SUPERFIT COMBINATIONAL ELUSIVE BUG DETECTION

R. Barzin, S. Fukushima, W. Howden, and S. Sharifi
CSE, UCSD, La Jolla, CA, 92075

Abstract

Software that has been well tested and analyzed may fail unpredictably when a certain combination of conditions occurs. In Bounded Exhaustive Testing (BET) all combinations are tested on reduced versions of a problem/application with the idea that bugs associated with combinations for full versions of a program may also show up when combinations are tested for the reduced version. In previous work, a class oriented JUnit framework approach to BET was introduced, along with the idea of a BET test pattern. In this paper we considered the application of BET to system testing, using an extension of the FIT (Framework for Integrated Testing) framework called SuperFIT. This approach is described along with a simple example of the application of a SuperFIT test generation tool.

Keywords testing, bugs, elusive, patterns, combinations, frameworks, JUnit, BETUnit, FIT, SuperFIT

1. Elusive bugs, partition testing and BET

The characteristic feature of an elusive bug is that it is caused by a combination of conditions that are not reflected in functional specifications. Elusive bugs are not reliably detected using single-property partition testing rules such as "test each functional input partition", "test each partition boundary". Even sophisticated boundary analysis approaches may miss the relevant test [1]. In [5], Hamlet described a formal basis for analyzing the limitations of partition testing [5]. Relevant analysis is also described by Weyuker and Jeng in [6].

Partition testing was extended using approaches that try to identify "meaningful" combinations. In [2], Richardson combined specification conditions with implementation conditions. Other work introduced the use of *cause-effect graphs*, which identify functionally relevant combinations from specifications [3]. Ostrand provided a language and test generation tool that facilitated the generation of combinational tests that were implicitly defined in a test specification [4].

The combination problem has also been attacked using combinationally oriented program coverage measures, such as data flow testing, in which pairs or groups of statements that have a data flow relationship must be tested together on at least one test [e.g. 7-9]. Unfortunately, data flow testing has not been reported as being markedly more effective than ordinary branch and coverage testing [10]. Also, we have the following problem: if the defect in the code is the omission of a condition that is part of the combination that causes the problem, then the required test will not be produced.

Defect-catalog approaches to test selection attempt to categorize and list classes of potential bugs in particular application areas. They may also include associated test specifications. The catalog approach was systematically described by Marick in [11]. More recent work, aimed at specific application areas, can be found in publications such as [12]. In [13] Binder gave an in-depth description of a pattern-oriented approach to the description of testing methodology. In [14] Howden investigated a pattern-oriented cataloging approach that was specifically aimed at combinational tests for the detection of elusive bugs. Although this pattern-based approach was found to be useful, there were difficulties. It was not easy to classify defects into meaningful classes and to create a hierarchy that made them easily retrievable. More importantly, the approach involves constructing descriptions of defect revealing tests after such defects occur, when what we really want is to have patterns for those defects that have not yet occurred. To some extent, this is avoided by generalizing to classes of defects, but elusive bugs often correspond to some new application specific peculiarity.

An alternative approach is to try testing "all combinations" in a limited version of an application, with the hope that failures that are caused by combinations in a full version are also caused by combinations in the limited version. This idea was present in an early form in the work on path-oriented testing methods. Symbolic evaluation testing methods, for example, examined all paths up to some number of loop iterations, with the idea that bugs would show up on limited versions of the application that only required small numbers of loop iterations. [e.g. 15-19]. This idea of testing limited versions of an application that correspond to some small number of iterations of a loop is a well known method [20].

In more recent research, data-oriented instantiations of the limited problem approach are referred to as BET (Bounded Exhaustive Testing). The assumption that bugs which show up in larger versions of a problem will also show up in some smaller version has come to be called "the bounded testing hypothesis" [23]. In its modern form, BET differs from the path-oriented work in that it involves direct generation of test data combinations, rather than indirect generation from a set of control paths. Automated test generation was found to be difficult in the path oriented approach.

References [21, 22] describe modern BET approaches. Both include class testing tools. They work directly with application class descriptions to produce all instances of data and class structures up to some size, which are then used to automatically test instances of an application class.

The work that is described here follows the BET approach to the detection of elusive bugs. It differs from earlier BET work in that it is *framework based*. The initial work on BET frameworks, described in [24], was class oriented, and produced an extension to JUnit called BETUnit. Frameworks based on JUnit 3.8 and 4.1 were developed. Both allow the definition of combinational test cases and test suites that can be automatically run using an associated test runner.

The advantages of the framework approach include the easy incorporation of alternative "domain generator" plug-ins that allow a tester to use different combinational strategies for test generation. This differs from the approach used in the Korat tool [20], for example, which contain fixed, built-in combinational mechanisms, which may not directed enough when the goal is to test over particular kinds of promising combinations. Simple domain generators return the elements of a simple finite sequence, one at a time. More complex generators can be built from simple or other complex generators and may return structures or combinations. Combinational generators include "all-permutations", "all stratified permutations", and "all-pairs". The framework approach is also convenient to use because of its reliance on test fixtures. By fixtures, we mean code that maps a test specification onto program execution. This "delinking" of test descriptions and application execution facilitates the use of new implementations of the same test, or alternatively, reuse of application dependent test code in new tests.

In both the earlier and the current work, a pattern-oriented approach was adopted. The tester identifies testing situations in which BET testing may be useful and then constructs test cases with a BET component. These are constructed within the relevant framework, using automated test generation and execution. Test patterns guide the tester in performing required test construction tasks, and suggest alternative kinds of strategies. In our research on JUnit BET we found that test patterns may contain both a test generation and an associated test oracle component. Certain kinds of test generators go with certain kinds of oracles. For example, BETUnit test generators were used that generate abstract metadata along with tests. The metadata can be used to construct an abstract, partial program simulator that acts as an oracle. Several kinds of such patterns were identified in the BETUnit research.

Our previous BET work focuses on class oriented automated testing. The work in this paper describes the application of the framework strategy to systems testing, and is based on the FIT framework.

2. SuperFit

2.1 Introduction

In an interactive application the program starts in a particular state and then goes through a sequence of intermediate states as a sequence of user steps is performed. We normally try to test each state, or even all possible pairs of states, but it is usually not possible to test all possible sequences of states. Elusive bugs have been observed which occur when steps are performed in particular sequences. SuperFIT is a framework for automatically generating and testing all possible sequences up to some predetermined length. Our approach to the specification of test step sequences, and hence the basis for SuperFIT, is the FIT testing framework [25].

2.2 Review of FIT (Framework for Integrated Testing)

FIT belongs to the general class of table-driven testing methods. Tables specify sequences of steps. The steps are meant to be application dependent but independent of test code. A major idea in the approach is that tables can be constructed and understood by non-programmers. Each step has to be mapped on to the application under test.

In FIT there are several kinds of tables, but we will only consider *action tables* here. Each FIT action table is associated with an action table fixture. The fixture has methods whose names correspond to the steps in the table. A table test runner executes the steps in a test by calling the methods in the associated fixture.

FIT action tables have four kinds of steps: *start*, *press*, *enter* and *check*. Each step in the table starts with one of the four action keywords. This is followed by the name of the associated fixture method, and then any appropriate parameters.

A *start* step is followed by the name of the fixture class to be used for the table. An *enter* step is followed by the name of a method, and then a list of the input parameters for that method. A *press* step has a method name, but no parameters. A *check* step names a method that returns a value that is compared with the parameter in the step. It functions as an assertion, or test oracle, that confirms the validity of computed intermediate and final values.

2.3 Basic structure of SuperFIT

The core of the SuperFIT approach is the test generation model and model traverser. The model is a directed graph, similar to those used in model-based testing [eg. 26-29]. Tests are generated by traversing the graph. SuperFIT models differ in that they incorporate test fixtures, and are test generation models rather than specifications. The SuperFIT test generator traverses paths in the model graph, using a depth-first strategy, up to some predetermined path length. For each path, it generates a FIT test table.

SuperFIT model edges can be labeled with three kinds of steps: Start, Feasibility and Generate. As in FIT, the Start step identifies a test fixture to be used for interpreting the other steps in the model. Also, as in FIT, the other steps identify methods in the test fixture. If an identified method has parameters, values for these will appear in the associated SuperFIT step.

Feasibility steps identify a method that is used to see if the current transition is "feasible" in the context of the traversed subpath which has led to that transition. This is necessary because SuperFIT models can be abstract in the sense that, depending on the context leading to a step, a next step may or may not be possible in the actual system. If a Feasibility step returns F, then the associated transition is not followed and the test generator backs up the last branch point at which there is an unfollowed transition.

SuperFIT fixture methods associated with Generate steps construct FIT table steps. In many cases, a Generate step will simply copy an identified FIT step. Some FIT steps, such as *enter* steps have input parameters. SuperFIT Generate steps can identify either simple input parameters, or domain generators that can be used to generate a finite set of possible alternatives. Suppose, for example, that a domain generator g() is identified in a Generate step for a FIT *enter* step. The SuperFIT model traverser will generate instances of the *enter* step for each of the elements returned by g(). When the SuperFIT graph traverser is in backup mode, backing up to a previously unexplored alternative, it will initiate a new path for either an unfollowed graph transition, or an unused domain generator alternative.

Additional, more sophisticated model features are also possible. For example, model transitions having both a Feasibility and a Generator step may specify a Generate fixture method that depends on data returned by the Feasibility method.

Feasibility steps are similar to guard conditions in state models. In some examples of model-based testing, all model paths are feasible [e.g. 32] so that feasibility is not an issue. In others, embedded directives avoid the generation of infeasible paths [e.g. 30]. Others construct a set of constraints from path conditions whose solution corresponds to a test that causes that path to be followed [e.g. 29]. In this last case, infeasibility corresponds to constraint sets with no solution. The constraint set approach to model-based test generation is similar to early attempts at automated test generation involving symbolic evaluation [eg. 18]. The computational difficulties in these approaches limited their general acceptance. The examples we have been working with are typical interactive database oriented applications in which feasibility conditions usually determine whether or not certain data would exist in a data base after a short sequence of simple data base operations. For this important class of applications, we found it possible to devise effective feasibility checks.

As in the case of BETUnit, test patterns have been found that document ways of applying a BET framework approach. SuperFIT patterns suggest approaches to model construction, such as the FeasibilityGuardAndStateCheck pattern described below. Other patterns describe ways of solving specific problems in the testing process, such as how to perform feasibility checks.

2.4 Example

2.4.1 Application description

We will illustrate FIT and SuperFIT with a simple Dating System (DS) example. In the example the user is first presented with a screen with two choices: Start and Stop. If Stop is chosen the program terminates. If Start is chosen a Logon screen is displayed. The user can enter a name in the textbox and then press the Enter button. If the name is "William" the Administrator Screen comes up, on which the user can choose the Delete or Add button. This will bring up a screen for entering a name, which causes a member of the DS to be deleted or added. In the case of the Delete option, if there is no member with this name an error screen comes up, otherwise the member is deleted and a success message is displayed. In the case of the Add option, if there is already a member with that name, an error message is displayed. Otherwise the member is added and a success message is returned.

If the name entered at LogOn is a member in the system, the member screen is displayed. There are two options: GetADate and SetMemberData. If the date option is chosen, a screen comes up on which the user can choose entries for several categories. The user can then push the Enter button and the system looks for a date. If a date is found, the data for that member is displayed. Otherwise, a NoDate failure message is displayed. If the SetMemberData option is chosen, then a screen comes up that allows the user to enter his or her data. When the Enter button is pushed the data is entered in the data base. Finally, if the name that was entered at logon is not in the data base then an Unauthorized User message is displayed.

The display logic for DS prevents certain kinds of actions. For example, the administrator William cannot ask for a date, and a member of the dating system cannot add or delete members.

The DS program is composed of three subsystem tiers: the GUI, the BusinessLogic, and the DataBase. In its application to an interactive system, the usual approach in FIT is not to exercise the system through the GUI, but to directly call methods in the lower level tiers. In this case, we would test the system by making direct calls on methods in the BusinessLogic interface. In a sense, the test fixtures in FIT simulate the GUI by calling lower tier methods when certain actions are assumed to have been carried out through the GUI.

Table 1 contains a sample test in which the user logs on, is found to be an administrator, and then unsuccessfully tries to delete someone named Fred. The first step (*start*) causes the creation of an instance of the test fixture DSTest. The "*press* start" step results in the state in which the user can enter a name and log on, which is simulated by the following two steps in the table. Following this, there is a *check* to see if the system is in the expected Admin options state S. This is accomplished by the getUserType() method in the DSTest fixture. getUserType() calls the appropriate method(s) in the DS BusinessLogic subsystem. When in the state S, the user can enter a name and choose the delete option, simulated by the next two steps. This

Table 1. Sample FIT test table for deleting a member

start	DSTest	
press	start	
enter	name	William
press	login	
check	getUserType	Admin
enter	name	Fred
press	deleteMember	
check	latestResult	Unsuccess
press	end	

action should result in a message telling the user that the delete was not successful. In the associated fixture, this corresponds to the "*check* latestResult" step, which should return Unsuccess. Finally, the "*press* end" step shuts down the DS application.

2.4.2 SuperFIT table generator model

Figure 1 contains part of the SuperFIT model for the Dating System example, represented as a directed graph. Feasibility confirmation steps are represented as state chart transition conditions, enclosed in square brackets. For simplification, the Generate keyword has been omitted from SuperFIT steps in which FIT table steps are generated. All FIT *enter* steps correspond to output that would be produced from corresponding superFIT Generate steps. *enter* step argument(s) may be simple values, or domain generators which produce finite sets of possible inputs to that FIT step. The graph in Figure 1 represents the initial part of the model where someone logs on. Name is a domain generator that generates a small set of possible input names. The model specifies Feasibility steps to control choices of paths and generates FIT *check* steps that will confirm that the state of the system is what it is expected to be. The entire model for the DS example consists of 4 graphs similar in size and structure to the model in Figure 1.

Path pattern recognition was used to implement feasibility steps. It was implemented using the following methods, which operate on a partial path up to an embedded Feasilility step that:
IsLatestLoginAuth(), IsLatestLoginAdmin(),
GetLatestLogin(), IsLatestEnteredNameInDB(),
GetLastEnteredName(),
HasRequiredProperties(), GetMemberData(),
IsAnyMemberWithRequiredProperties().

The functionality of the methods is described by their names. For example, GetLatestLogin() returns the name of the person that was last logged in. This is done by looking backwards along the path to find the last *press* step for the "login" button. If it is found, then we go back one step more to get the name of the person. If none is found, then no one has logged in yet, so the method returns an empty string.

In our prototype we implemented the above methods directly. This would be onerous if similar methods had to be reconstructed for every new application model. Underlying application-independent methods can be developed to facilitate the construction of application-dependent feasibility checkers. For example, the first two methods could be implemented using a generic SuperFIT path pattern recognizer isLastEntered(s), with the obvious interpretation. More complex generic methods might allow the use of path index variables so that one method could be used to identify the path position of a pattern, which is then handed to another pattern recognition routine that starts from that position.

2.4.3 Sample defects

When the sample DS program was originally constructed, a number of defects occurred. Several elusive defects that made it past traditional testing efforts were detected by SuperFIT. For example, if there is an attempt to delete a nonexistent member before an existing member is deleted, then the system will fail. However, deleting a nonexistent member after having already deleting an existing member (which may or may not be the member that the user tries to delete later) does not result in a failure. This is exactly the kind of elusive bug for which SuperFIT works well.

Another elusive defect failure occurs if a member is added to the DS data base by the administrator during a session, and the administrator terminates the system session before that member logs on and sets their data. When the system is restarted, it fails. This is again a particular combination that may not be tested using traditional methods. Multi-session paths were made possible in the testing of DS by including a program-start step. The program terminate step corresponds to choosing the End option on the Start/Enter screen.

We ran the SuperFIT automated testing system with different path lengths and with domain generators designed to return small sets of data for the input to FIT *enter* Steps. Table 2 describes two of the BET tests that were run. For each test we give the test size control parameters, such as path length and number of alternative inputs to *enter* Steps. There were five *enter* steps, for gender, name, occupation, religion and email address. In the first test, for example, the domain generator for religion was constructed to return five alternatives. Paths will be traversed for each of the combinations of

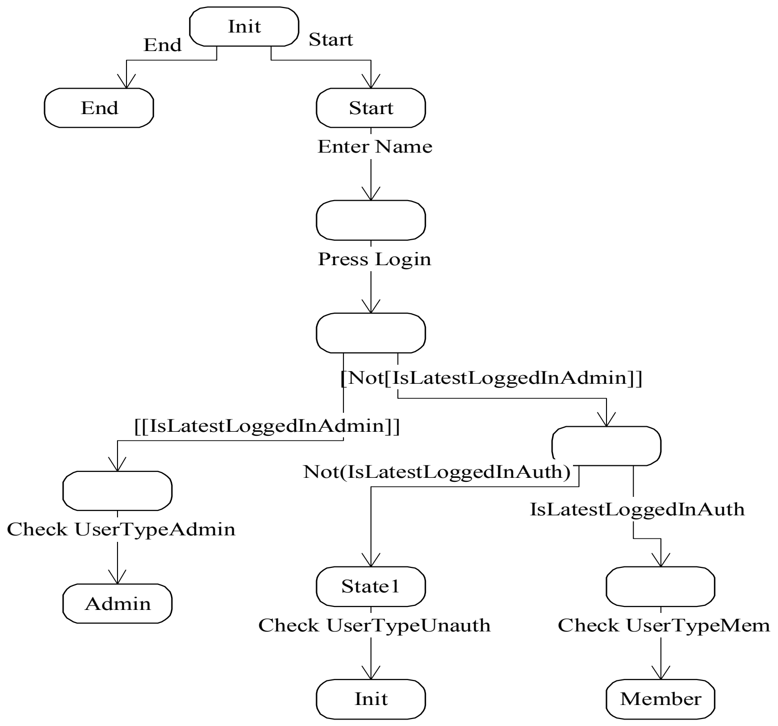

Figure 1: SuperFit Test Generation Model

of inputs that can occur along the path. The table also shows the number of tests that were generated for each setting.

All of the known elusive bugs in the DS were detected during the execution of the FIT tables generated by SuperFIT. The "delete non-existent member" bug, for example, was detected by the test described in Table 1. When this table is run by FIT we get a test report in which it is noted that instead of getting the expected "Unsuccess" message to the last *check* step, an "error" message is generated that corresponds to an application program failure.

DS is a database oriented system.. Two possible approaches to testing such systems are: i) have tests that specify the initial setting of the data base before the user action steps begin, or ii) assume that the data base is always in the same initial condition. The second facilitates test repeatability. For example, when the SuperFIT tester specifies Feasibility statements, they will be accurate relative to an assumed data base initialization. We assumed that the data base would always be initially empty. This seems the simplest approach, and it also had the following advantage: it is difficult to predict in advance what initial combinations of data will be good for the detection of elusive bugs. If we start with an empty data base, then we will by default get all BET-oriented initial settings by traversing path prefixes that enter those combinations.

2.4.4 SuperFIT test patterns

Abstract test generation models may have states S with one or more outgoing transitions that in real life can only actually occur for particular concrete instances of S. Consequently, as in the above DS example, it is necessary to put Feasibility guard conditions on the transitions to avoid the generation of invalid test scripts, leading to script failures. Suppose that a state S is followed by a transition having a user action A, leading to a state S'. Suppose that A can lead to different antecedent states, depending on which concrete instantiation of S has occurred during program execution. This can be represented in the model using transition guards. Suppose S' is an antecedent state, associated with a guard condition G on the transition for A. Typically, S' will often be closely associated with G. This leads to a natural FIT *check* step at S' which will validate that the state S' that the test constructer expects to occur when A is traversed with guard G is the actual state that is reached during program execution.

Table 2. Statistics for sample DS SuperFIT Tests

Max path length	20				
Domain generator bounds	#Gender =2	#Name= 5	#Occup. =5	#Religion =5	#eaddr =5
Num of Tests	2372				
Max path length	40				
Domain generator bounds	#Gender =2	#Name =2	#Occup. =2	#Religion= 2	#eaddr= 2
Num of Tests	7371				

In the example of Figure 1, when the system is in abstract state "Start", and EnterName/Press Logon is performed, if guards Not[IsLatestLogInAdmin] and [IsLatestLogInAuth] hold, then there is a transition to the "Member" state in which various member actions can be performed (i.e. can be generated in the FIT table being constructed). To confirm at run time, that the system really is in the Member state before performing an action that assumes this, a "*check* UserTypeMem" validation step can be generated for insertion in the FIT table under construction.

This situation occurs often enough that it is useful to characterize it as a SuperFIT test pattern, to assist future testers in building test generation models. Similarly, in the above example, we illustrated the use of "path pattern feasibility checking" in constructing SuperFIT feasibility checkers, another useful SuperFIT testing pattern.

3. Summary and conclusions

The framework approach was found to have the advantages of object orientation: reusable, expandable, and easy to integrate with other features such as generic domain generators. The SuperFIT project was directed at interactive systems. We extended the FIT approach, resulting in a tool that can be used to generate FIT tables. As in the earlier, BETUnit class-oriented research [24], the focus was on elusive bugs and the use of bounded exhaustive combinations for their detection.

Our approach in SuperFIT uses ideas from model-based testing, where tests are generated by traversing model paths. The targeted application domain, interactive programs based on data bases, allows the use of simple guards in abstract models to prevent the traversal of infeasible paths.

In SuperFIT we used abstract models with transition conditions because one of the basic ideas in the FIT strategy is to use test specifications that are constructible by non-programming testers and other stakeholders.

In previous work [14], we attempted to use the patterns concept for a catalog oriented approach to elusive bugs. Particular kinds of defects were specified using associated defect revealing test patterns. This proved to have limits, of which one was the "hindsight factor": we only add patterns after a defect is discovered. This may not help us with kinds of defects yet to be seen. The promise of BET is to finesse this limitation. Previously unknown defect test patterns "arise" anonymously and are invoked during the BET test generation and execution process.

We found the pattern approach to be an effective strategy for formulating test generation descriptions and in this work we applied it to BET oriented system testing. Several useful BET test patterns were developed during the SuperFIT project, including *PathPatternFeasibilityCheckers* and *FeasibilityGuardAndStateCheck*. In the first of these, when the SuperFIT test generator encounters a transition condition it computes the condition truth value from the pattern of steps that occurs on the earlier part of the path. The second pattern concerns a strategy for SuperFIT model construction in which transition arcs that contain Feasibility guards are followed by SuperFit steps that generate corresponding FIT *check* steps. The *check* step validates, during test execution, that the state which has been reached in the test is consistent with the SuperFIT

Feasibility guard condition that led to its occurrence.

Future work includes continued application of SuperFIT to interactive, data-based systems, and BET testing patterns development.

4. Acknowledgements

The authors would like to thank H. Hamedtoolloei and I. Sadeghi who helped to build the SuperFit system.

5. References

[1] White, L J., Cohen, E.I., AND Zeil, S.J., A Domain Strategy for Computer Program Testing, in *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, Eds., North-Holland, Amsterdam, 1981.

[2] Richardson, D.J., Clarke, L. A Partition Analysis Method to Increase Program Reliability, Proceedings ICSE 5, IEEE, 1981.

[3] Myers, G., *The Art of Software Testing*, Wiley Interscience, 1979.

[4] Ostrand, T.J., Balcer, M.J., The Category-Partition Method for Specifying and Generating Functional Tests. *Commun. ACM,* 31-6, 1988.

[5] Hamlet, R. Partition Testing Does not Inspire Confidence, *IEEE TSE*, 16- 12, Dec. 1990.

[6] Weyuker, E., Jeng, B., Analyzing Partition Testing Strategies, *IEEE TSE*, 17-7, 1991.

[7] Laski, J.E., and Korel, B., A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9-3, 1983.

[8] Clarke, Lori. A., Podgurski, Andy, Richardson, Debra J., Zeil, Stephen, J., A comparison of data flow path selection criteria, *Procs. ICSE 8*, IEEE, 1985.

[9] Rapps S., Weyuker, E.J., Selecting Software Test Data Using Data Flow Information, *IEEE TSE*, 11-4, 1985.

[10] Frankl, Phyllis G., Weiss, S.N., An Experimental Comparison of the Effectiveness of the All-uses and All-edges Adequacy criteria, *Proceedings of the Symposium on Testing, Analysis, and Verification*, IEEE, 1991.

[11] Marick, B. *The Craft of Software Testing, Prentice Hall*, 1995.

[12] Jha,Ajay, Kaner, Cem, " Bugs in the brave new unwired world." *Pacific Northwest Software Quality Conference,* Portland, OR, October 2003.

[13] Binder, R. *Testing Object Oriented Systems*, Addison Wesley, 2000.

[14] Howden, W.E., Software Test Selection Patterns and Elusive Bugs, *Proceedings COMPSAC*, IEEE, Edinburgh, 2005.

[15] Howden, W. E., Methodology for the Generation of Program Test Data, *IEEE Computer*, 1975.

[16] Clarke, L.A. A System to Generated Test Data and Symbolically Execute Programs*, IEEE TSE*, 2, 1976.

[17] King, J.C., Symbolic Execution and Program Testing, *CACM*, 19, 1976.

[18] Boyer, R.S., Elsaps, B., Levitt, K.N., SELECT - A Formal System for Testing and Debugging Systems by Symbolic Execution, Proceedings 1975 ICRS, IEEE, 1975.

[19] Howden, W.E., *Functional Program Testing and Analysis*, McGrawHill, 1987.

[20] Beizer, B. *Black-Box Testing*, Wiley, 1975.

[21] Boyapati, C., Khurshid, S., Marinov, D. Korat, Automated Testing Based on Java Predicates, *Proceedings ISSTA*, IEEE Press, 2002.

[22] Cheon, Y., Leavens, G., 2002. A Simple and Practical Approach to Unit Testing: The JML and the JUnit Way, In *ECOOP 2002 -- Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 2002, Proceedings.* Volume 2374 of *Lecture Notes in Computer Science*. Springer-Verlag.

[23] Jackson, D. Schechter, H, Shlyahter, A, Alcoa: the Alloy Constraint Analyzer, Procs. 22nd ICSE, IEEE 2000.

[24] Howden, W.E. and Rhyne, C, Test Frameworks for Elusive Bug Testing, *Proceedings ICSOFT*, Barcelona, 2007.

[25] Mugridge, R., and Cunningham, W., *FIT for Developing Software - Framework for Integrated Tests*, Prentice Hall, 2005.

[26] Offutt, J., Liu, S., Abdurazik, A., Amman, P., Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 2003, vol. 13, 25-33.

[27] Andrews, A.A., Offutt, J., Alexander, R.R., Testing Web Applications by Modeling with FSM's, *Software Systems and Modeling*, 4(3):326-345, July 2005.

[28] Farchi, E., Harman, A., Pinter, S.S., Using a model-based test generator to test for standard conformance. *IBM System Journal*, Vol. 41, No.1 2002.

[29] Robinson, H., Finite State Model Based Testing on a Shoestring, *STAR West*, 1999.