TEST FRAMEWORKS FOR ELUSIVE BUG TESTING

W.E. Howden
CSE, University of California at San Diego, La Jolla, CA, 92093, USA
howden@cse.ucsd.edu

Cliff Rhyne
Intuit Software Corporation, 6220 Greenwich D., San Diego, CA, 92122 USA
cliff_rhyne@intuit.com

Keywords: Testing, elusive, bugs, frameworks, bounded exhaustive, JUnit

Abstract: Elusive bugs can be particularly expensive because they often survive testing and are released in a deployed system. They are characterized as involving a combination of properties. One approach to their detection is bounded exhaustive testing (BET). This paper describes how to implement BET using a variation of JUnit, called BETUnit. The idea of a BET pattern is also introduced. BET patterns describe how to solve certain problems in the application of BETUnit. Classes of patterns include BET test generation and BET oracle design. Examples are given of each.

1 INTRODUCTION

1.1 Background

A variety of defect detection testing guidelines have been proposed. Some are based on empirical evidence that certain kinds of test cases, such as so-called "corner cases" or "boundary cases", are more prone to be associated with defects. Others require that all parts of the code have been tested, as in branch or statement testing.

We are interested in a certain kinds of defects that are not amenable to discovery using standard testing methods, which we have called "elusive bugs". This kind of bug often has the following characteristics: i) it occurs late in the testing cycle or after release, ii) it occurs when a certain combination of conditions takes place, and iii) it is not reliably found by standard testing methods.

Approaches to the discovery of these kinds of defects include the use of area-specific defect lists [e.g. Jha, Kaner, 2003] and test selection patterns [e.g. Howden, W.E., 2005]. Lists and patterns do not work well for the following reasons: they quickly become too long, they are difficult to organize into useful classes, and they are based on hindsight - the next defect may require yet another addition to the list.

This paper is concerned with the use of "Bounded Exhaustive Testing" (BET) for the detection of elusive bugs. More specifically, it is concerned the development of a testing tool similar to JUnit that facilitates BET.

Our discussion of the problem, and the BETUnit approach, will involve the following example.

1.1.1 Account Break Example

A production accounting program contained code for processing a sorted file of transactions. Each record had an account number and was either financial or non-financial. The program was supposed to construct totals for the financial records for each account, which would appear as a group in the sorted file. It processed the records one at a time, and was supposed to output the current account's total when it observed an "account break". A break occurs when the account number in the transaction record changes.

The bug in the program occurred because it failed to check for account breaks when the last record of a group was non-financial. Under certain circumstances, this would result in the incorrect addition of one account's transactions to the total for the next account.

One of the functional properties that a tester

might focus on is correct processing of account breaks. In addition, he would probably test for processing of both financial and non-financial records. But it is the combination of these two factors, along with data that would cause the defect to result in incorrect output, which is relevant.

1.2 Bounded Exhaustive Testing

In general, it is not possible to test a program over all possible inputs. In Bounded Exhaustive Testing, a bounded subcase of the application is formulated, and all possible behaviors of the subcase are tested. It is argued that many of the faulty behaviors that can occur for the general case will also occur for the bounded subcase. Our experience indicates that, in particular, the combinations associated with elusive bugs will occur in both the full-sized application and the bounded subproblem

Similar ideas have been used in the past. For example, when a program has loops for which the numbers of iterations that are carried out depends on input data, it is common to use bounded tests that will cause 0,1 and possibly one or two larger numbers of iterations. In [Howden, W.E., 2005], an approach to bounded exhaustive testing is described which uses real input to bound the problem and symbolic input to summarize the complete behavior of a program within the bounded domain. Model checking is also a kind of exhaustive approach, in which all states in a bounded version of the problem are examined. However, in model checking the focus is on analysis rather than testing.

Recent BET research has been carried out in the context of class-based unit testing and involves straight testing rather than testing combined with symbolic evaluation or analysis. In addition, it has resulted in new research on methods for defining and generating bounded input domains. One of the first of these, the Iowa JML/JUnit project, described in [Cheon, Y., Leavens, G., 2002], has a method for defining and generating BET tests. BET was also the focus of the Korat project research, described in [Boyapati, C., Khurshid, S., Marinov, D., 2002.].

1.3 Overview Of Paper

This paper is organized as follows. Section 2 is a review of JUnit. Section 3 describes the BETUnit approach and Section 4 describes an example scenario for BetUnit usage. Section 5 describes other work, and in particular, the Iowa JML/JUnit and Korat projects. Section 6 contains conclusions and future work.

2 JUNIT – REVIEW

JUnit consists of a collection of classes and a test automation strategy. The TestRunner class runs tests. It is given a file containing a class definition for a subclass of the TestCase class which is expected to contain a suite() method. suite() will return an object of type TestSuite, which contains a collection of test cases. Each of these test cases is an instance of a subclass of TestCase containing a test method. TestRunner executes each test object in the suite. It does this by calling its run() method.. It passes a TestResult instance as an argument to run(), which is used to record test results. run() runs the setUp(), runTest() and tearDown () methods in the TestCase subclass instance. runTest() runs the test method in that instance. There are several approaches to implementing runTest(). One is to have it run the method whose name is stored in a special class variable in the instance.

The user of JUnit has two options: default and custom. In the default use, the default definition for the suite() method is used when TestRunner is given an instance of a subclass S of TestCase. S must also contain all of the test methods, whose names must have the prefix "test". The default suite() method will use reflection to find the test methods. It will then build instances of S for each of the test methods. For each test x, the name of the test method can be stored in the special class variable that is used by the runTest() method to identify the test method to run. These TestCase instances will be added to the TestSuite instance that suite() it creates, which is what it returns to TestRunner.

In the custom approach, the tester creates a new subclass of TestCase with a custom suite() method definition. Typically, suite() will create an instance TestSuite, and then add instances of subclasses of TestCase. Each of these subclasses will have a defined test method that will be run when the TestRunner executes the suite. This method can be identified by the class variable x used to store the name of the method to be run. It can be set by using the TestCase constructor which takes a string parameter that will be stored in x. It is possible to create a composite of TestSuite and TestCase subclass instances. The composite forms a tree, with the TestCase instances at the leaves. The run() method for a TestSuite instance will call the run() methods of its TestSuite children. At the leaves, the run() methods for the TestCase instances will be

called.

JUnit uses an assertion approach to oracles. A special exception class called Assertion is defined. It is subclassed from Error rather than Exception because it is not supposed to be caught by the programmer's code. Programmers are required to insert assertions in their code, in the form of calls on the Assertion Class methods that generate the exceptions.

JUnit uses the Collecting Parameter pattern to return results. A collecting parameter is one that is passed around from one method to another in order to collect data. As mentioned above, TestResult objects are created and then passed to run() methods. They are used to record the results of tests. When a run() method catches an assertion violation, it updates the TestResult object passed to it. Different TestResult objects can be defined but there is a simple default version that records the kind of exception and where it was created.

Assertions are not always adequate or practical for use as test oracles. In the cases where it is not feasible to construct an assertion-based oracle we can settle for robustness testing, in which the only results that are reported are system failures. This can be done since the run() method in JUnit catches error exceptions and reports them. In other cases the best solution to the oracle problem is to have an externally defined second version of the program whose results can be compared with the application results. WE call this the "2-version oracle pattern".

Many varieties of JUnit have been produced. There's a JUnit port to C++ called CppUnit. NUnit is for the .net framework. Also, there is a rich library of tools available for JUnit. Development environments like Eclipse come with a GUI for running and examining the results of JUnit tests. JUnit tests can be run from Ant build scripts.

3 BETUnit

3.1 Possible Approaches

The goal of our BET work was to have set of classes like those in JUnit, which would allow the user to have full flexibility in creating tests, while at the same time having the convenience of a test runner and a set of test case classes from which he could inherit capabilities for creating his test cases. A central focus was the automated generation of combinations that will reveal elusive bugs.

One approach would be to have users subclass TestCase and provide a custom suite() method that would construct a complete suite of BET tests. Unfortunately, BET can involve a very large number of tests, so the memory requirements for this suite could be enormous, more than can be handled with a standard JUnit implementation.

Another approach is to have the user break down a BET set of test cases into batches. However, this is a clumsy high maintenance solution.

Any successful approach will have to allow for just-in-time generation of BET tests, where each test is run before the next is generated. One approach is to define a new subclass of TestCase that does this called BETCase, which is described here.

Another goal for the BETUnit project was to develop generic combinational generators that could be used to automatically test different input combinations, and to facilitate the use of different kinds of combinators, such as all-pairs, in addition to standard BET exhaustive combinations.

We used the concept of a *test domain*, which is a set of tests or test components. These are defined using TestDomain classes. Users of the domains generate instances of specialized subclasses of TestDomain, possibly supplying parameters to the constructor. TestDomain subclass instances are then used to automatically generate tests from the associated test domain. For example, intDomain generates elements from a specified range of integers. The constructor has min and max integer parameters. More complex TestDomains may have constructors that take other TestDomain instances as parameters. TestDomain classes all have a next() and a more() method. The latter returns true if there are more items to be generated. TestDomains also have other methods, such as reset().

3.2 Sample Approach

The approach described here involves the use of a subclass BETCase of TestCase. Testers will construct a subclass of BETCase, rather than a subclass of TestCase.

When a tester subclasses BETCase, he will supply a definition for initDomain() and a definition for a test method m. m is the method that creates instances of the CUT and then tests the CUT by executing its methods. initDomain() constructs a data generator object of type Domain and assigns it to a BETCase class variable called "testDomain". initDomain() will be called by the constructor for BETCase. When you call the first()/next() methodsof the testDomain object, it automatically generates the next test object to use in an execution of the test method m defined in the BETCase

subclass.

initDomain() may be written to use one of the standard BET generators, or it could contain its own class definitions to define or modify a standard generator.

The run() method in BETCase is similar to the run() method in TestCase, but with some added twists. It calls the setUp(), runTest(), and tearDown() methods. It calls these methods repeatedly until there are no more test data objects that will be returned from the generator set by initDomain(). runTest() will run the defined test method m, which in the variation of JUnit described above, is identified from the special class variable in the TestCase instance. Unlike classic JUnit, the test method m is expected to have input. runTest() calls the test method m with the data returned from executing x.next(), where x is the domain generator object set by initDomain(). next() is expected to return a data object of the type expected by m. For each test cycle, run() uses x.more() to see if there are more tests in the sequence to be generated.

3.3 Variations

There are several possible variations on the above. For example, we could write test methods m that get their input from a special class variable rather than a parameter.

Also, we could allow the use of more than one domain generator in a BETUnit subclass. This would involve the tester adding a set of testDomain variables, giving them values in initDomain(). It would be necessary to incorporate some kind of correspondence mechanism from the testDomain variables to either the class variables in BETCase or the test method input parameters, whichever approach is used for input to a test. In the case of parameters, typing could be used to perform the mapping correspondence.

3.4 Default suite() Option With Multiple Test Method BETCase Instances

In the non-default mode for JUnit, the user defines a suite() method in the TestCase subclass supplied to the TestRunner. A similar approach is used in BETUnit. The user creates a subclass of TestCase (i.e. of BETCase) with a suite method that returns a suite object. The test case vector in the suite object will contain one or more instances of BETCase subclasses, each having the required definitions for initDomain(), and the new run() method.

As mentioned earlier JUnit has a default approach in which the user can define a TestCase subclass with a set of test methods, all of which begin with "test". The default suite() method uses a special constructor for a TestSuite instance which takes the TestCase as a parameter. The suite constructor then generates individual instances of the TestCase subclass, one for each test method, which it puts in its suite vector. A class parameter is set to identify the test method of interest for that subclass instance.

The BETUnit analog to the JUnit default version is the following. The user supplies a BETCase subclass with the default suite() method. The subclass has the new versions of run(), the domain generator variable, and one or more test methods whose name has the prefix "test". The default suite method uses a TestSuite constructor with the name of the BETCase subclass as a parameter. This constructor use reflection to find the test methods, and creates an instance of the BETCase subclass for each. Note that in this approach, it is assumed that all the test methods have the same input type, and are tested over the same set of BET tests, generated by the test generator installed by initDomain(). This limitation could be removed, at the cost of more complexity if it is seen to be necessary.

3.5 BET Runner

The same test runner used in JUnit can also be used in BETUnit. The user prepares a BETCase subclass, with the custom or the default suite() method. suite() generates a TestSuite instance, whose run() method is called by the test runner.

3.6 BET Oracle And Data Generation Test Patterns

Test patterns describe strategies for accomplishing different kinds of testing goals or tasks. Earlier we mentioned the 2-version oracle pattern, in which a second version of a program is used to serve as an oracle for a primary version. This pattern is attractive when an oracle version can be produced more easily than the application code, and using a different design approach to avoid coincidental common defects. We identified a useful 2-version pattern for BET testing of the Account-Break example. The central control structure of the application under test is a loop that reads in records, and has to perform account break activities when it detects that the account number has changed. The program has an asynchronous structure in the sense

that it is not possible to know in advance when the account break will occur. When test data is being generated by BET, this information is known in advance and could be supplied as input along with the record files, so that a "synchronous" oracle version could be built. Such a version would know exactly when the account breaks occur, so that it could use "for" loops with fixed bounds and increments rather than "do-while" loops. We call this the *Synchronized Sequence Oracle* pattern, and expect it to be widely applicable in BET oriented testing. We have also identified other BETUnit-oriented oracles that take advantage of the fact that i) the test cases will be "small" and ii) certain kinds of meta-data can be generated for each input case, which can be used to construct a different kind of design for the application.

Other BET-oriented patterns can be identified that are associated with test data generation. Recall that the goal with BETUnit is to find elusive bugs, and that these are associated with combinations. The *Sequence Permutation Test Generation* pattern is a variation on all possible combinations test generation. In this pattern, the tester prepares a seed set of data items such as instances of records. The test data generator generates all permutations of this set. It differs from all-combinations in that there is no exhaustive generation involving the possible data that is stored in the records, only the seed set is used that are permuted. Other BET oriented test generator patterns include the *Stratified Permutation* pattern used in the following example.

4 EXAMPLES

In the following two examples we will describe the use of two test generators that were applied to the account break example described earlier in the paper. The Synchronized Sequence Oracle pattern was used to construct an oracle. In the first example, the Sequence Permutation pattern is used was used for automated test generation, and in the second the Stratified Permutation pattern was used. For both generators, the bug was discovered. For each example we give some statistics associated with the generation and testing process. Both of the test generator classes that were constructed were just-in-time domain generators, generating the next test input as it is needed.

4.1 Permutation Generator

In this example, we assume the availability of the PermDomain() class. Instances of PermDomain will return all possible permutations of a set of objects. The set of objects is passed to the constructor in a parameter. PermDomain has a "just in time" generator for giving us the next permutation. Results are returned in a vector.

For this example, PermDomain is wrapped in AccountFileDomain. The next() method for AccountFileDomain returns an object with two parts, a metadata part and an instance of AccountFile. AccountFile instances are vectors of record objects. Record is also a class, whose instances are the kinds of records seen in the account break example. Since there are two kinds or records, we subclass to produce FinancialRecord and NonFinancialRecord.

AccountFileDomain instances are created with a seed vector of records that is used to create a PermDomain instance. The next() method of AccountFileDomain calls the next() method of PermDomain. Since the input files to the application are expected to be sorted, the next() method in AccountFileDomain also sorts them .

We used the Synchronized Sequence Oracle pattern for this example. The metadata part of an object returned by the next() method of AccountFileDomain is a vector with a number of items equal to the number of accounts in the associated AccountFile object. Each entry gives the number of records for an account. This information is determined by AccountFileDomain. It can determine this by first counting the number of different accounts, and then counting the number for each account. It is important that it compute this from the seed, rather than by reading through a candidate sorted input, detecting when the account breaks occur. If it did this, it would be duplicating the asynchronous nature of the application, and could have the same defects. The test methods in our BETCase subcase were written so that when they are handed an input test object, they know that the first part is the metadata and the second part is the input AccountFile. The whole object is given to a synchronous sequence oracle implementation of the account break application, which computes a result for the given file. The comparison is wrapped in an assertion.

The example was run with 4, 8 and 12-record seeds. The first had 1 account with 4 records, the second 2 accounts with 4 records, and the third 3 accounts with 4 records. The figures in Table 1

Table 1: Simple permutation generation

Input Size	***Test Case Count***	***Duration (sec)***
4 (1/4)	24	0.047
8 (2/4)	40,320	6.2

Input Size	*Test Case Count*	*Duration (sec)*
12 (3/4)	479,001,600	11,760 (3 h, 16 m)

indicate the numbers of tests run and the amount of time required.

4.2 Stratified Permutation Generator

The previous example is not as efficient as it could be since each input needs to be sorted before it can be passed to the function being tested. Many tests will be identical. Rather than generate and then rerun duplicates we devised an alternative approach in which we separated each account into its own permutation domain so that redundant tests are no longer generated. In the new approach we used a new general purpose domain generator called StratifiedPermDomain. There are several possible approaches to a StratifiedPermDomain. In one approach the domain generator takes a set of k vector objects. It generates all combinations in which there is a sequence of k objects, with the i'th object taken from the i'th set. Instances of StratifiedPermDomain are constructed with a vector of vectors. We used a variation on this idea.

For this application StratifiedPermDomain is wrapped in a StratifiedFileDomain generator. The constructor for this generator creates in instance of StratifiedPermDomain which it uses to generate AccountFile objects. As in the other example it also returns meta-data that is used by the synchronized sequence oracle for the application testing.

In our experiment with a stratified generator, we used seeds with 8, 12 and 16 records. The first seed had 2 accounts with 4 records each, the second had 3 accounts with 4 records, and the third had 4 accounts with 4 records each. Table 2 summarizes the numbers of tests run and the duration of the tests. The improvement in test case count and test duration in the new approach is staggering. Only 0.0029% of the test cases are generated from an input size of 12, and the test finishes in a little over a minute instead of over 3 hours. The domain creates $(x_1! * x_2! * ... x_n!)$ inputs, where n is the number of accounts and x_i is the number of records for account i.

Table 2: Stratified permutation generation

Input Size	*Test Case Count*	*Duration (seconds)*
8 (2/4)	576	.407
12 (3/4)	13,824	3.4
16 (4/4)	331,776	87

5 OTHER BET RESEARCH

5.1 Iowa JML Approach

A research team at Iowa State University developed a framework for Java class testing that incorporated a BET component. There were 5 key features in their approach: use of JML for assertions, pre and postconditions, postconditions as test oracles, automated generation of JUnit test classes from Java classes, exhaustive testing of all combinations of input values, and user determination of finite sets of values for seeding the tests.

The Iowa system takes a Java class and generates a TestCase subclass that contains a set of class variables that are used to organize the tests. Each of these is an array variable. The type of the first one is the type of the class under test, and is used to hold instances of that class, each constructed with a different set of actual parameters for the class constructor. The others correspond to the types of the parameters of the methods in the CUT. Each will be assigned a finite set of values that represents that type in the tests. A test method is generated for each of the CUT methods. The test method t for a CUT method m contains a set of local array variables, one for each parameter for m. These are initialized to the values from the class variables for parameter types. Each test method contains a set of nested loops that iterate over the test method arrays, constructing all possible combinations of values with one element from each array. This is then used to run the method in the class under test.

The tester subclasses the above test, constructing a custom setUp() method that assigns values to the class variable test data arrays. Recall that these hold two kinds of things. One is a set of instances of the CUT, created with different values for the constructor parameters. The others are arrays of values for the types of the CUT method parameters. These values are in turn assigned to the local variable arrays in a test method when it is run.

This system carries out the type of BET testing we are interested in, but is restricted in various ways. Since the combination mechanism is automatically generated there is no opportunity for the tester to try different kinds of combinatory mechanisms such as all-pairs or the permutation BET generators

described above in the Account Break examples. Another limitation is that the same set of instances of a type must be used for all method parameters with that type. There is no idea that different finite sets of values might be appropriate for different method parameters of the same type. There is also a common set of CUT object instances that must be used for the different test methods for the different CUT methods. Finally, there is the dependence on JML for assertions, which is not widely known or used. Related to this is the use of a postcondition oracle written in JML, which may not be appropriate for some programs that do not have a natural declarative specification.

5.2 Korat

Korat, like the Iowa system, focuses on automated test data generation and execution, use of preconditions to filter out invalid tests, and use of postconditions/assertions as program oracles.

The central driving entity in Korat is "finitization". This involves the creation of a finite domain of values that can be assigned to fields (class variables). For all primitive types, a finite set of values is chosen for the domain. For each field type class for a class, a finite set of instances of that class is created. The instances are indexed to identify them. This is called a *class domain*. Null is included in the domain for field variables whose value is a class instance. All of the primitive type finite domains, together with the class domains, form a total finite domain of field values D. The primitive domain contents and the number of instances of a class in a class domain are specified in a finitization object. There is a correspondence between each field in the objects in a domain D and the subsets of D that are candidate values for that field. A candidate vector is an assignment of properly typed elements of D to fields in elements of D.

If a class method has no parameters, then a set of values for its class's variables, taken from the finitization specified domain, forms the input for testing that method. If the method has parameters, then a class can be constructed whose fields correspond to those parameters, which is then used in the construction of the finitization domain for the parameters. This will result in the definition of a set of inputs for testing the method.

The candidate vector generation process incorporates two main optimizing strategies. The first uses preconditions to filter out invalid combinations. There could be many of these because any instance of a class C in a domain could be assigned as the value of any field of an object in the domain that has that type.

Precondition filter effectiveness is expanded as follows. Suppose that a precondition predicate evaluates to False for a candidate vector. This means the candidate is not a suitable test. Suppose that the precondition only involves the values of a subset of the candidate vector. This means that the precondition is independent of the other values in the vector, so any candidate vector which differs from the tested one only in the non-used variables can also be rejected as invalid

The second optimization strategy recognizes that when a class domain is constructed, the instances are not really distinguishable, so that test inputs that differ only in domain class instance indices will cause the same program behavior. Consequently, the test generator is designed to only generate a single instance of a possible class of such "isomorphic" values.

The test automation procedure in Korat is built into the system. It uses a fixed strategy for all applications, with special features to optimize the numbers of tests generated.

BETUnit also automatically generates tests but is more flexible. For example, in BETUnit we would not need an isomorphism suppresser for a program that manipulates binary trees because this could be built into the binary tree generator that was used by the test data generator for the application.

Korat differs from BETUnit in that all tests in Korat are generated before test execution begins. BETunit specifically depends on just-in-time generation to avoid huge memory requirements. Korat, like BETUnit, has test domains from which values are chosen. Unlike Korat, BETUnit incorporates the generation of the combinations into the domain objects from which the elements of the combination are drawn. This follows the expert/object animation pattern in object oriented programming. In the case of Korat, the combining mechanism is built into the system, making it more difficult to use alternative combining strategies such as all-pairs as opposed to all-combos.

The emphasis of BET is different from Korat. In JML/Junit and Korat the emphasis is on fully automatic testing. The emphasis in BETUnit is finding elusive bugs. It assists the user by providing automated test data generation classes that can be used to focus BET on possible elusive bug hiding places. Ongoing work focuses on the automated elusive bug detection aspect of BETUnit.

6 SUMMARY AND FUTURE WORK

Testing for elusive bugs involves running tests that explore different kinds of input combinations. Testing using rules that are based on specific application-oriented kinds of combinations has not been generally successful. There are often too many combinations and, in any case, we do not know what combinations to look at until after the defect has been discovered and analyzed. One way to solve this problem is to use some form of bounded exhaustive testing. However, we do not want to do this just for the sake of automating testing, we want to maintain some control over the combinatory mechanisms. BetUnit accomplishes this, both in the way it is a subspecialty of JUnit, and its use of separately defined combinatory mechanisms.

In our work we developed the concept of a *BET Pattern*. This is a test pattern that offers suggestions on certain aspects of BET Testing. One area of BET Patterns is test data generation where we defined two combinatory patterns: Permutation and Stratified Permutation. Both were applied to an Account Break example. We also found that we needed to consider BET-oriented oracle patterns. The effectiveness of postconditions for oracles, where they correspond to logical expressions, is too limited. As is well known in testing, many programs have an algorithmic rather than a declarative specification. For such applications we need a secondary version of the application program to test the output of the prime application program. The problem with this approach is that the two versions may contain the same defects. One way of avoiding this is to use different design strategies. We have identified several kinds of oracle design patterns, including the Sequence Synchronizing Oracle pattern used in the Account Break example, that produce "orthogonal" application program designs. In this example, the input generator returns certain metadata along with the suggested input. This metadata makes it possible to write a simpler oracle version of the program. More specifically, instead of waiting for some condition to happen during a computation, it knows in advance from the metadata exactly when it will happen. At the implementation level, this allows simple "do" loops instead of complex "while" loops.

In the version of BET described earlier, each BETCase has a single domain generator. This means that all of the test methods defined in a single BETCase must use the same parameters. In a way this is more limited than the Iowa approach. In that approach, the test values for the sets of test method parameters must all be defined by a common set of type finitizations, but each test method may have a different subset, i.e. the test methods do not have to have the same parameters. This limitation could be removed in BetUnit by allowing the definition of multiple Domain variables in a BETCase, which was included in our prototype.

In the Account Break example, BETUnit was successfully applied using the Permutation Generator BET pattern and the Synchronized Sequence Oracle pattern. So far, only a small number of domain generators have been implemented. A solid library of complex and primitive domains will be needed. We are now doing this, and applying our test approach to a wide variety of problems. The version of BETUnit described in this paper is based on JUnit 3.8. This provided a very flexible tool. We are now exploring the use of JUnit 4 with BETUnit to see if its advantages are maintained in this context. In addition, we are examining alternative BET strategies using frameworks other than JUnit.

ACKNOWLEDGEMENTS

The authors would like to thank Nathan Farrington for his help in the development of BETUnit.

REFERENCES

Boyapati, C., Khurshid, S., Marinov, D., 2002. Korat: Automated Testing Based on Java Predicates, In *ISSTA*, IEEE Press.

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.

Howden, W.E., 2005. Software Test Selection Patterns and Elusive Bugs, In *Proceedings COMPSAC 2005*, IEEE Press.

Howden, W.E., 1987. pp. 112, 114. *Functional Program Testing and Analysis*, McGraw Hill.

Jha. A., Kaner, C., 2003. Bugs in the brave new unwired world. In *Pacific Northwest Software Quality Conference.*

.