HIGH-LEVEL CONSTRAINT SUPPORT FOR COMBINATORIAL TESTING

by

ANTHONY C. OPARA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2008

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Dr. Lei, whose assistance, stimulating suggestions, thoroughness and encouragement helped me throughout the duration of my research work and thesis writing. I am also grateful to Mr. David Levine and Dr. Donggang Liu for serving as committee members for my defense. Finally, I am grateful to members of my family and friends for all their support and motivation throughout my research work.

July 3, 2008

ABSTRACT

HIGH-LEVEL CONSTRAINT SUPPORT FOR COMBINATORIAL TESTING

ANTHONY C. OPARA, M.S.

The University of Texas at Arlington, 2008

Supervising Professor: Dr. Jeff Yu Lei

Combinatorial testing constructs test cases by combining different input parameter values based on some effective combinatorial strategy. This software testing approach has displayed very promising attributes and is rapidly gaining popularity in recent years. However, existing work does not provide adequate support for constraint handling. Constraints are often specified as part of an input parameter model and they may be due to several reasons such as incompatibility between certain hardware and software components. A test generation algorithm needs to take these constraints into account during the test generation process to exclude combinations that are invalid from the domain semantics**.**

In this thesis, we describe a general approach to handling constraints for combinatorial testing. Our approach includes a formal notation that allows the user to specify constraints at a higher level of abstraction. We discuss how to deal with the problem of “future conflicts”, which arises when a selected value satisfies all the constraints at one point in the test generation process but fails to satisfy some constraints in the future. Our approach can be combined with different combinatorial test generation algorithms, and we demonstrate this by showing how to extend an existing combinatorial test generation algorithm, called In-Parameter-Order-General

(IPOG), to handle constraints. We describe a Java based combinatorial testing tool developed in the course of this research work called FireEye, which implements an extended version of IPOG that supports constraint handling, and report some experimental results that demonstrate the effectiveness of our approach.

.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

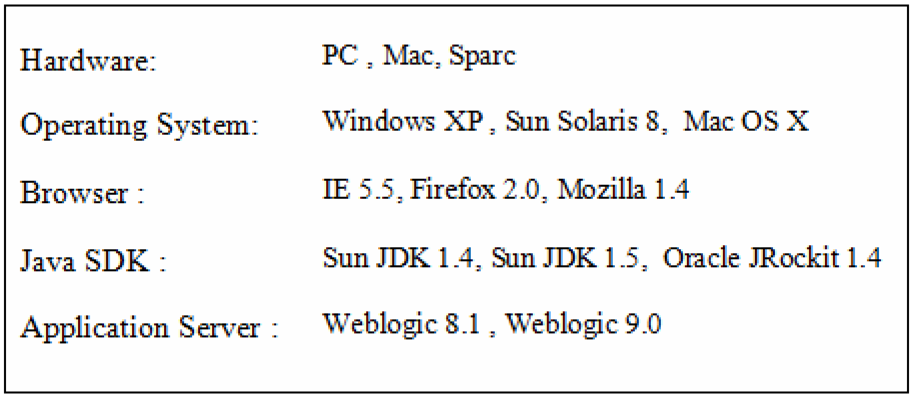
1.1 Overview

The trend in software development is increasingly moving towards the development of software systems that are larger, complex, more distributed and highly configurable. While these complex systems solve many complex problems, they are more error prone, thus the need for more detailed testing requirements. Many of today's systems are built to be run on different operating systems and hardware configurations and consist of various components interacting to render software services to users. These components, which may have multiple versions, are configured by manipulating several compile and run-time options. Each new configuration item is often associated with a combinatorial increase in the number of potential runtime configurations and could possibly behave differently on different platforms or even possess different bugs. Thus, ideally each configuration will have to be tested.

Software Testing, which is the process of executing a program with the intent of finding faults in a program or system, is one of the approaches that can be used to ensure software quality in today's systems. The software testing process begins with a test data generation stage followed by a test execution and then the test evaluation stage. During test generation, data which will be fed the system during the test execution phase is generated. The quality of the test data determines how effective the testing result will turn out. For large and complex systems, it is often impossible to test every combination of interacting parameters; hence it is necessary to reduce the set of test configurations, while still satisfying the required test coverage. Combinatorial testing is one approach to software testing that generates test cases selectively and not exhaustively, by combining values of different input parameters, based on some combinatorial strategy. A considerable number of combinatorial testing tools and algorithms exist, that can be used to test highly configurable, complex and large systems, and some examples are IPOG [2], AETG [5], DDA [3] and PICT [8], which implement one of several combinatorial testing strategies. From the domain semantics, some combination of parameter–values for a given system may be invalid or forbidden and will have to be excluded during test generation. Invalid parameter-value combinations are known as constraints and are often defined as part of the input parameter model of a system. Existing combinatorial testing tools do not provide adequate support for constraint handling and this is a big limitation to the combinatorial testing approach especially with testing today's systems. Existing support for constraints includes one or more of the following restrictions [1]:

1. Requiring the user to explicitly define every illegal configuration.
2. Requiring the user to re-model the input model into disjoint subsets of valid Combinations.
3. The use of a proprietary constraint handling method that cannot be re-implemented.
4. The use of third party constraint handling tools, which is often tweaked to work with existing test generation tools

Having the user enumerate every illegal configuration or re-model the input model is time consuming and error-prone, while the use of third party constraint handling tools often lead to unpredictable application behavior, arising from an inflexible integration with existing test generation tools constraints. Figure 1 presents a model used to perform a compatibility test of web-based distributed banking software. This system can run on different platforms and we want to test that any claim such as: "our banking software can run on windows XP OS and Mac OSX" are valid. We consider a combination of Hardware (PC, Mac, Sparc), Operating system (Windows XP, Sun Solaris 8, ac OS X) ,Browser(IE5.5, Firefox 2.0, Mozilla 1.4), Java SDK (Sun JDK 1.4, Sun JDK 1.5, Oracle JRockit 1.4) and Application server (Weblogic 8.1, Weblogic 9.0), which represents different configurations supported by our banking application. There are a total of 162, (3 x 3 x 3 x 3 x 2), different combinations of the input model parameter values and to fully test the system, we require a test suite for each configuration. For each test suite that is run, it is possible that a different behavior is exhibited by the system and a different set of bugs may be revealed in the application. For instance, a problem with Firefox browser may not be revealed under a Sun JDK 1.5 Plug-in but may be revealed under the JRockit 1.4 Plug-in.



Hardware:	PC , Mac, Sparc
Operating System:	Windows XP , Sun Solaris 8, Mac OS X
Browser :	IE 5.5, Firefox 2.0, Mozilla 1.4
Java SDK :	Sun JDK 1.4, Sun JDK 1.5, Oracle JRockit 1.4
Application Server :	Weblogic 8.1 , Weblogic 9.0

Figure 1: Input Model for web application compliance testing

As with most real life applications, the input model parameters are not independent of each other and some interacting parameters may have combinations, which represent limitations to the input domain. For instance, PC's, Macs and Sparc hardware will run their dedicated operating systems. This represents a constraint on our input model and in this case is due to inconsistencies between hardware and software components. The following is a list of valid constraints placed on our input model:

1. MAC hardware **requires** a MAC OS X operating system.
2. Sparc hardware **requires** a Sun Solaris 8 operating system.
3. PC hardware **requires** a Windows XP operating system.
4. A combination of Weblogic 8.1 Application Server and Oracle JRockit 1.4 Java SDK **requires** Firefox 2.0 Browser.
5. MAC OS X **cannot occur** with Weblogic 8.1 Application server.

Any test case that includes one or more of the invalid combinations, which represents the constraints listed above, will fail to execute properly in the current domain context. For instance, a test case (MAC OS X, Weblogic 8.1) cannot be executed successfully and must be excluded from the resulting test suite. A test that includes an invalid combination like (SPARC, Windows XP) will be rejected or fail during execution and this may compromise test coverage, if some valid combinations are only covered by this test. It would appear that removing such offending test cases from the resulting test suite is a sufficient solution. However, such a test case may cover other valid combinations not covered else where in the test suite.

This thesis describes our constraint handling strategy, which includes a formal notation that allows the user to specify constraints at a higher level of abstraction. Our approach to handling constraints in combinatorial testing deals with the problem of future conflicts and can be integrated seamlessly with any existing test generation algorithm.

We also describe FireEye, a java-based combinatorial testing tool we developed, which implements an extended version of the IPOG test generation algorithm.

1.2 Structure of Thesis

The rest of this thesis is organized as follows: In the next chapter, we provide an overview of the IPOG strategy and a survey of some existing support for constraint handling. Chapter 3 describes our approach to constraint handling, which is to treat the problem as a constraint satisfaction problem. In Chapter 4, we present our algorithmic support for to handling constraints in a given constrained input model. Chapter 5 describes the FireEye tool and highlights its major functionalities and chapter 6 presents some experimental results used to validate our constraint handling strategy. Finally, we present our conclusion in Chapter 7.

CHAPTER 2

RELATED WORK

2.1 Overview

A considerable amount of work has been done on combinatorial testing strategies in the past, and one of the most prominent strategies is pairwise testing, which requires that for each pair of input parameter of a system, every valid combination of values of these pair be covered by at least one test case [2].

A generalization of pairwise testing is t-way or t-wise testing, and it requires that every combination of any t parameter values be covered by at least one test, where t is known at the strength of coverage. We present below an overview of the IPOG (In-Parameter-Order-General) strategy, which implements the t-way testing strategy and provides a seamless framework through which we provide an extension to handle constraints. We also examine current support for constraints by existing algorithms and tools.

2.2 The IPOG Testing Strategy

The IPOG strategy extends the IPO (In-parameter-order) strategy to enable the generation of a t-way test set for a given input model. The IPOG framework is defined as follows: Given a system with t or more parameters, the IPOG strategy generates a t-way test set for the first t parameters, it then extends the test set to generate a t-way test set for the first t + 1 parameters, and continues with the extension of the test set for each additional parameter, until a t-way test set for every parameter has been generated. For an additional parameter, an existing test set is extended in two steps:

- Horizontal growth: An existing test set is extended by adding a value for the new parameter. The new value is selected in a greedy approach, where the goal is to choose the value that covers the largest number of combinations.

- Vertical growth: A new test is added to the test suite, if no existing test can be changed to cover the new combination.

The IPOG strategy is deterministic and can be easily applied to general software applications. To illustrate the IPOG test generation strategy, we use a sample system with four parameters Browser: [IE, Firefox], OS: [Windows, Linux], Cache: [true, false] and RAM(GB): [1, 2, 4] as input to algorithm IPOG and describe the key steps in the algorithm to generate a 3-way test set:

- Initialize test set (ts) to zero, put input parameters in an arbitrary order, and add into ts a test for every combination of the first t parameters. In our case, t = 3 and after execution of these procedures we have the following result:

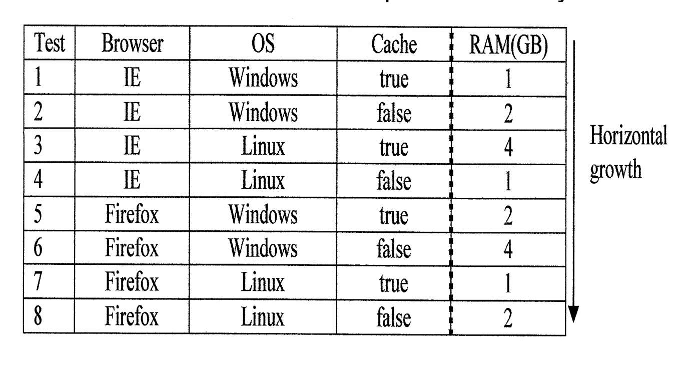
Table 1: 8 possible combinations of the first 3 parameters

Test	Browser	OS	Cache
1	IE	Windows	true
2	IE	Windows	false
3	IE	Linux	true
4	IE	Linux	false
5	Firefox	Windows	true
6	Firefox	Windows	false
7	Firefox	Linux	true
8	Firefox	Linux	False

- Compute the set of all t-way combinations that must be covered in order to cover Pi, where i is the current parameter chosen by the algorithm. In the case of our system, i = RAM (GB) and the system needs to cover all 3-way combinations of the following parameter groups, (Browser, OS, RAM (GB)), (Browser, Cache, RAM (GB)), (OS, Cache, RAM (GB)).

- Choose one value from RAM (GB) greedily (value covers the largest number of combinations) and extend the test set by adding the new value - horizontal growth. The result of the horizontal extension is shown in Table 2.

Table 2: Horizontal extension step for Internet system

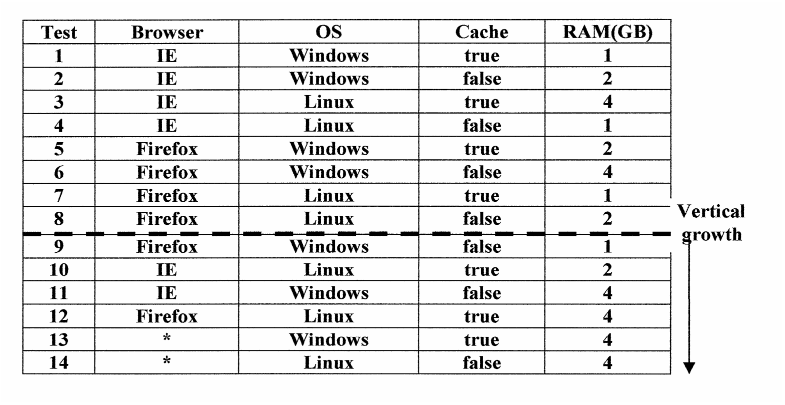


Test	Browser	OS	Cache	RAM(GB)
1	IE	Windows	true	1
2	IE	Windows	false	2
3	IE	Linux	true	4
4	IE	Linux	false	1
5	Firefox	Windows	true	2
6	Firefox	Windows	false	4
7	Firefox	Linux	true	1
8	Firefox	Linux	false	2

Horizontal growth

- Cover the remaining uncovered combinations, one at a time by either adding a new test or changing a test to cover a combination. The latter operation can be performed only on don’t care values (or values that can be replaced by any value without affecting the test set coverage), while the former is executed if no existing test can be changed. For our system above, after horizontal extension, the following combinations have not been covered, (Firefox, Windows, 1), (Firefox, Linux, 3), (IE, Windows, 3), (Windows false, 1) etc. To cover these combinations, new tests have to be added. For instance to add a test for (Firefox, Linux, 3), a vertical extension is performed, by adding a new test (Firefox, Windows, *, 1), test 9, to figure 3 Table 2 above, where * represents don’t care value. To cover the combination (Windows, false, 1), the algorithm will change the value of cache from * to 1 in the 9th test above. Table 3 below shows the result of the vertical extension step of IPOG.

Table 3: Vertical extension for Internet system



Test	Browser	OS	Cache	RAM(GB)
1	IE	Windows	true	1
2	IE	Windows	false	2
3	IE	Linux	true	4
4	IE	Linux	false	1
5	Firefox	Windows	true	2
6	Firefox	Windows	false	4
7	Firefox	Linux	true	1
8	Firefox	Linux	false	2
9	Firefox	Windows	false	1
10	IE	Linux	true	2
11	IE	Windows	false	4
12	Firefox	Linux	true	4
13	*	Windows	true	4
14	*	Linux	false	4

Vertical growth

2.3 Existing Constraint Support

While the concept of constraints has often been described and several special handling strategies have been proposed, it appears that there is no general, re-implementation solution to the problem of handling constraints. A general and re-implementable solution that will keep the burden of manipulating constraints off the software tester is required. In this section, we give a brief survey of constraint handling by existing tools and algorithms.

2.3.1 Deterministic Density Algorithm (DDA)

The DDA algorithm is a greedy algorithm which generates a test suite by adding one test at a time to a test suite. For each subsequent test to be added many are created and then the test that covers the most combinations is chosen. DDA constructs one row of a test suite at a time using a steepest ascent approach [3].

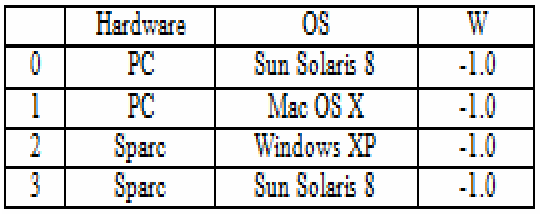
The authors classify constraints into hard constraints, which represents forbidden configurations and soft constraints which are either avoids or neutral configurations. Avoids are combinations that are allowed but are undesirable in a test suite, while neutral combinations are combinations that have no coverage preference.

To exclude a hard constraint in DDA, the weight of the forbidden configuration is w = -1.0, for example the first constraint listed for our banking system in figure 1, is a hard constraint hence the tuple (MAC, Windows XP) is forbidden and must be assigned a weight w = -1.0.

To handle avoid combinations, a negative weight, -1.0 < w < 0.0 is assigned to the combinations and the magnitude of the negative weight chosen indicates the importance of avoiding the tuple. Neutral combinations are assigned a weight of w = 0.0

The weights assigned to a constrained input model in DDA may be applied to individual combinations or assigned to individual factors or levels. In the latter case, the algorithm calculates the weights by multiplying the (factor, value) weights between pairs. Table 4 shows weights assigned to hard constraints in figure 1, between hardware and OS.

Table 4: Weights of constraint pairs for DDA



	Hardware	OS	W
0	PC	Sun Solaris 8	-1.0
1	PC	Mac OS X	-1.0
2	Sparc	Windows XP	-1.0
3	Sparc	Sun Solaris 8	-1.0

The DDA algorithm currently supports only soft constraints and according to them, hard constraints are referred to as “constraint satisfaction” problem. We believe the requirement for specifying constraints in DDA is both time-consuming and error-prone since the user will have to specify a weight for every invalid combination. The fact that hard constraints are not handled also means that for some input spaces, forbidden configurations which are not testable have no way of being excluded from the resulting test suite and this could be problematic in some test environments.

2.3.2 Automatic Efficient Test Generator (AETG)

AETG is a one-row-at-a-time greedy algorithm for generating covering arrays. To generate one row, the first t-tuple is selected based on the one involved in the most uncovered pairs, while the remaining parameters are assigned levels randomly. Levels in turn are selected based on the one that covers the most new t-tuple [5]. In AETG, constraint handling involves the use of multiple relations or explicitly defining disallowed tests.

2.3.2.1 Constraints as multiple relations

In this method, a constrained input model is rewritten into two or smaller constraint-free input models or relations. Test cases are then generated for each input model or relation and the final test suite is constructed by merging the different test suites from the multiple relations. To construct multiple relations for a constrained model, a parameter P involved in a constraint with the least number of values is selected and then the input model is split into intermediate models, one for each value of the selected parameter P. Next, the intermediate sub-models that contain constraints involving the selected parameter are further split by removing values of the other parameter such that the constraints are eliminated. This last step is applied recursively until all constraints have been eliminated. This method is also considered to be error-prone, heavily time consuming and inefficient for dealing with implicit constraints.

2.3.2.2 Constraints as disallowed tests

This method requires the user to explicitly supply a list of invalid sub-combinations. A disallowed test for a relation specifies a set of test cases that are not valid for that relation. The AETG combination strategy does not select any test cases including the invalid sub-combinations; instead it finds valid test cases to satisfy the desired coverage.

2*.3.3 Intelligent Test Case Generator (Whitch)*

Whitch includes two algorithms for finding coverage arrays, Combinatorial Test Services (CTS) and Test Optimizer of Functional Usage (TOFU). Both algorithms use algebraic constructions to generate covering arrays and constraint handling requires that invalid combinations be expressed as all possible invalid configurations [6]. Consider the constraint in figure 1, (MAC hardware **requires** a MAC OS X operating system). This implies that the combination (MAC, Windows XP) is invalid in our domain context. In whitch, an enumeration of all combinations containing MAC and Windows XP is required for the banking systems. There are two of such configurations - (MAC, Windows XP), (MAC, Sun Solaris 8), for each hardware and operating system configuration. As the number of parameters and values increase, the set of invalid combinations may explode. Hence this method may not be feasible in some cases.

2.3.4 Pairwise Independent Combinatorial Testing tool (PICT)

PICT uses an algorithm similar to AETG with optimizations to increase execution speed. In PICT [7], constraints are specified in the form of propositional formulas, where IF-THEN statements describe constrained value combinations in a test domain.

In [7], a description of the constraint handling support says that constraints are translated into a set of exclusions (invalid combinations) at the preparation phase and that test sets produced after the preparation phase are generated not to violate any exclusions. We do not have enough details on the method for avoiding the problem of removing an invalid test from a test not covered anywhere else and also no clear way of re-implementing the system. We also note that PICT does not support constraints involving mathematical expressions. For instance, the constraint [P1* 2 = P2 => P3 = 100] can not be specified in PICT

CHAPTER 3

A CONSTRAINT HANDLING STRATEGY

3.1 Overview

In this chapter, we describe our approach to handling constraints in an input model. Our strategy consist of : (1) The use of a constraint grammar, which allows the specification of constraints at a higher level of abstraction (2) treating constraints as a constraint satisfaction problem and using forward checking to detect future conflicts. We also describe the different classes of constraints and how we handle each class.

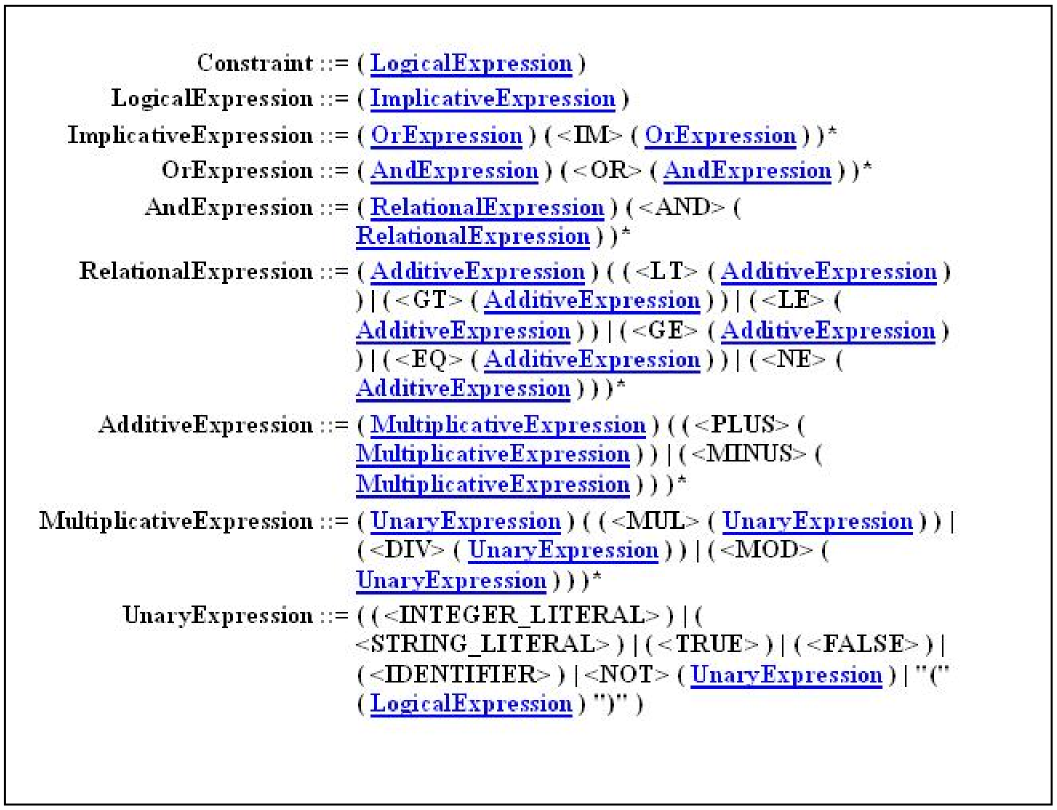
3.2 Constraint Specification

Constraints can be specified in a number of different forms and styles. An intensional representation involves the use of variables and operators e.g. P1 = a => P2= b, which states that when variable P1 has value of a in a configuration, then variable P2 must have a value of b.

Extensional representation involves encoding constraints as forbidden/unwanted tuples, which define combinations of parameter- value that cannot execute successfully or not required in the context of the given system under test. For a system with parameters P1 = [0, 1], P2 = [2, 3] and P3= [4, 5], extensionally (0, 2), (1, 4) may be specified as constraints in the system. This representation may give rise to implicit relationships among option value choices, which may lead to multiple forbidden tuples. In the example, (2, 4) is implicit and needs to be removed as well. Depending on the nature of the constraints, a large number of tuples may need to be specified and these may be problematic for some systems. We allow the specification of constraints at a higher level of abstraction according to a constraint grammar.

3.2.1 Constraint Grammar

Figure 2 shows the BNF for our constraint grammar.



```
              Constraint ::= ( LogicalExpression )
       LogicalExpression ::= ( ImplicativeExpression )
   ImplicativeExpression ::= ( OrExpression ) ( <IM> ( OrExpression ) )*
            OrExpression ::= ( AndExpression ) ( <OR> ( AndExpression ) )*
           AndExpression ::= ( RelationalExpression ) ( <AND> (
                             RelationalExpression ) )*
    RelationalExpression ::= ( AdditiveExpression ) ( ( <LT> ( AdditiveExpression )
                             ) | ( <GT> ( AdditiveExpression ) ) | ( <LE> (
                             AdditiveExpression ) ) | ( <GE> ( AdditiveExpression )
                             ) | ( <EQ> ( AdditiveExpression ) ) | ( <NE> (
                             AdditiveExpression ) ) )*
      AdditiveExpression ::= ( MultiplicativeExpression ) ( ( <PLUS> (
                             MultiplicativeExpression ) ) | ( <MINUS> (
                             MultiplicativeExpression ) ) )*
MultiplicativeExpression ::= ( UnaryExpression ) ( ( <MUL> ( UnaryExpression ) ) |
                             ( <DIV> ( UnaryExpression ) ) | ( <MOD> (
                             UnaryExpression ) ) )*
         UnaryExpression ::= ( ( <INTEGER_LITERAL> ) | (
                             <STRING_LITERAL> ) | ( <TRUE> ) | ( <FALSE> ) |
                             ( <IDENTIFIER> ) | <NOT> ( UnaryExpression ) | "("
                             ( LogicalExpression ) ")" )
```

Figure 2: BNF for Constraint Grammar

The following represents valid constraints according to the grammar:

- Numeric Parameter > number , Numeric Parameter = number, String Parameter = “string literal” , Boolean Parameter = true, Boolean Parameter != false
- Numeric Parameter1 > Numeric Parameter2, Boolean Parameter1 != Boolean Parameter2
- Numeric Parameter > number => String Parameter = “string literal”
- String Parameter = “string literal” && Boolean Parameter = true => Numeric Parameter <= number
- Numeric Parameter1 * Numeric Parameter2 = number => String Parameter = “literal”

A parser generated for the grammar is used to parse the constraint texts and during test generation, values are supplied for each parameter in the constraint and evaluated with a Boolean result.

3.3 Constraint Satisfaction Problem

We represent and solve a constraint problem as a CSP, which consist of:

- A set of variables X = {x1, x2, …,xn}
- For each xi, a finite set Di of possible values (its Domain)
- A set of constraints restricting the values that can be simultaneously assigned to the variables.

A solution to a CSP is an assignment of a value from its domain to every variable, in such a manner that every constraint is satisfied. Each constraint in a CSP may involve one or more parameters, where the number of parameters in the constraint is known as its arity. A constraint involving two parameters is known as a binary constraint, while a constraint involving one parameter is known as a Unary constraint and a higher order or non-binary constraint involves more that two parameters. A CSP with all binary or unary constraint is represented in a constraint graph.

3.4 Constraint Graph

A constraint graph consists of binary or unary constraints in which each node represents a variable, and each arc represents a constraint between variables represented by the end points of the arc. A unary constraint is represented by an arc from and to the same node. Consider the following constraints:

- OS != “Sun Solaris 8”
- OS = “Mac OS X” => Application Server != “Weblogic 8.1”
- Hardware = “Sparc” => Application Server != “Weblogic 8.1”
- Hardware = “Mac” => OS = “Mac OS X”

Figure 3 shows the resulting constraint graph for these constraints.

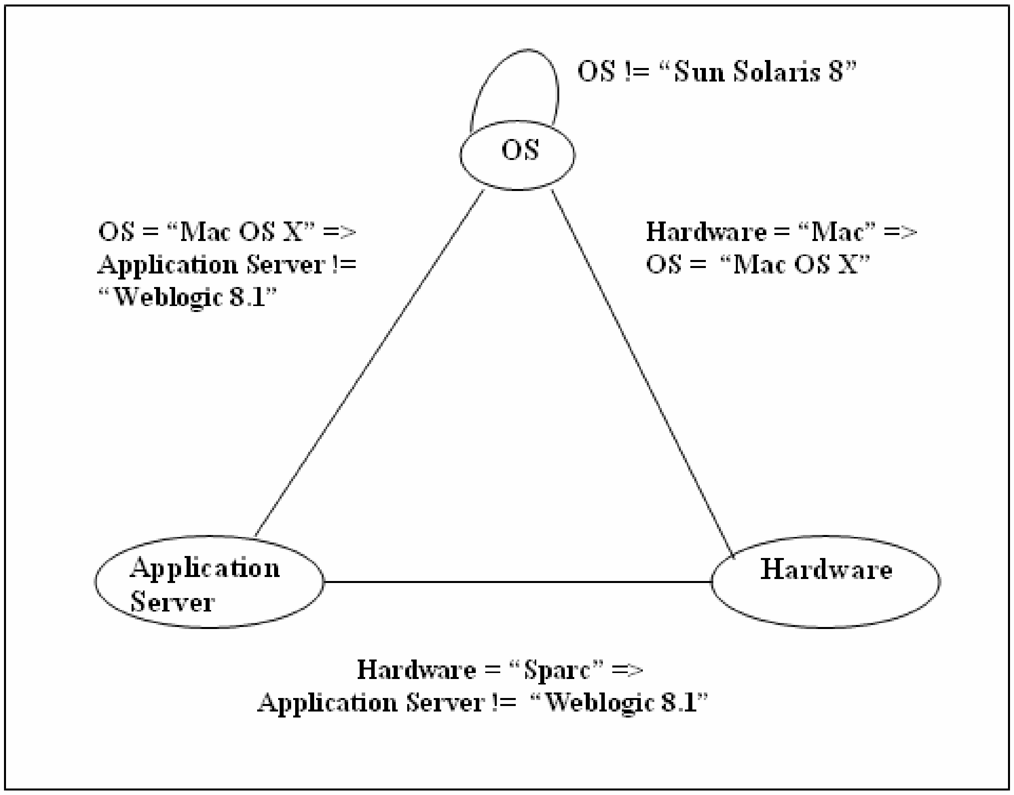

OS != "Sun Solaris 8"
OS
OS = "Mac OS X" => Application Server != "Weblogic 8.1"
Hardware = "Mac" => OS = "Mac OS X"
Application Server
Hardware
Hardware = "Sparc" => Application Server != "Weblogic 8.1"

Figure 3: Constraint graph for unary and binary constraints

3.5 Non-Binary Constraints

A Non-Binary constraint is a constraint that involves more than two variables (arity >2). We solve non-binary constraints by converting each non-binary constraint to an equivalent binary CSP, using the well known Hidden Variable Encoding [9]. This method encodes each non-binary constraint to a variable (called "hidden" variable) that has as domain the valid tuples of the constraint. For each tuple in the domain of the hidden variable Hv, the encoding introduces compatibility constraints between Hv and each original variable xi in the constraint c.

Consider the tenary constraint: [(Application Server = "Weblogic 8.1") && (Java SDK = "Oracle JRockit 1.4") => (Browser = "Firefox 2.0")] defined for the system in figure 1, to convert this constraint to its binary equivalent, we introduce a hidden variable Hv to the constraint , where the domain of Hv is {(Weblogic 8.1, Oracle JRockit 1.4, Firefox 2.0) } . These values represent the combination values for (Application Server, Java SDK, Browser), that satisfy the non-binary constraint. A binary compatibility constraint between Application Server and Hv, Java SDK and Hv and between Browser and Hv now exists as a result of the introduction of the hidden variable and is shown in figure 4

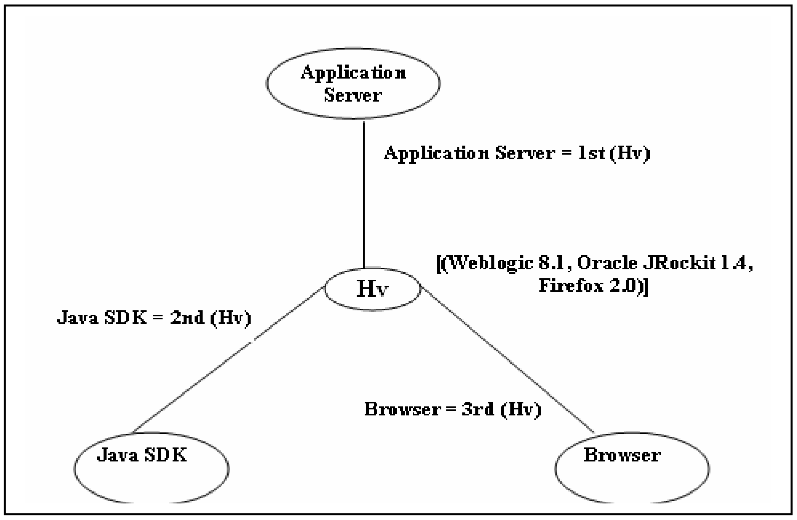

Application Server
Application Server = 1st (Hv)
[(Weblogic 8.1, Oracle JRockit 1.4, Firefox 2.0)]
Hv
Java SDK = 2nd (Hv)
Browser = 3rd (Hv)
Java SDK
Browser

Figure 4: Hidden variable encoding of a non-binary CSP

As part of our algorithmic extension to handle constraints, we implement a hidden variable translation algorithm, which converts every non-binary constraint in a CSP to its binary constraint equivalent.

3.6 Future Conflicts

The problem of future conflict arises when a selected value satisfies some constraints at one point in the test generation process but fails to satisfy every constraint in the future. This problem may lead to backtracking and removal of combinations earlier marked as covered and hence may affect test coverage. Consider a CSP with three parameters: P1[0,1], P2[1,3] , P3[1,2,3] and two constraints: P1 < P2 and P2 < P3 and the combination of values for P1,P2 :

(0,3) and P2, P3 : (3, 3). A test generation algorithm executes the following steps to cover the combinations:

- Assign 0 to P1 and 3 to P2
- Check the constraint between P1 and P2
- Add combination to test suite since P1 < P2 is true.
- Assign 3 to P3
- Check constraint between P2 and P3
- P2 is not less than P3, backtrack and remove 0,3 since no value of P3 can satisfy the constraint between P2(3) and P3

This is clearly a problem and we detect and prevent this type of problem by using Forward Checking.

3.7 Forward Checking

Forward checking is a constraint propagation technique that is used to propagate the implications of constraints on one variable to onto other variables in the constraint graph. The Forward checking technique performs look ahead to detect impossible combinations as soon as possible, thereby preventing future conflicts. During forward checking whenever a variable X is assigned a value a, the algorithm checks every unassigned value of variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X. Any current assignment of value a to x that results in a "domain wipe-out" of any future variable, is inconsistent and not part of a solution to the CSP.

3.7.1 Preventing future conflict with forward checking

Forward checking can be applied to the problem in section 3.6 as follows:

- Assign 0 to P1
- Check arc P1, P2 (every value of P2 is consistent with 0)
- Assign 3 to P2
- Check arc P2, P3

- Delete 1, 2, 3 from domain of P3 since none is less than 3 . This reduces the domain of P3.
- Empty domain of P3 implies 0,3 is inconsistent.

CHAPTER 4

ALGORITHMIC SUPPORT FOR CONSTRAINT HANDLING

4.1 Overview

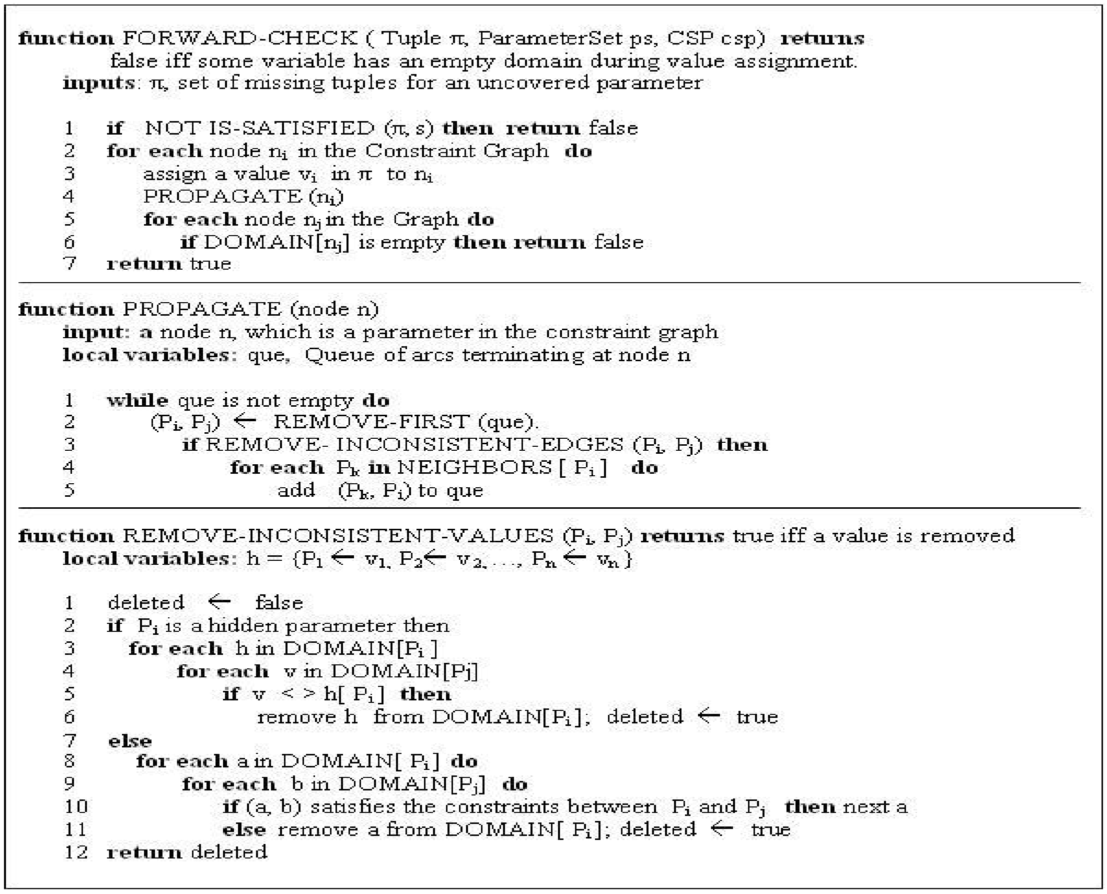
In this section, we present the FireEye constraint handling strategy, Our motivation for providing this extension is to develop a general constraint handling strategy that is tightly integrated into an efficient test generation framework, easily re-implementable, scales well for a large number of parameter-values and frees the software tester from the trouble of manipulating constraints by listing all invalid combinations or re-modeling the system. We also show how to integrate our solution with the IPOG test generation algorithm.

4.2 The FC-CS Algorithm

Figure 5 shows the FC-CS algorithm, which implements forward checking and maintaining arc consistency strategy for constraint handling. The algorithm follows from the AC-3 algorithm for arc consistency but also considers hidden variables during search and arc consistency. After applying algorithm FC-CS, either every arc is arc consistent or some parameter has an empty domain, indicating that the CSP cannot be made arc consistent. In the case of a domain wipe-out, the current assignment is regarded as invalid and therefore rejected. Algorithm FC-CS consists of three functions, FORWARD-CHECK, PROPAGATE and REMOVE-INCONSISTENT-VALUES.

FORWARD-CHECK takes as input a tuple π, consisting of the combination of values to be checked against the set of constraints in the CSP, a set of input parameters ps and a CSP cs and returns false if some variable in the graph experiences a domain wipe-out during value assignment. PROPAGATE takes as input a node with a current assignment v in π and after execution every arc in the constraint graph is either arc-consistent or some node has an empty domain. For each arc (Pi, Pj), if a value is deleted from domain of Pj then PROPAGATE revises those arcs that terminate at Pj.

REMOVE-INCONSISTENT-VALUES takes as input an arc (Pi, Pk) and returns true if a value is removed from the domain of Pi. A value is removed from the domain of a variable if the algorithm detects that the variable does not support the current assignment of value to Pi.



```
function FORWARD-CHECK ( Tuple $\pi$, ParameterSet ps, CSP csp) returns
        false iff some variable has an empty domain during value assignment.
    inputs: $\pi$, set of missing tuples for an uncovered parameter

    if  NOT IS-SATISFIED ($\pi$, s) then return false
    for each node $n_i$ in the Constraint Graph do
        assign a value $v_i$ in $\pi$ to $n_i$
        PROPAGATE ($n_i$)
        for each node $n_j$ in the Graph do
            if DOMAIN[$n_j$] is empty then return false
    return true
```

```
function PROPAGATE (node n)
    input: a node n, which is a parameter in the constraint graph
    local variables: que, Queue of arcs terminating at node n

    while que is not empty do
        ($P_i$, $P_j$) $\leftarrow$ REMOVE-FIRST (que).
            if REMOVE- INCONSISTENT-EDGES ($P_i$, $P_j$) then
                for each $P_k$ in NEIGHBORS [ $P_i$ ]  do
                    add ($P_k$, $P_i$) to que
```

```
function REMOVE-INCONSISTENT-VALUES ($P_i$, $P_j$) returns true iff a value is removed
    local variables: h = {$P_1 \leftarrow v_1$, $P_2 \leftarrow v_2$, ..., $P_n \leftarrow v_n$}

    deleted $\leftarrow$ false
    if $P_i$ is a hidden parameter then
      for each h in DOMAIN[$P_i$]
          for each v in DOMAIN[Pj]
              if v < > h[ $P_i$ ] then
                  remove h from DOMAIN[$P_i$]; deleted $\leftarrow$ true
    else
       for each a in DOMAIN[ $P_i$] do
           for each b in DOMAIN[$P_j$] do
               if (a, b) satisfies the constraints between $P_i$ and $P_j$ then next a
               else remove a from DOMAIN[ $P_i$]; deleted $\leftarrow$ true
    return deleted
```

Figure 5: FC-CS Algorithm for Constraint Handling

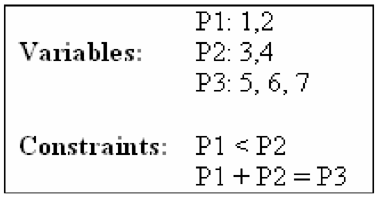
4.2.1 *The FC-CS Strategy*

The FC-CS strategy is as follows:

- Assign each value v to a node Xj in the constraint graph
- Invoke propagate to propagate the effect of the current assignment
    - Insert every arc (Xi, Xj) terminating at Xj into a Queue

        - Remove each arc (Xi, Xj) from the Queue
        - Look at each unassigned variable Xi connected to Xj by a constraint
        - Delete any value in the domain of Xi that is inconsistent with Xj
            - After deleting a value from the domain of Xi, every arc (Xk, Xi) pointing at Xi is re-inserted into the Queue and re-checked.
    - If there is any node Xj with empty domain, reject the current assignment.

To describe algorithm FC-CS, we define a sample CSP and apply the algorithm to the CSP. Figure 6 shows our sample CSP, which is our input to algorithm FC-CS.



Variables:	P1: 1,2
	P2: 3,4
	P3: 5, 6, 7
Constraints:	$P1 < P2$
	$P1 + P2 = P3$

Figure 6: Sample CSP input to FC-CS

A greedy algorithm like IPOG will generate its test one- row at a time, selecting values from each parameter to generate a t-way test set. During test generation, FC-CS will be invoked to determine if a tuple satisfies the constraints in the CSP. Consider the input π = (1, 3, 6), ps = {P1, P2, P3} and the CSP in figure 6. The initialization phase of FC-CS involves:

- Sorting input parameters according to their domain sizes
- Converting every non-binary constraint in the CSP into binary equivalent.
- The initialization of the constraint graph, with all parameters in the input model added to the graph and edges added between nodes that have binary constraints between them.

In FORWARD-CHECK the value 1 from π is assigned to node P1 and PROPAGATE is invoked. Figure 7 shows a snapshot of the constraint graph with 1 assigned to P1.

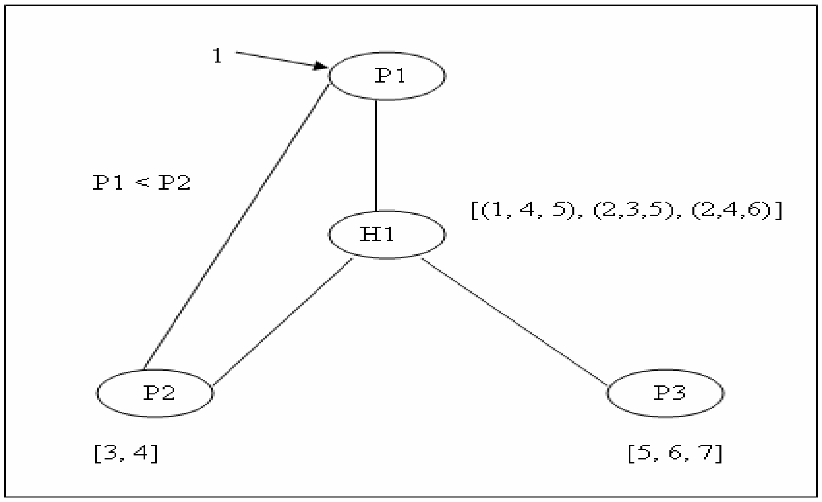


Figure 7: Constraint Graph showing assignment P1 ← 1

In PROPAGATE, every arc terminating at P1 is inserted into the Queue [(P2, P1), (H1, P1)] and each arc is revised by invoking REMOVE-INCONSISTENT-VALUES on the arc. REMOVE-INCONSISTENT-VALUES checks arc P2, P1, and finds every value of P2 consistent with the current value of P1 (1). Next the arc (H1, P1) is checked and the tuples (2, 3, 5), (2, 4, 6) are temporarily deleted from the domain of H1 since the value of P1 is not 1 in those tuples. This deletion causes the arcs (P1, H1), (P2, H1), (P3, H1) to also be inserted into the Queue and re-checked since they may have lost support as a result of the deletion. Figure 8 shows the result of invoking PROPAGATE on node P1 ← 1.

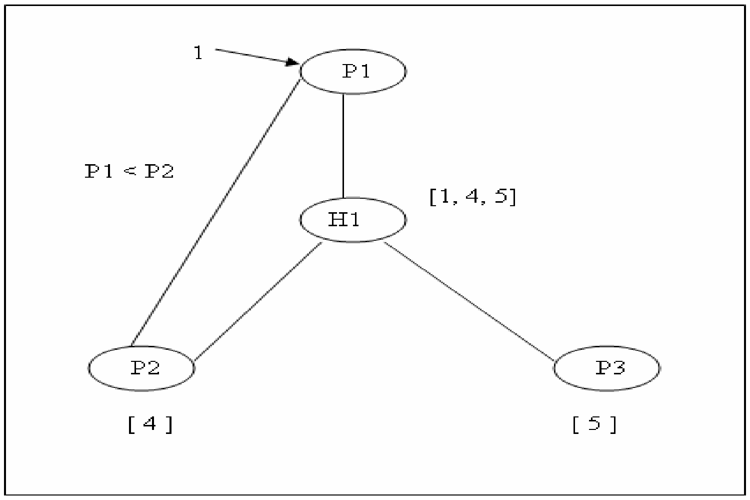


Figure 8: Constraint Graph showing P1 ← 1 and Domain [H1] reduced

Next the algorithm assigns the value 3 to P2 and the effect is propagated. Arcs (P2, P1) and (P2, H1) are added to the queue in PROPAGATE. When REMOVE-INCONSISTENT-EDGES is invoked with (P2, H1), the remaining tuple (1, 4, 5) is removed from domain Hi since the assignment of P2 ← 3 is not supported by H1. The domain Hi experiences a domain wipe-out and the function PROPAGATE returns to FORWARD-CHECK, which returns true to the test generation algorithm, indicating an invalid combination (1, 3, 7).

Figure 9 shows the assignment of 3 to P2 which results in a domain wipe-out for Hi

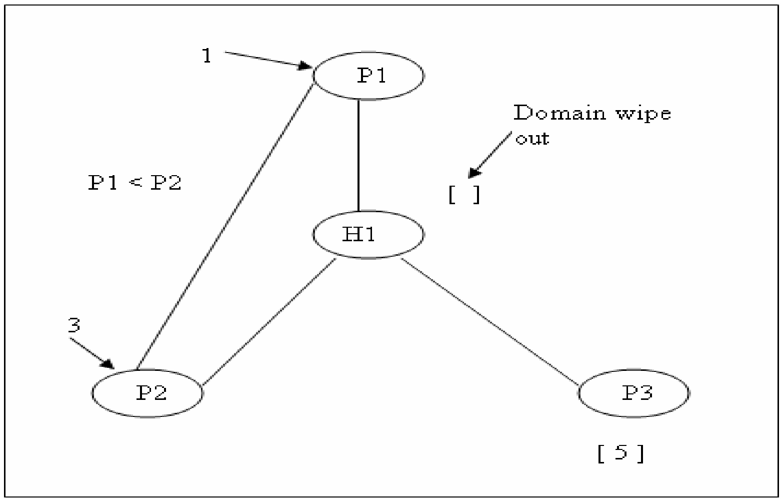


Figure 9: Constraint Graph showing domain wipe out after assigning P2 ← 3

4.3 Integrating FC-CS with IPOG

The algorithmic extension to IPOG happens at four major points and these points of extensions are described below.

- IPOG adds into ts a test for every combination of the first t parameters: This step is modified to include a check for every test against the constraints in the CSP for satisfaction. Any test found to violate any constraint in the CSP is not added to ts. Figure 10 shows an algorithm that is used to check if a tuple satisfies every constraint in the CSP.

```
function IS-SATISFIED (tuple t, CSP cs) returns true if t satisfies every
        constraint in cs
        inputs: t, value combination for {$P_1, P_2, P_3 \dots P_n$}. cs, a binary CSP
              with variables {$P_1, P_2 \dots P_n, H_1, H_2, \dots, H_n$}
        local variables: h = {$P_1 \leftarrow v_1, P_2 \leftarrow v_2, \dots, P_n \leftarrow v_n$}; support = false;

1    for each $cs_i$ in cs do
2          if no of variables in $cs_i$ > 2 then
3              support $\leftarrow$ true
4              for each h in DOMAIN[$H_i$] do
5                  for each v $\rightarrow$ Pi in t
6                      if v <> h[Pi] then support $\leftarrow$ false
7              if not support then return false
8          else
9              if values $v_1, v_2, \dots v_n$ in t violates the constraint $cs_i$ between
                      $P_1, P_2, \dots, P_n$
10                 then return false
11   return true
```

Figure 10: Algorithm IS-SATISFIED

- IPOG generates a t-way combinations of values, π involving parameter P_i and t-1 parameters among the first i – 1 parameters. We extend IPOG at this point to include a check that every tuple in π satisfies the constraints in CSP. Any tuple that does not satisfy the constraints in CSP will be removed from π. This check involves invoking the MAC-CS algorithm each time we pick up a tuple in π .
- IPOG extends the test set ts by adding a value for the new parameter (Horizontal growth). When a test is selected for extension, we invoke the FORWARD-CHECK algorithm to check that the test satisfies every constraint in the CSP.
- IPOG extends the test set ts by adding a new test or changing an existing test (Vertical Growth). In both cases, we invoke FORWARD-CHECK to validate the test and ensure the test satisfies every constraint in the CSP.

4.4 Don't Care Values

A don't care value represents a value of a parameter that does not contribute to the test coverage of a test suite during test generation. A don't care value may be represented by a character e.g. "*" to denote that any value of the parameter can be used to replace "*".

4.4.1 Handling Don't Care Values in Constrained inputs

In the presence of constraints, a value is regarded as don't care, if and only if it does not contribute to the test coverage and does not violate any constraints in the input model. To enforce this condition, every value of the parameter needs to be checked against the set of constraints and if any violation is found, the value is not a don't care value. In the integration of FC-CS with IPOG, we check every don't care value to ensure that it satisfies every constraint,

- If we find that no value of the parameter satisfies all the constraints, we remove the test case.
- If every value of the parameter satisfies every constraint, then we leave it as don't care.
- If any value satisfies every constraint, we replace the don't care with the satisfying value.

4.5 FC-CS Complexity

We consider next, the complexity of the constraint handling strategy. For a binary CSP, space complexity is O(md), where m is the number of constraints and d is the maximum domain size. Worst case time complexity for binary CSP is $O(n^2d^3)$, where n is the number of variables. A binary CSP has a maximum of $O(n^2)$ arcs and checking consistency of an arc (X_i, X_j) requires $O(d^2)$ time. Each arc (X_k, X_i) can be inserted into the queue, 2d times. Total time complexity is $O(n^2 d^2 2d) = O(n^2d^3)$.

For a non-binary CSP, space complexity is dominated by the space required to store domain of the hidden variable and this is $O(ed^k)$, where e is the number of hidden variables and k is the maximum arity of the constraints. Worst case time complexity for non-binary CSP is given as $O(ekd^{k+1})$. This time complexity is dominated by the call to REMOVE-INCONSISTENT-

VALUES. For every deletion of a value from a domain of one of the k original variables constrained with a hidden variable Hi, REMOVE-INCONSISTENT-VALUES will be invoked once. This means REMOVE-INCONSISTENT- VALUES can be called at most kd times. For each value a, we perform O(d^k) checks to find out if a tuple in DOMAIN[Hi] supports value a, and for kd values, we have O(kdd^k) = O(kd^{k+1}). In general, for e hidden variables, worst case time complexity is O(ekd^{k+1}).

CHAPTER 5

FIREEYE: A T-WAY TESTING TOOL WITH CONSTRAINT SUPPORT

5.1 Overview

FireEye is a combinatorial testing tool that can be used to generate t-way test suites. The test suites generated by FireEye are more effective when compared to manually generated tests, in detecting faults that are caused by interactions among different participating parameters.

Today's complex, large and heavily distributed systems increase testing requirements and Tester's generally require tools that help to automate the process of testing, FireEye with its robust design and rich set of functionality help provide the automation required by testers and also helps to meet the increased testing requirements of today's systems.

In this chapter we describe the design and major features of FireEye.

5.2 FireEye Architecture and Design

5.2.1 Architectural Overview

The overall design goal of FireEye is ease of use, extensibility of the core engine, minimal configuration and platform independence. Figure 11 shows a high level system architecture diagram for FireEye.

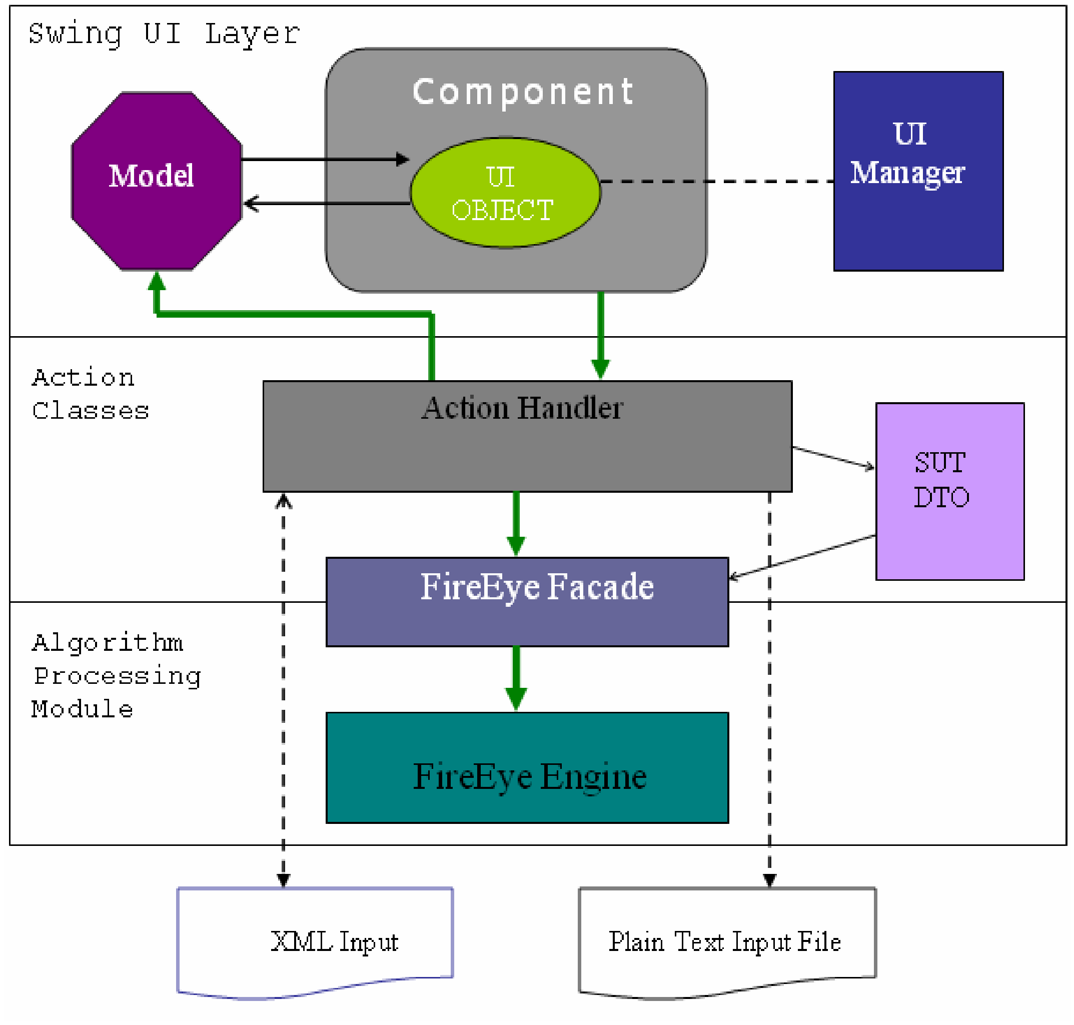


Figure 11: FireEye System High Level Architecture Overview

5.2.2 General Workflow

The workflow that is generally executed is as follows

1. User launches the main UI component.
2. User creates or edits a system.
3. User initiates a build event.

4. Build Action handler invokes build method exposed by the FireEye facade, passing the system configuration object.
5. Build Action handler receives the build result and updates the swing component model.
6. The controller component of the view updates the view with test results.

5.2.3 FireEye GUI Design

The core design principle for the FireEye GUI is ease of use. We designed the GUI components to make the job of the software tester easy and enjoyable. We present below a use case diagram (Figure 12) for the FireEye GUI.

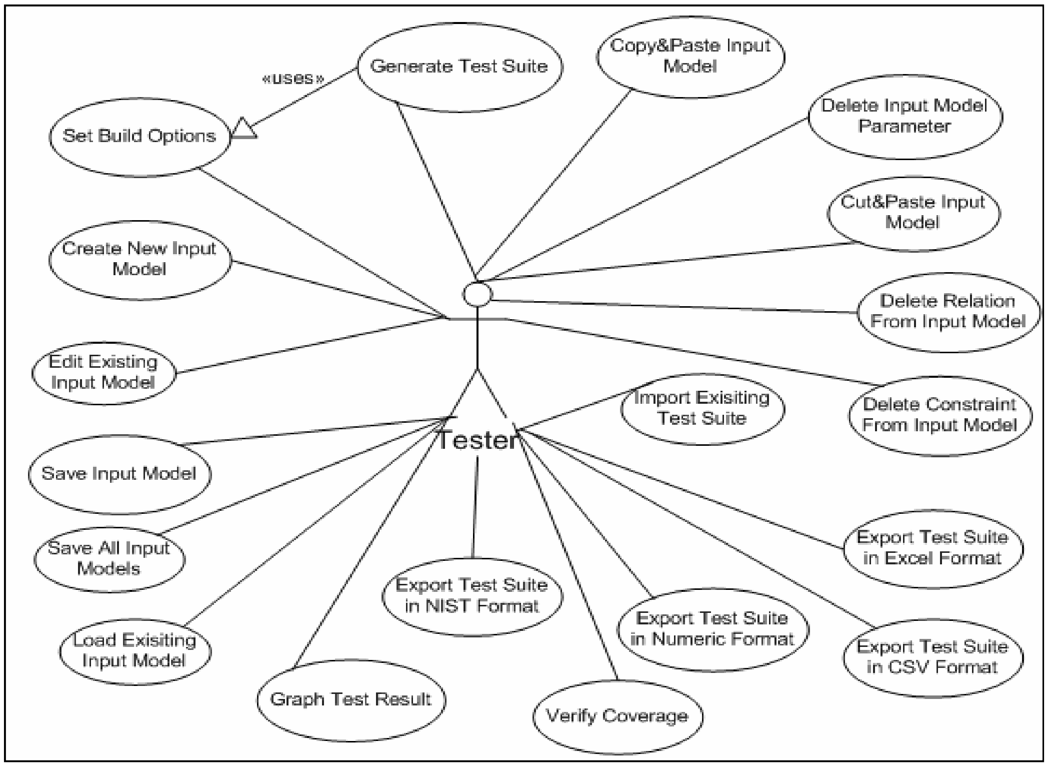

Generate Test Suite
Copy&Paste Input Model
«uses»
Set Build Options
Delete Input Model Parameter
Cut&Paste Input Model
Create New Input Model
Delete Relation From Input Model
Edit Existing Input Model
Import Exisiting Test Suite
Delete Constraint From Input Model
Tester
Save Input Model
Save All Input Models
Export Test Suite in Excel Format
Export Test Suite in NIST Format
Export Test Suite in Numeric Format
Export Test Suite in CSV Format
Load Exisiting Input Model
Graph Test Result
Verify Coverage

Figure 12: FireEye GUI use case diagram

5.3 Technology Stack

The technologies, tools and frameworks used in developing FireEye are as follows:

5.3.1 Graphical User Interface

The following technologies and frameworks were used to implement FireEye user interface.

- Java Swing: This is a GUI framework that implements a modified model-view-controller, sometimes referred to as separable model architecture. The swing architecture consists of a model, which represents the data for the application, a user-interface object, which consist of the controller and the view, and a user interface manager, which is used by components and programs to access look-and-feel information.
- Swing layout: This is an extension library to swing, which enables the creation of professional cross platform layouts.

5.3.2 Utility libraries

- JDOM: A Java-based open-source library for accessing, manipulating and outputting XML data from Java code.
- JXL: An open source java library for reading, writing and modifying excel spreadsheets dynamically.
- JFreeChart: An open source java chart library for displaying quality charts in client applications.
- JH: A java help library used to develop help viewer components for application help system.
- XercesImpl: An open source implementation of Xerces, which is a Java XML parser.

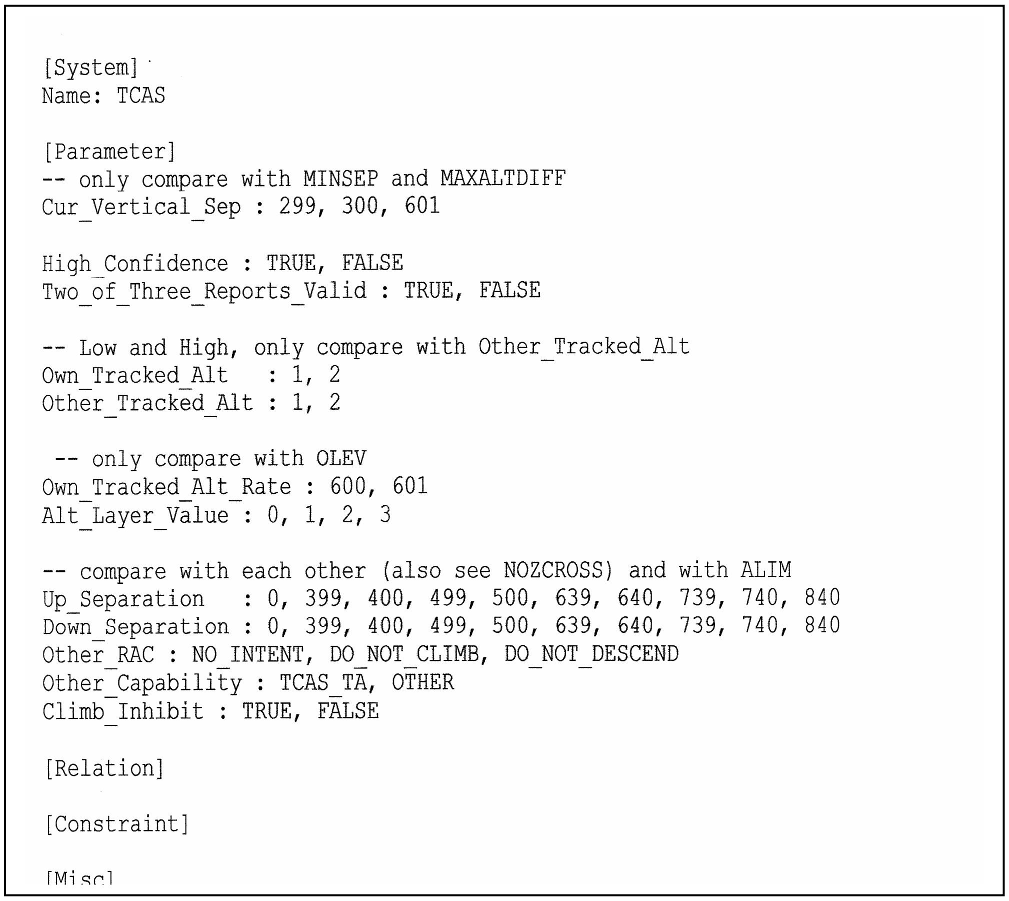
JCommon: A collection of utility classes used by JFreechart.

5.4 FireEye Major features

In this section we provide an overview of the major features of FireEye.

5.4.1 Input Model

We mentioned earlier that the major design goal of FireEye is ease of use, this is true as evidenced by our inputs to FireEye. Two input file formats are accepted by FireEye, a plain text file (Figure 13) and an XML input file (Figure 14). The plain text file can be created outside the application and may be converted to the GUI input format by opening the text file from the open window command and then saving the system in the XML format.



```
[System]
Name: TCAS

[Parameter]
-- only compare with MINSEP and MAXALTDIFF
Cur_Vertical_Sep : 299, 300, 601

High_Confidence : TRUE, FALSE
Two_of_Three_Reports_Valid : TRUE, FALSE

-- Low and High, only compare with Other_Tracked_Alt
Own_Tracked_Alt   : 1, 2
Other_Tracked_Alt : 1, 2

 -- only compare with OLEV
Own_Tracked_Alt_Rate : 600, 601
Alt_Layer_Value : 0, 1, 2, 3

-- compare with each other (also see NOZCROSS) and with ALIM
Up_Separation   : 0, 399, 400, 499, 500, 639, 640, 739, 740, 840
Down_Separation : 0, 399, 400, 499, 500, 639, 640, 739, 740, 840
Other_RAC : NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND
Other_Capability : TCAS_TA, OTHER
Climb_Inhibit : TRUE, FALSE

[Relation]

[Constraint]

[Misc]
```

Figure 13: Plain Text Input Model File for FireEye

A tester specifies test parameters and values, parameter groups and constraints in the input file. By default, the tool generates a pairwise test set and also provides an interface through which the tester can specify a different value for it.

```
<?xml version="1.0" encoding="UTF-8"?>
<System name="TCAS">
  <Parameters>
    <Parameter id="0" name="Cur_Vertical_Sep" type="0">
      <values>
        <value>299</value>
        <value>300</value>
        <value>601</value>
      </values>
    </Parameter>
    <Parameter id="1" name="High_Confidence" type="2">
      <values>
        <value>true</value>
        <value>false</value>
      </values>
    </Parameter>
    <Parameter id="2" name="Two_of_Three_Reports_Valid" type="2">
      <values>
        <value>true</value>
        <value>false</value>
      </values>
    </Parameter>
  </Parameters>
  <Relations>
    <Relation Strength="2" Default="true">
      <Parameter name="Cur_Vertical_Sep">
        <value>299</value>
        <value>300</value>
        <value>601</value>
      </Parameter>
      <Parameter name="High_Confidence">
        <value>true</value>
        <value>false</value>
      </Parameter>
      <Parameter name="Two_of_Three_Reports_Valid">
        <value>true</value>
        <value>false</value>
      </Parameter>
    </Relation>
  </Relations>
  <Constraints>
    <Constraint text="Cur_Vertical_Sep&gt;300=&gt;High_Confidence=
    Two_of_Three_Reports_Valid">
      <Parameters>
        <Parameter name="Cur_Vertical_Sep" />
        <Parameter name="High_Confidence" />
        <Parameter name="Two_of_Three_Reports_Valid" />
      </Parameters>
    </Constraint>
  </Constraints>
</System>
```

Figure 14: XML input model file for FireEye

5.4.2 T-way Test Set Generation

This is the main feature of FireEye. When an input model is supplied, FireEye generates a t-way test set for the input, where t can be specified by the user or the default value of 2 is used. FireEye supports t-way generation for $2 \le t \le 6$ and allows the use of a variety of test set generation algorithms available from the GUI options window. The following is a list of supported algorithms developed by the ACTS group: IPOG, IPOG-D, IPOG-F, IPOG-F2 and Paintball. With FireEye, test set generation can proceed in two modes namely: scratch and extend. In scratch mode, a test set is built from the scratch, while in extend mode; a test set is generated by extending an existing test set.

5.4.3 Mixed Strength Test Generation

FireEye's mixed strength test generation feature allows different parameter groups to be created and covered with different strengths. A tester may discover that certain parameter interactions leads to a different set of defects or increases the number of detects than other interactions, hence the parameters should be tested thoroughly, setting a higher t on the entire set of parameters may produce too many test cases. The use of mixed-strength test generation will help to achieve a higher coverage without the problem of a large test suite. For example consider a system with 8 parameters, P_1, P_2…P_8. A default relation, consisting of all the parameters and strength of 2 can be created, then additional relations can be created, if the tester finds that some parameters require better coverage, he may for instance create a relation that includes P_2, P_3, P_4 and P_5 with strength 4 if P_2, P_3, P_4 and P_5 could potentially interact with each other and their interaction may trigger certain defects.

5.4.4 Constraint Support

FireEye includes support for constraint handling by generating test cases that are free of invalid or unwanted combinations of input parameters. We discussed the constraint handling of strategy of FireEye in chapter 3. A user interface for specifying constraints is provided in FireEye. The interface allows the specification of constraints involving different types (numeric, Enum and Boolean), any arity and with any of logical, relational, mathematic and equality operators. To implement our constraint handling strategy, we define a constraint grammar, which supports high level input of constraints. FireEye's constraint grammar supports identifiers, string literals, integer literals, digits, Booleans and operator tokens and a set of classes generated by JavaCC implements the lexical analyzer and parser for the grammar. The following texts represent constraints that are valid with respect to our constraint grammar:

- <Identifier><operator><String Literal>

  e.g Application Server != “Weblogic 8.1”
- <Identifier1> <operator> <Identifier2>

  e.g Loan_Amount < Household Income
- <Identifier1><operator1><digits><Implication>

  <Identifier2><operator2> <Boolean>

  e.g Loan_Amount > 500,000 => Approved = false

We provide an editor that enables the user to enter constraints similar to the ones above and these constraints are parsed according to the grammar before they are processed by the test generation engine during test generation.

5.4.5 Coverage Verification

Using this feature, a test set generated by FireEye or supplied by user can be verified for t-way coverage satisfaction. A 100% coverage is reported if a test set covers all t-way combinations.

5.5 FireEye GUI components

This section describes major GUI windows used to implement FireEye’s functionalities.

5.5.1 Main Window

Fig 15 shows FireEye’s main application window. This window contains menu items used to invoke other functional windows.

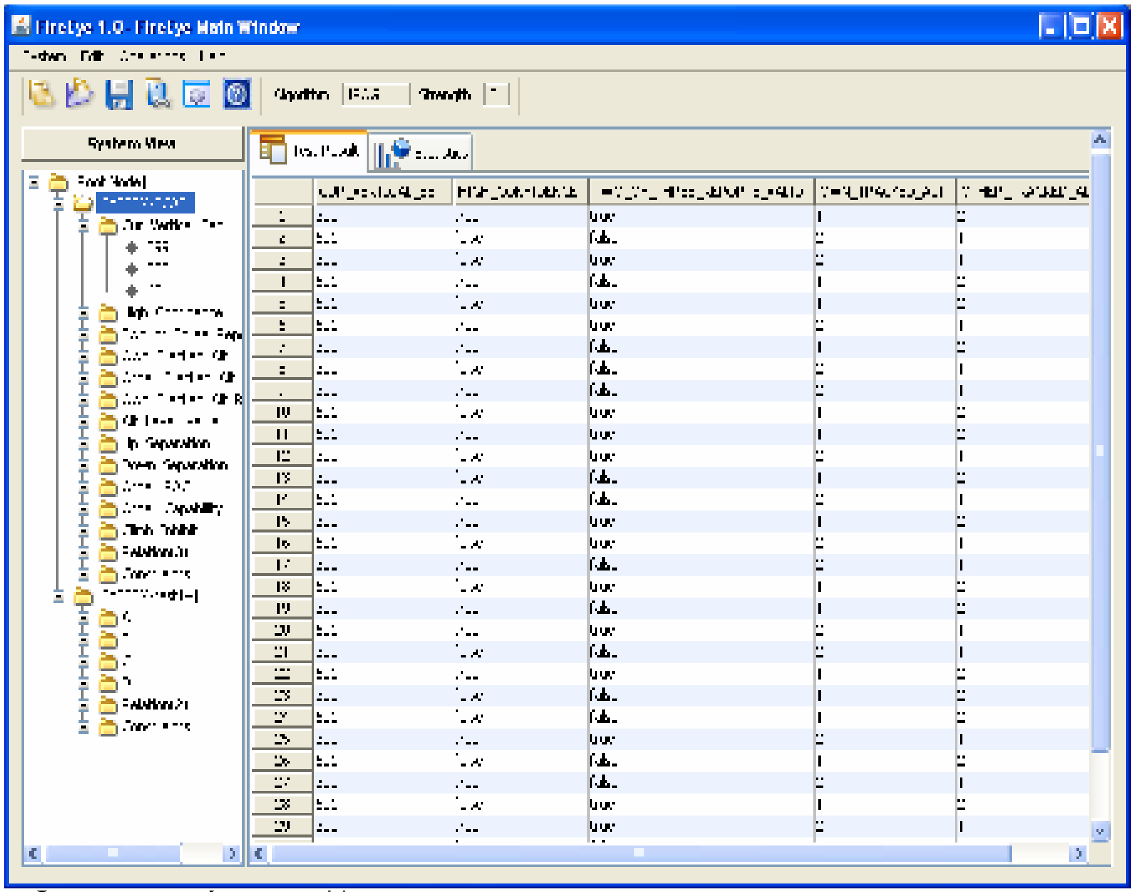
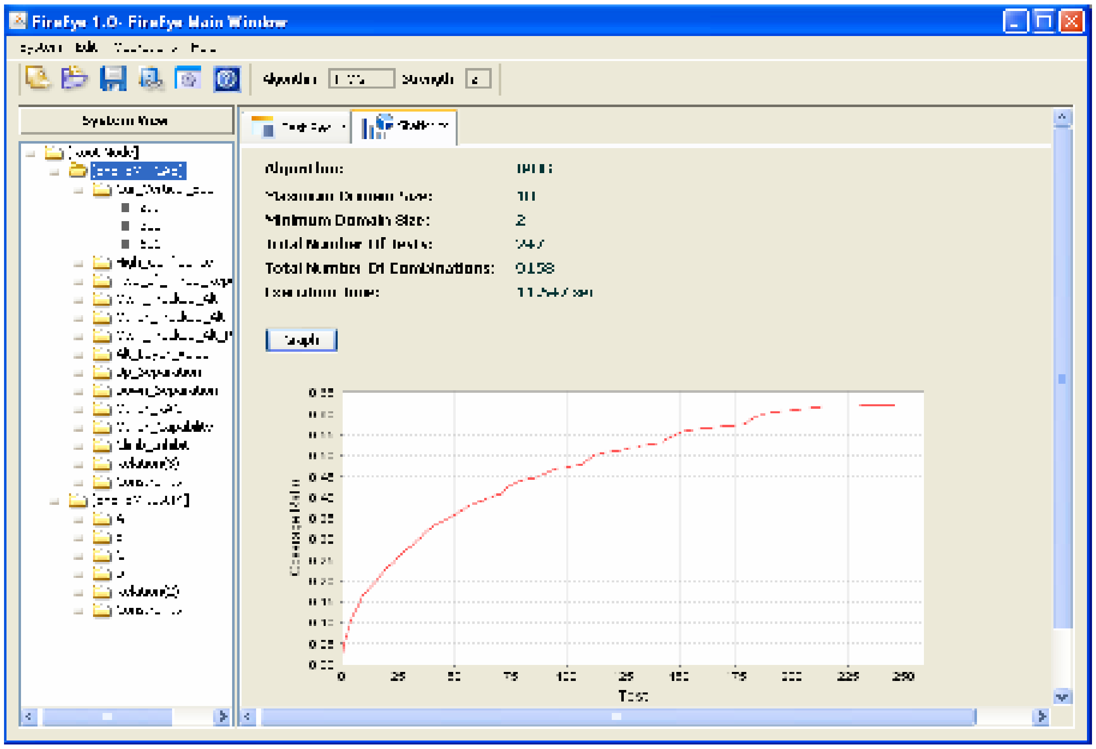


Figure 15: FireEye Main Window

The main window consists of a split pane, a menu bar and a toolbar. The left pane displays a tree structure that is used to represent system configurations. Multiple systems can be displayed in the tree and each system is shown as a three-level hierarchy namely: a top node level, which represents the system, a second node level, which represents parameters, relations or constraints and a third leaf level, which represents a set of values.

The right pane contains a tabbed pane, which consists of two tabs, namely: Test Result, shown in figure 15 and statistics shown in figure 16. The test result tab contains a table, which is used to display a test set for the currently selected system. Each row in the table represents a test and each column represents a parameter. The statistics tab displays relevant statistical information about the test set. It contains a graph button and a pane for displaying graphical results. The growth rate of the test coverage with respect to the tests in the test set displayed in the Test Result tab can be graphed by clicking on the graph below.



Algorithm:	IPOG
Maximum Domain Size:	10
Minimum Domain Size:	2
Total Number Of Tests:	247
Total Number Of Combinations:	9158
Execution Time:	11.547 sec

Figure 16: Main Window showing Statistics Tab

5.5.2 New System Window

Fig 17 shows the new system window used to create a new system configuration, edit an existing system configuration, add relations and add constraints. The new system window contains a tabbed pane made up of three tabs namely: Parameters, Relations and Constraints.

The Parameters tab allows the user to enter a system name, a list of parameters and a list of values for each parameter.

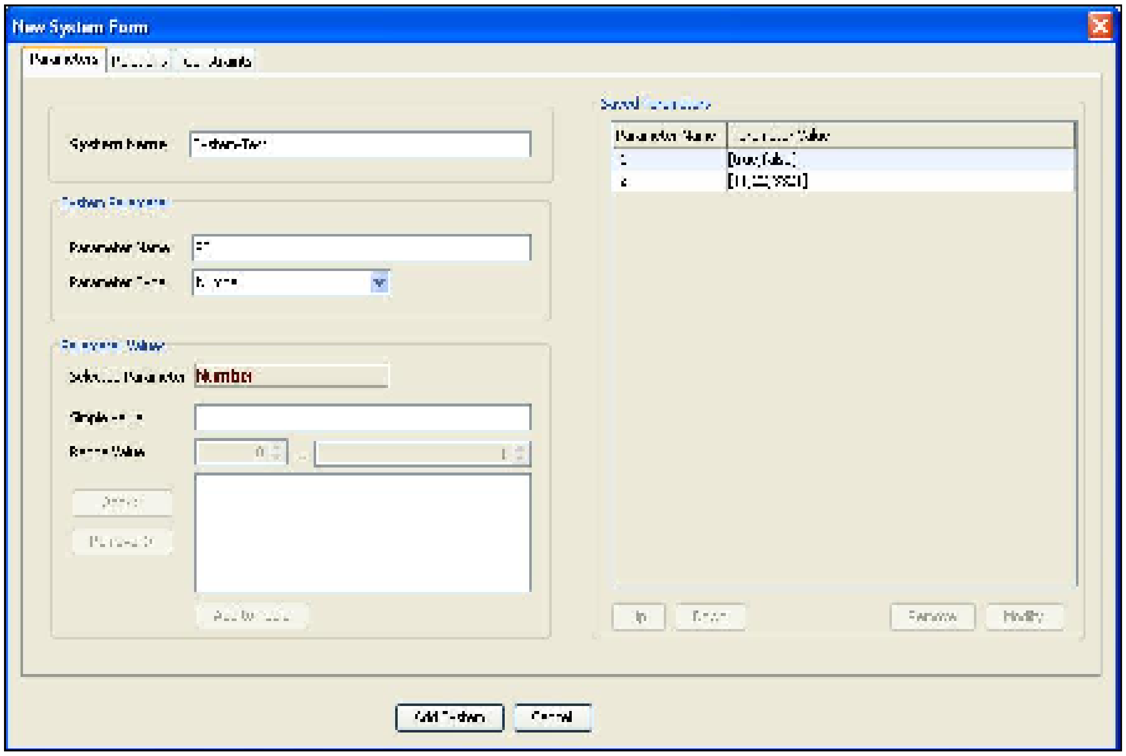


Figure 17: New System window

The Relations tab is shown in Figure 18, and it allows the user to create parameter groups with different strengths. The Constraints tab shown in Figure 19 allows the user to enter constraints as predicate functions. The tester can express constraints involving parameters and operators. The following operators are supported: logical, relational, equality and mathematical operators, and the tester can enter constraints involving any number of parameters.

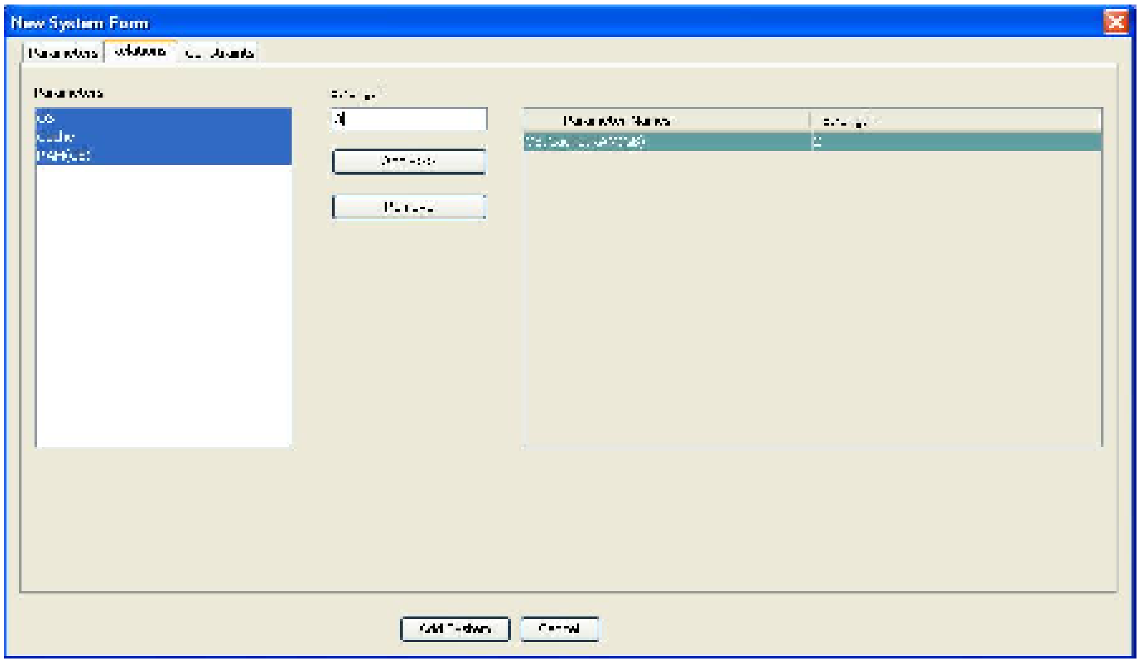


Figure 18: New System Window showing Relations Tab

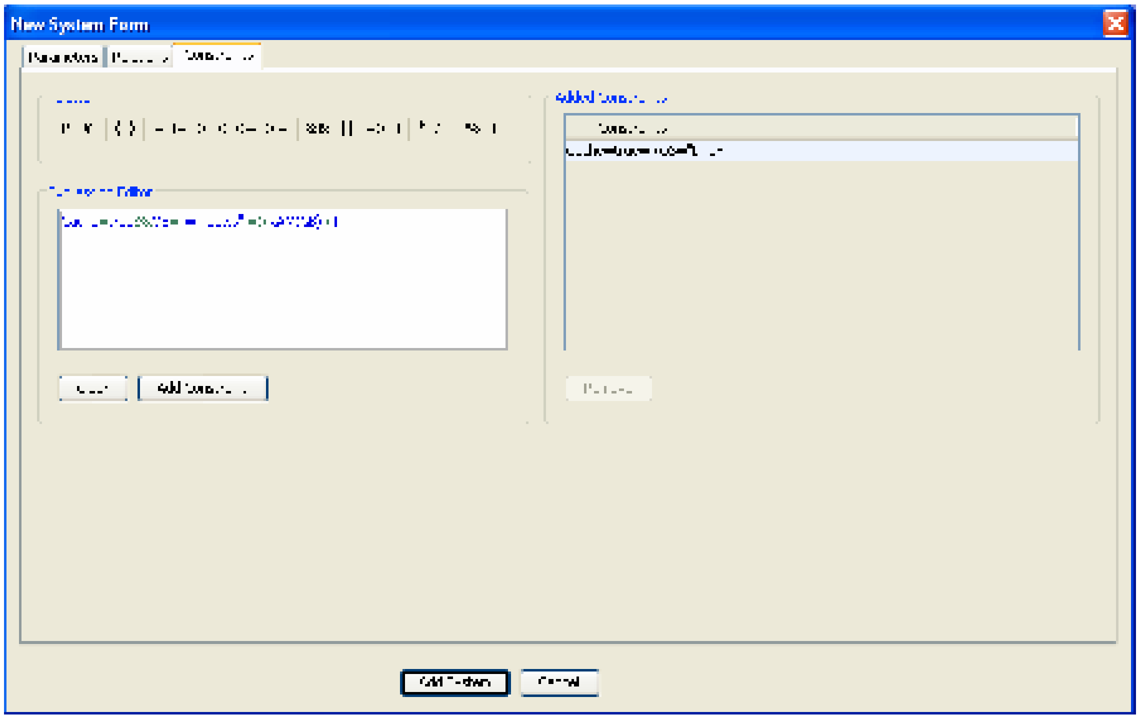
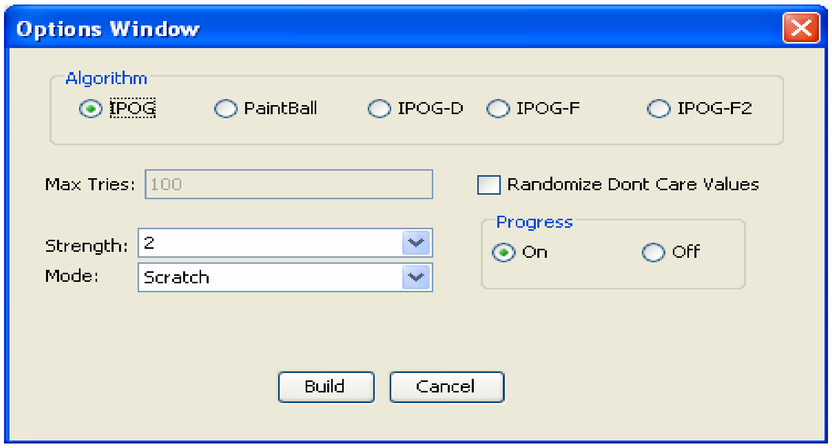


Figure 19: New System Window showing Constraint Tab

5.5.3 Build Options Window

The build options window is used to set or modify the build configuration for a current system. Figure 20 shows the build options window and the following option may be specified using this window

- Algorithm: This option allows the user to choose which algorithm to be used for test generation. The user can choose between IPOG, IPOG-F, IPOG-D, IPOG-F2 and Paintball.
- Maxtries: This option is relevant to the paintball algorithm, and specifies the number of candidates to be generated randomly at each step.
- Randomize don't care values: This option when checked, allows all don't care values in the resulting test set to be replaced with a random value.
- Mode: The user can select between scratch or extend, to indicate how the test set should be built. Scratch mode specifies that the test set be built from the scratch, while extend specifies that the test set should be built by extending an existing test set.
- eProgress: This option is used to enable the display of progress information on the console.



Algorithm: [✓ IPOG] PaintBall IPOG-D IPOG-F IPOG-F2

Max Tries: 100

Randomize Dont Care Values ☐

Strength: 2

Mode: Scratch

Progress: [✓ On] Off

Figure 20: Build Option Window

CHAPTER 6

EVALUATION AND RESULTS

6.1 Objectives

In this chapter we show some experimental results obtained while investigating the effect on test suite size, of running IPOG with FC-CS algorithm on a constrained input model and the effect on execution time, of running IPOG with FC-CS algorithm on a constrained input model. We ran IPOG with FC-CS on an Intel Pentium 1.73MHz processor with 2GB of RAM running Windows XP.

6.2 Impact of constraint handling on test suite size

To investigate the impact of IPOG with FC-CS on size of test suite, we generate random CSP, and compare our result with PICT and IPOG. Table 5 shows a subset of size data on constrained input models. A random CSP has n variables each with domain size d; and c number of constraints with arity k, and t satisfying tuples. We chose n between 3 and 6, c between 1 and 4, k between 1 and 4 and t between 2 and 3. For each n, we generate k values, where k_i represents the size of the domain [n_i].

For each subsystem configuration, we run IPOG-Test on the constrained input, we also run PICT on the constrained and unconstrained input model. We record for each run, 4 test suite sizes and the result is shown in table 5. Table 5 shows the following data sets:

1. The unconstrained input, model, SUT (n, d[ ]),d[ ], represents the array of domain sizes.
2. CSP parameters (c, k, t) added to the SUT to form the constrained input model. The result of running IPOG on the unconstrained input.
3. The result of running IPOG on the unconstrained input.
4. The result of running PICT on the unconstrained input

5. The result of running IPOG with FC-CS on the constrained input.
6. The result of running PICT on the constrained input.

The overall result show a slight variation between the constrained and unconstrained input when t = 2 but when t = 3, we notice more variation. We can conclude from the results, that our implementation of IPOG with FC-CS constraint handling strategy produces test suite sizes, which compares favorably with known existing test tool that supports constraints.

Table 5: Size data comparison of existing tool with IPOG and FC-CS

	SUT(n , d[ ])	Constrained SUT (c, k , t)	IPOG on Unconstrained SUT	PICT on Unconstrained SUT	IPOG with FC-CS on Constrained SUT	PICT on constrained SUT
1	(3, [3, 5 ,8])	(1, 3, 3)	45	44	44	43
2	(4, [7, 3, 7 ,8])	(4, 3, 2)	61	56	57	58
3	(4, [3, 5, 6 ,6])	(3, 1, 3)	60	62	60	60
4	(4, [8, 4, 5 ,7])	(3, 3, 2)	56	56	55	56
5	(3, [3, 5, 5])	(4, 1, 3)	75	76	24	24
6	(3, [4, 6 ,8])	(2, 3, 2)	48	48	46	46
7	(3, [8, 3 ,8])	(3, 2, 2)	64	64	59	59
8	(5, [8, 6, 6, 3 ,8])	(3, 3, 3)	414	432	425	430
9	(5, [6, 8, 6, 7,7])	(4, 3, 3)	453	466	454	464
10	(4, [7, 6, 7 ,7])	(3, 1, 3)	343	378	252	269
11	(5, [6, 8, 6, 7 ,7])	(4, 2, 3)	453	466	425	413
12	(5, [7, 4, 6, 8 ,7])	(2, 2, 3)	397	424	381	392
13	(5, [8, 5, 7, 5 ,4])	(3, 2, 2)	58	56	51	53
14	(4, [7, 8, 8,7])	(2, 2, 3)	461	459	479	472
15	(4, [7, 6, 7, 7])	(1, 3, 3)	343	378	422	424

6.3 Impact of constraint handling on execution time

To investigate the execution time impact of running IPOG with FC-CS on a constrained input model, we use two classes of CSP's namely, binary and non-binary. For binary CSP, we use two system configurations. System A has n fixed to 4 and d varied from 5 to 20, while System B has d fixed to 15 and n varied from 2 to 10 in increments of 2. In both systems we set t to 3. Table 6 shows the result of building System A for different d values and Table 7 shows the result of building System B for different n values.

Table 6: System A showing time for d between 5 and 20

d	Time(Sec)
5	0.42
6	0.66
7	1.17
8	1.83
9	2.58
10	4.09
11	5.55
12	7.72
13	10.53
14	14.38
15	18.42
16	23.95
17	30.98
18	39.50
19	48.67
20	60.0

Table 7: System B showing time for n between 2 and 10

n	Time(Sec)
2	199.61
4	314.72
6	634.98
8	751.33
10	831.30
12	1056.39
14	1226.38

The result shown in Table 8 and 9, show that the running time of IPOG with FC-CS on binary CSP depends on the number of variables and the maximum domain size. We performed curve fitting analysis and extrapolation on the data and conclude that that for binary CSP, the asymptotic time complexity is bounded by $O(n^2d^k)$.

To investigate non-binary CSP with hidden variable encoding, we define 3 system configurations and each system has n = number of variables, d = maximum domain size, k =

arity of constraints, c = number of constraints and e = number of hidden variables. System A had n = 5, k = 3, c = 3, e = 2 and d is varied between 5 and 15, while System B has n = 5, k = 3, c = 10, d = 10 and e is varied between 1 and 10 and System C has n = 5, k is varied between 1 and 4, d = 10, c = 3, and e = 1. Table 8, 9 and 10 show the result of running IPOG with FC-CS on the system configurations with t = 3.

Table 8: Result of System A with d [5 to 15]

d	Time(sec)
5	1.11
6	2.53
7	5.52
8	11.34
9	25.3
10	52.63
11	106.83
12	214.92
13	409.17
14	760.55
15	1333.33

Table 9: Result of System B with e [1 to 10]

e	Time(sec)
1	24.23
2	47.47
3	80.75
4	95.38
5	125.33
6	148.1
7	163.56
8	186.55
9	222.7
10	253.22

Table 10: Result of System C with k [1 to 4]

k	1	2	3	4
Time(sec)	2.78	5.95	11.70	699.4

We note from the table above that IPOG with FC-CS performs better on non-binary constraints when the number of satisfying tuples is small. We performed curve fitting analysis on the execution time data, and the analysis showed that the running time is bounded by O(ekd^{k+1}).

CHAPTER 7

CONCLUSION

In this thesis we provide a general, scalable and re-implementable solution to the problem of constraint handling. We describe our algorithmic extension and compare the results produced by our algorithm with the results from an existing tool. Our constraint handling strategy allows the entry of constraints at higher level of abstraction, which is a departure form the method of specifying constraints by existing tools. The resulting size of test suites generated by a combination of IPOG with FC-CS is comparable with that generated by existing constraint handling tools and our strategy is effectively integrated into the test generation algorithm and not through any third party tool.

REFERENCES

[1] Myra B. Cohen, Matthew B. Dwyer, Jiangfan Shi, " Interaction Testing of Highly-Configurable Systems in the Presence of Constraints", in Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2007

[2] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, James Lawrence, "IPOG: A General strategy for T-Way software Testing". In proceeding to the 14th Annual IEEE International conference and workshops on the Engineering of computer-based systems.

[3] Renee C. Bryce and Charles J. Colburn, "The density algorithm for pairwise interaction testing" In Software Testing, verification and reliability. 17:159 – 182, 2007

[4] R.C. Bryce and C.J. Colburn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints". Journal of Information and Software Technology, 48(10): 960 – 970, 2006

[5] David M. Cohen, " The AETG System: An Approach to Testing Based on Combinatorial Design", In proceeding of the IEEE Transactions on Software Engineering Vol. 23, No 7, July 1997.

[6] IBM alphaworks. IBM Intelligent Test case Handler

[7] http://www.alphaworks.ibm.com/tech/whitch,2008

[8] J. Czerwonka, "Pairwise Testing in real world", In Pacific Northwest Software Quality Conference, pages 419 – 430, 2006.

[9] Samaras Nikolaos, Sterqiou Kostas, "Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results," Journal of Artificial Intelligence Research, vol. 24, no. 7, pp. 641–684, 2005.

[10] F. Bacchus, P. van Beek, “On the conversion between Non-Binary and Binary Constraint Satisfaction Problems”, in proceeding National Conference on Artificial Intelligence (AAAI-98), Madison, Wisconsin, 1998.

[11] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. 2nd Ed. Prentice-Hall, 2003.

BIOGRAPHICAL INFORMATION

Anthony Opara was born in Nigeria. He received his Bachelor of Science in Computer Science in 2001 from the University of Nigeria, Nsukka (UNN). In fall 2006 he started his graduate studies in Computer Science at the University of Texas at Arlington. He received his Master of Science in Computer Science in July 2008.