>c.f2=1
M(N1, N2)	all c: Config |#c.f1=1=>c.f2=0

Fig. 6 shows the Alloy facts which represent the relations and constraints in the cause-effect graph of Fig. 3.

```
fact {
        all c: Config | (#c.f5=1 and #c.f6=1) <=> #c.f1 = 1
        all c: Config | (#c.f2 = 1 <=> #c.f1=0)
        all c: Config | #(c.f2+c.f7)>=1 <=> #c.f3=1
        all c: Config | #c.f3= 1 <=> #c.f9=1
        all c: Config | (#c.f3=1 and #c.f8=1) <=> #c.f4=1
        all c: Config | #c.f4=1 <=> #c.f10=1
        all c: Config | #(c.f8+c.f6)<=1
        all c: Config | #c.f7=1=>#c.f5=1
}
```

Fig. 6. Alloy facts for the cause-effect graph in Fig. 3

3.3 Pairwise Test Generation

Once a cause-effect graph has transformed to the Alloy model, we generate all possible combinations of pairs of nodes of the graph. If there are N nodes, there are a total of ${}_{2N}C_2$. Of course, we need to remove the pairs with repetitions of the same node. For example, the cause-effect graph in Fig. 1 generates 180 pairwise tests with 10 repetitions of the same node removed.

In the next step, we need to eliminate the pairs which are not valid with respect to the Alloy model. A combination of nodes is said to be valid if it satisfies the semantics of the cause-effect graph. In order to determine if a pair satisfies the semantics of a cause-effect graph, we just check if there exists a configuration where the pair holds true. For example, consider the situation which both of the causes C2 and C4 in Fig. 1 are true. We know that C2 and C4 can never hold simultaneously because an *Exclusive-or* constraint between them exists. In the first place, we formulate the pair as the following Alloy predicate:

```
pred t68 {
        some c: Config |    #c.f6=1 and #c.f8=1
}
```

We perform the semantic check by giving a bound of exactly one on the Configuration signature. Then, the Alloy analyzer looks for an instance of the predicate, which would fail in the present case. In the example cause-effect graph, 36 invalid pairs are captured, leading to 144 valid pairs which should be covered by tests.

Finally, we generate pairwise tests from the Alloy model with valid pairs. Observe that we are concerned with tests consisting of only causes. Thus, nodes except causes need to be projected out from valid configurations. In the case of the example Alloy model, this can be done with the following predicate:

```
pred construct_test[c: Config, t: Test] {
        #t.t1=#c.f5 and
        #t.t2=#c.f6 and
        #t.t3=#c.f7 and
        #t.t4=#c.f8
}

sig Test {
        t1: lone C1,
        t2: lone C2,
        t3: lone C3,
        t4: lone C4
}
```

For example, in order to determine if it is possible to derive a test which cover two pairs, (C1:1, C2:0) and (C4:1, I1:0), the following predicate is employed:

```
pred testGen {
    some t: Test|some c: Config |    #c.f5=1 and #c.f6=0 and
    construct_test[c, t]
    some t: Test|some c: Config |    #c.f8=1 and #c.f1=0 and
    construct_test[c, t]
}
```

We can generate pairwise tests by including all the Alloy representations of valid pairs in the above 'testGen' predicate and solve the resulting Alloy model at once. However this approach may fail if the number of pairs to be covered is huge.

In this paper, incremental test generation is performed. In the incremental approach, we continue to add a new pair until solutions can be found within the specified scope. If the Alloy analyzer fails to find any solutions to the Alloy model, the current scope value is incremented by one and tries to solve the model in that incremented scope. This process continues until no more pairs to be considered exist or the specified scope is reached. If the specified scope is reached and there still remain pairs to be considered, we reset the scope and the process is repeated for the remaining pairs. We then delete all the redundant tests in the final test suite.

Table 2. A test set generated using our approach

C1	1	1	0	1	1	0
C2	0	1	1	1	0	0
C3	0	0	0	1	1	0

C4	0	0	0	0	1	1
I1	0	1	0	1	0	0
I2	1	0	1	0	1	1
I3	1	0	1	1	1	1
I4	0	0	0	0	1	1
E1	1	0	1	1	1	1
E2	0	0	0	0	1	1

In the example cause-effect graph, 6 tests are generated as follows: (1,0,0,0), (1,1,0,0), (0,1,0,0), (1,1,1,0), (1,0,1,1), and (0,0,0,1). Here the value in the i-th position of each test corresponds to Ci. If traditional pairwise test generation is carried out for the example using PICT developed at Microsoft [13], the following 5 tests would be generated: (0,1,1,1), (0,0,0,0), (1,0,1,0), (1,1,0,0), and (1,0,0,1). However, the test (0,1,1,1) is not valid because there exists the *Require* relation between C1 and C3. Table 2 shows the resulting test sets when using our approach.

Note that the traditional 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. The proposed approach generates the tests (1,1,1,0) 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.

The most relevant work to our technique is the *WBPairwise* algorithm [3]. Unlike traditional strategies mentioned in the above, the *WBPairwise* further considers internal operations which combine inputs. However, it does not deal with the constraints among the causes such as *Exclusive-or*. For certain cases, furthermore, the *WBPairwise* algorithm does not guarantee full pairwise coverage because it depends on which test sets are firstly generated on each node.

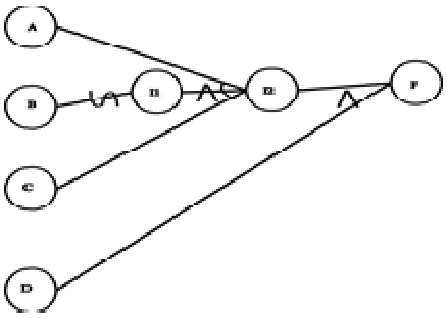

Fig. 7. The cause-effect graph where the *WBPairwise* fails to satisfy full pairwise coverage

For example, if the *WBPairwise* algorithm generates tests which cause only one Boolean value (either True or False) in the node I2 in Fig. 7, then it does not generate tests which cover every pairwise combination of the two nodes D and I2.

Table 3. Pairwise test set generated by our approach for the cause-effect graph in Fig. 7

A	1	1	0	1	0	0
B	0	1	1	0	0	0
C	1	0	1	1	1	0
D	0	0	0	1	1	0

The work in this paper is based on the observation that test generation problem can be formulated as a SAT problem and take advantage of a SAT solver for solutions. Consequently, we can readily cope with any constraints among inputs and outputs. As far as Alloy models for test generation can be solved by its SAT solvers, our SAT-based technique can generate pairwise tests even in situations where other methods including the *WBPairwise* algorithm fails. Furthermore our approach can generate all possible optimal pairwise tests because Alloy performs exhaustive search for given bounds. For example, Table 3 shows one of optimal test sets generated by our SAT-based technique for the cause-effect graph in Fig. 7.

4. CONCLUDING REMARKS

We presented a technique which generates pairwise tests from a cause-effect graph. The work in this paper is based on the observation that the problem of generating pairwise tests from a cause-effect graph can be formulated as a SAT problem and we can take advantage of a SAT solver for effective pairwise test generation. Using a SAT solver allows us to effectively handle any constraints among inputs. In addition, our approach can generate all possible optimal pairwise tests because Alloy performs exhaustive search for given bounds. In contrast, previous pairwise test generation approaches remain uncertain if the generated tests are optimal in terms of the number of tests and also are not able to enumerate all possible optimal pairwise tests. In order to produce more effective pairwise tests, our technique considers how inputs and outputs are related and what operators are used to connect them.

As future work, we plan to conduct more extensive evaluation of our approach using more complex cause-effect graphs to investigate scalability issues.

ACKNOWLEGEMENTS

This research was financially supported by Hansung University.

REFERENCES

[1] G. J. Meyers, *The Art of Software Testing*, John Wiley & Sons, 1979.
[2] C. Nie and H. Leung, "A survey of combinatorial testing," ACM Computing Surveys, vol. 43, no. 2, 2011.
[3] J. Kim, K. Choi, D. M. Hoffman, and G. Jung, "White box pairwise test case generation," Proc. 7th International Conference on Quality Software, 2007, pp. 289-291.

[4] G. Perrouin, S. Sen, J. Klein, J. B. Baudry, and Y. le Traon, "Automated and scalable t-wise test case generation strategies for software product lines," Proc. ICST2010, pp. 459-468.
[5] D. Jackson, *Software Abstractions*, The MIT Press, 2006.
[6] K. C. Tai and Yu Lei, "A test generation strategy for pairwise testing," IEEE Trans. on Software Engineering, vol. 28, no. 1, 2002, pp.109-111.
[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design," IEEE Trans. on Software Engineering, vol. 23, no. 7, 1997, pp. 437-444.
[8] Pedro Flores and Y. Cheon, "PWiseGen: generating test Cases for pairwise testing," Proc. the 2011 IEEE International Conference on Computer Science and Automation Engineering (CSAE 2011), 2011, pp. 747-752.
[9] C. Lott, A. Jain, and S. Dalal, "Modeling requirements for combinatorial software testing," Proc. the 1st International Workshop on Advances in Model-based Testing, 2005, pp. 1-7.
[10] M. Patil and P. J. Nikumbh, "Pair-wise testing using simulated annealing," Proc. 2nd International Conference on Computer, Communication, Control and Information Technology (C3IT-2012), Feb. 2012, pp. 25-26.
[11] M. B. Cohen, M. B. Dwyer, and J. Shi. "Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach," IEEE Transactions on Software Engineering, vol. 34, no. 5, 2008, pp. 633-650.
[12] Julien Brunel, David Chemouil, Vincent Ibanez, and Nicolas Meledo, "Formal Modelling and Safety Analysis of an Avionic Functional Architecture with Alloy," Proc. ERTS^2 (Embedded Real Time Software and Systems), 2014.
[13] J. Czerwonka, "PICT-Pairwise testing in the real world: practical extensions to test-case scenarios," Proc. the 24th Pacific Northwest Software Quality Conference, Oct. 2006, pp. 419-430.

Insang Chung

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 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.