ELUSIVE BUGS, BOUNDED EXHAUSTIVE TESTING AND INCOMPLETE ORACLES

W. E. Howden
CSE, University of California at San Diego, La Jolla, CA, 92093, U.S.A.

Keywords: Testing, elusive, bugs, oracles, necessary, sufficient, incomplete, BET, JUnit, Elusive Bug Hypothesis.

Abstract: Elusive bugs involve combinations of conditions that may not fit into any informal or intuitive testing scheme. One way to attack them is with Bounded Exhaustive Testing, in which all combinations of inputs for a bounded version of an application are tested. Studies of BET effectiveness for known bugs indicate that it is a promising approach. Because of the numbers of tests that are involved, BET normally depends on automated test generation and execution. This in turn requires the use of an automated oracle. In some cases the construction of a complete automated oracle would require the development of a second version of the application. This may be avoidable if incomplete oracles are used. Two classes of incomplete oracles are identified: necessity and sufficiency oracles. Examples are given of experiments using a necessity and a sufficiency oracle.

1 ELUSIVE BUGS

In black box testing, a program is tested over all its functions and subfunctions. In (Howden, 1980) it was observed that bugs are often closely associated with semantically meaningful "functions" at some level of the implementation, varying from the statement level to the system application level. If all of these functions are tested using standard black box tests a defect appears to have a good chance of being detected.

However, such "broad-based" functional testing can fail to find bugs that occur due to the effects of combinations of conditions. In standard black box functional testing, it is suggested that tests include "functionally relevant" input data combinations. But sooner or later, it seems that a certain "elusive bug" combination takes its toll. In some cases, the combinations are functionally meaningful but belong to a population of combinations that is too large for all of them to be tested. In others, the combinations are coincidental and attain a meaning only after the fact because they are associated with a failure.

Even in the case where a combination of conditions is coincidental, the following *Elusive Bug Hypothesis* may hold:

i) the conditions in a combination are always functionally meaningful even when the combination is not,

ii) for a combination that is associated with failure, the failure occurs whenever that combination occurs.

The consequences of these two observations are that it is possible to identify the conditions that cause elusive bugs before they occur, and that relatively simple sets of tests such as those generated by BET have a good chance of being able to detect them.

2 BET (BOUNDED EXHAUSTIVE TESTING)

It has been observed that defects that cause system failures will often cause a failure in a small or "bounded" version of the application. This BET assumption lies behind a number of testing techniques. For example, in all of the early work in symbolic testing (e.g. Howden, 1977) the test tools choose a bounded set of paths through a program for evaluation. A standard approach was to limit testing to paths that traversed loops at most once or twice.

The bounded testing approach was systematically applied to unit testing of classes in Java in (Cheon, 2002). In this case, a tool was constructed that used seed data to construct all method input combinations. An alternative approach, with similar goals but with more complex

E. Howden W. (2008).
ELUSIVE BUGS, BOUNDED EXHAUSTIVE TESTING AND INCOMPLETE ORACLES.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 115-121
DOI: 10.5220/0001868501150121

test generation capabilities, and with built-in efficiency mechanisms, was described in (Boyapati, 2002). The origin of the phrase BET appears to be in (Sullivan, 2004).

BET is not only associated with exhaustive testing over a limited version of an application, but with automatic testing. Even with the limitation to a bounded version of an application, there are usually many possible combinations of input data. It is necessary to both generate tests and to evaluate behaviour with an automatic tool. For many applications, there is no simple way to automatically validate test output. The problem was discussed in (Weyuker, 1982) where it was suggested that the best course may be to construct a second program, possibly simplified and less efficient. This approach was followed, for example, in (Memon, 2005). Another solution may be to use an incomplete oracle.

3 INCOMPLETE ORACLES

A test *oracle* is a means of determining if the test output or behaviour of a program is correct. This term appears to have been introduced in (Howden W.E., 1978).

The use of an automated oracle is critical to the success of BET. In cases where a completely automated oracle is not possible, the use of an "incomplete" oracle may be necessary. A *complete oracle* C is a means of evaluating program behaviour Y, such that for input X, C(X,Y) = Valid if and only if the behaviour Y is correct for input X and C(X,Y) = Invalid iff the behaviour is incorrect. An *incomplete oracle* is based on relationships that are either necessary or sufficient but not both.

3.1 Necessity Oracles

Necessity oracles can detect if behaviour is incorrect, but cannot, in general, tell if behaviour is correct. More formally, suppose that, for program P, N(X,Y) is an oracle with the following property. If N(X,Y) = Invalid, then the output Y produced by program P for input X is invalid. Otherwise, if N(X,Y) = Unknown, the output Y can be either valid or invalid. A necessity oracle may be associated with a base input/output relationship $N_r(X,Y)$ which has the property that $N_r(X,Y)$ = False if and only if N(X,Y) = Invalid.

The value ranges example described in (Richardson, 1994) is an example of a necessity oracle. With this kind of oracle it is possible to determine if certain necessary properties of correct output have occurred, even if it is not possible to determine if the output is itself correct. Another common kind of necessity oracle is a *robustness oracle*. Such oracles simply look for system crashes or unexpected exceptions (Miller, 2000). These are necessary but not sufficient properties for correct behaviour. *Structural necessity oracles* examine necessary relationships between properties of the structure of the input and output, such as data structure sizes. If a file is incorrectly empty, or not the right length, then a behaviour violation has been observed.

For some applications, automated test data generators can be written so that they produce *test case meta-data*. An associated oracle may compare test input metadata properties directly with output properties. For example, suppose that a program is supposed to merge two sorted files *f* and *g* to produce a sorted file *h*. The test data generator may return the file lengths *lengthf* and *lengthg* along with the test files that it generates. A structural necessity oracle could then evaluate the relationship *lengthh* = *lengthf* + *lengthg*, where *lengthh* is the length of the output file *h*.

3.2 Sufficiency Oracles

Sufficiency oracles can determine if behaviour is correct some of the time, but not all of the time. More formally, suppose that S(X,Y) is an oracle with the following properties. If S(X,Y) returns Valid, then Y is the correct output/behaviour for input X. Otherwise, if S(X,Y) returns Unknown, then the output could be valid or invalid. S(X,Y) may be associated with a base relationship $S_r(X,Y)$ such that S(X,Y) = Valid iff $S_r(X,Y)$ = True. S(X,Y) returns Unknown for $S_r(X,Y)$ = false.

In the case of the above file merge program, we could construct the following simple sufficiency oracle. Suppose that *f* and *g* are the input files and *h* is the output file. Define S as follows:

S((*f*,*g*),*h*) = Valid iff
((empty(*f*) and empty(*g*) and empty(*h*)) or
(empty(*f*) and *h* = *g*) or
(empty(*g*) and *h* = *f*)

In our research project, the concept of a sufficiency oracle was developed to deal with the testing of stateful interactive systems, which are discussed further below.

3.3 Compound and Hybrid Oracles

Oracles can be joined together to make new oracles. Assume that for a necessity oracle N(X,Y) there is an underlying base relationship $N_r(X,Y)$ such that

$N(X,Y)$ = Invalid iff $N_r(X,Y)$ = False.

Alternatively, for a sufficiency oracle S(X,Y) assume there is a sufficiency relationship $S_r(X,Y)$ such that:

$S(X,Y)$ = Valid iff $S_r(X,Y)$ = True.

An oracle Q(X,Y) is *more general* than an oracle M(X,Y) if the set of pairs (X,Y) for which Q(X,Y) is defined (i.e. does not give the value Unknown) includes the set of pairs for which M(X,Y) is defined. M(X,Y) and Q(X,Y) are *equally general* if the set of pairs for which they are defined is the same.

Suppose that M(X,Y) and N(X,Y) are necessity oracles, with base relationships $M_r(X,Y)$ and $N_r(X,Y)$. Then we can use conjunction to form a new necessity oracle Q(X,Y) with base relationship:

$Q_r(X,Y) = M_r(X,Y)$ or $N_r(X,Y)$.

The following simple observations can be made.

Observation: suppose that we construct the conjunction $Q_r(X,Y)$ of the base relationships $M_r(X,Y)$ and $N_r(X,Y)$ for two necessity oracles. Then the oracle based on $Q_r(X,Y)$ will be at least as general as each of the two contributing oracles. If one contributing relationship does not imply the other then the oracle based on $Q_r(X,Y)$ will be more general than the contributing oracles.

Observation: suppose that we construct the disjunction $Q_r(X,Y)$ of the base relationships $R_r(X,Y)$ and $S_r(X,Y)$ for two sufficiency oracles R(X,Y) and S(X,Y). Then the oracle based on $Q_r(X,Y)$ will be at least as general as each of the two contributing oracles. If one contributing relationship does not imply the other then the oracle based on $Q_r(X,Y)$ will be more general than the contributing oracles.

The file merger example from Section 3.3 contains examples of sufficiency oracle disjunction. Define $R_r(X,Y)$ and $S_r(X,Y)$ as follows:

i) $R_r(f,g),h)$ = True iff empty(f) and $h = g$
ii) $S_r(f,g),h)$ = True iff empty(g) and $h = f$

$R_r()$ and $S_r()$ can individually be used to construct sufficiency oracles that return Valid when one associated file is empty and the output consists of the other file. The disjunction ($R_r()$ or $S_r()$) can be used to construct an oracle that is more general, i.e. determines valid behaviour for a wider range of inputs than the individual relationship oracles.

The above observations indicate that if we have two necessity oracles we can get another oracle that is at least as general as each individual oracle by taking the conjunction of their base relationships. If we have two sufficiency oracles we can construct an oracle that is at least as general by taking the disjunction of the base relationships.

Suppose that we have a necessity oracle relationship $N_r(X,Y)$ and a sufficiency oracle relationship $S_r(X,Y)$. Then we can combine them in the *hybrid oracle* Q(X,Y) as follows:

if ($S_r(X,Y)$ = True) return Valid
else if ($N_r(X,Y)$ = False) return Invalid
else return Unknown

This construct will be referred to as a *Valid-first hybrid*. In the analysis of hybrid oracles, the issue of consistency becomes relevant. A necessity relationship $N_r(X,Y)$ and a sufficiency relationship $S_r(X,Y)$ are *consistent* if for all (X,Y):

i) $S_r(X,Y)$ = Valid implies $N_r(X,Y)$ = Undefined, and
ii) $N_r(X,Y)$ = Invalid implies ($S_r(X,Y)$ = Undefined

Hybrid construction is understood to be carried out using consistent contributing relationships. A simple consequence of consistency is the following.

Observation Suppose that $S_r(X,Y)$ and $N_r(X,Y)$ are (consistent) base relationships for a sufficiency oracle S(X,Y) and a necessity oracle N(X,Y). Then $S_r(X,Y)$ implies $N_r(X,Y)$, i.e. the set of pairs (X,Y) for which $S_r(X,Y)$ returns True is contained inside the set of pairs for which $N_r(X,Y)$ returns true.

Observation Suppose that Q(X,Y) is formed using Valid-first hybrid construction from S(X,Y) and N(X,Y). Suppose that S(X,Y) and N(X,Y) are defined (i.e. do not return Undefined) for at least one pair (X,Y). Then Q(X,Y) is more general than either S(X,Y) or N(X,Y).

In joining together a necessity and a sufficiency oracle, we might consider the following *Invalid-first* construct.

if ($N_r(X,Y)$ = False) return Invalid
else if ($S_r(X,Y)$ = True) return Valid
else return Unknown

It seems natural to ask which construct is preferable. First consider the issue of generality.

Observation Suppose that N(X,Y) and S(X,Y) are necessity and sufficiency oracles. Let QV(X,Y) and QI(X,Y) be the hybrid oracles that are formed using Valid-first and Invalid-first hybrid construction. QV(X,Y) and QI(X,Y) are equally general, i.e. they are defined/undefined for the same sets of pairs (X,Y).

The two constructs are equally general, but Valid-first may seem preferable because it first attempts to return a conclusion of validity rather than invalidity and validity is a stronger result. However, as a consequence of base relationship consistency, it is easy to show the following.

Observation Suppose that N(X,Y) and S(X,Y) are necessity and sufficiency oracles. Let QV(X,Y) and QI(X,Y) be the hybrid oracles that are formed using Valid-first and Invalid-first hybrid construction. Then QV(X,Y) and QI(X,Y) return the same results for all pairs (X,Y).

In the case where two base relationships $N_r()$ (necessary) and $S_r()$ (sufficiency) are logically equivalent, i.e. $N_r()$ = true iff $S_r()$= True, then the oracle based on the relationships is a *complete hybrid oracle*. Otherwise we have an *incomplete hybrid oracle*, sometimes telling us when the program is correct, sometimes telling us when it is incorrect, and otherwise returning that validity or invalidity is unknown.

More complex compound oracles can easily be imagined, using a kind of *oracle algebra*.

A hybrid oracle might occur in situations where we can easily determine the correct behaviour for some kinds of inputs but only necessity for others. Consider once again the merge file example. We described a necessity oracle N((*f*,*g*),*h*) which returns Invalid when the lengths of *f*, *g* and *h* do not match. We also described a sufficiency oracle S((*f*,*g*),*h*) that can determine correct behaviour in the special cases where one or both of the input files is empty. These two can be combined in the following hybrid oracle:

if S((*f*,*g*),*h*) return Valid
else if N((*f*,*g*),*h*) return Invalid
else return Unknown.

The following section contains more detailed examples of a necessity and a sufficiency oracle.

4 NECESSITY ORACLE EXAMPLE

The program in this example takes input from a transactions file. The records in this file are either financial or non-financial. Each transaction contains a key field that identifies an account number. The records are sorted by this key. Each record is either non-financial or financial. Financial records contain a transaction amount. The program is supposed to prepare an output item for each non-financial record. In addition, the program is supposed to sum all the financial records for each account, which is then output to a financial data stream.

4.1 Account Break Bug

An account break occurs in an input record stream when there is a change in the account number. The program recognizes the occurrence of a break by keeping track of the current account number and detecting when this changes. The program has two separate transaction processing flows: one for financial data, and one for non-financial data. Along the non-financial flow, the program incorrectly fails to check for account breaks. Hence, if an account record group containing at least one financial record ends with a nonfinancial record, then the subtotal of its financial records is not finalized and output, but is used as part of the total for the following account group.

This is an elusive bug in the following sense. Each of the separate conditions involved in the failure are semantically meaningful, but the combination seems arbitrary and it is not obvious that it would normally be tested for.

4.2 Automated Test Data Generation and Validation Oracle

A sufficiency test oracle for this application could be built from a small set of tests that were constructed by hand. But if we wanted to test over larger amounts of data, hoping to catch elusive untested bugs, we would need both an automated test data generator and an automated oracle.

Both complete and incomplete oracles were investigated. A test data generator was built using the BET testing variation of JUnit called BETUnit (Howden, 2007). BETUnit facilitates the testing of all combinations of input specified by one or more BET-restricted domain generator classes. The test data generator for this example was designed so that it returned both test cases and test case meta-data. Recall from the discussion above that test case meta-data describes properties of a test case, which can be determined by the generator while it is generating a test case. Our test generator generated all sequences of account groups having 0 to 3 groups, where a group can have from 1 to 3 records. The primitive data type fields in each record could have any value in a small subset of possible values. Records could be either financial or non-financial.

The test data generator was constructed to return the following metadata: the number of account groups, the number of records in each account group, the total number of records, the total number of financial and non-financial records, and the total number of account groups with at least one financial record.

There is a strong motivation to try and construct a complete oracle. We investigated the approach in which a simpler, second version of the program was constructed to act as an oracle. In order to avoid the danger of constructing a second program that had the same defects as the first program, we used metadata to build a program with "synchronous" as opposed to "asynchronous" loop control. In the synchronous approach, the number of times a loop is supposed to iterate is known in advance. In the asynchronous approach, the number of iterations is not known in advance, but is determined by a condition that arises during iteration. In this example, the (asynchronous) application program detects when the last record in an account group has occurred by checking for a change in the account numbers. In the synchronous oracle program, the first kind of metadata listed above (the number of account groups and the number of records in each account group), was used to determine in advance how many times the account group processing loop was to be iterated.

We also investigated the use of a necessity oracle. In this case, a structural necessity oracle was designed that used the second class of metadata described above. A test runner was designed to examine the output generated by the program and to determine the total number *numFinanReports* of financial report outputs, and the total number *numNonFinanReports* of financial report outputs. It compares these with the test metadata *numFinanGroups*, the number of groups with a financial record, and *numNonFinanRecords*, the number of non-financial records. More specifically, the oracle component of the test runner checks to see, for each completed test, if:

numFinanReports = *numFinanGroups* and
numNonFinanReports = *numNonFinanRecords*.

In this case the defect is easily discovered, leading to the conclusion that for examples like this necessity oracles are simpler, more effective, and less costly than 2-version oracle programming.

5 SUFFICIENCY ORACLE EXAMPLE

The sample program in this case is a very simple dating system, constructed for the purposes of experimentation with different testing methods. The system has a GUI screen that allows users to log on, ask for dates, and set their personal properties. An administrator can add or delete dating members from the system.

5.1 Delete Non-existent Member Bug

The program contained several "natural" (i.e. non-seeded) bugs. In one, if the administrator attempts to delete a member from the system who does not exist, then an unexpected screen appears instead of the expected "no such member" screen. Furthermore, this bug does not occur if, in the current session, the administrator has previously deleted a user who is a member of the system. The bug is elusive in that it only shows up under certain specific kinds of combinations. Each element of the combination is semantically meaningful in its own right, and would probably be tested for, but the combination seems somewhat arbitrary and not corresponding to an expected kind of test.

5.2 Automated Test Data Generation and Validation Oracle

A model-based strategy was used to build an automated test data generator and oracle, both based on a *test model* M. Each state in the test model M corresponds to a screen image in which the user can enter input and/or initiate a transition to the next screen. Model states are associated with domain generators that generate values that can be input in that state. The transitions from a state S may lead to one or more next states. Tests, whose length is less than a predetermined BET-restricted path length k, are generated by traversing paths in the model M.

If there are two or more next states after a state S, then the transitions are labelled with *transition guards*. A transition guard is a function of the state of the program at S. In a conventional specifications model, the guards g on the transitions from a state S to next states are expected to be mutually exclusive and complete. When they are interpreted as sufficiency oracles, they need only be mutually exclusive in the sense that not more than one evaluates to True.

Suppose that the program P under test is executed corresponding to a path p in the model, correctly reaching a state S, and that W is a next state that can be reached from S along a transition with guard g. If the guard g evaluates to True then W is the correct next state. If no guard on a transition from S evaluates to True, then the next correct state for this execution is not known. If guard g on the transition to W evaluates to True, but the program is observed to transition to a state other than W, then the program's behaviour is incorrect. Note that there may be other paths from the system's initial state that correctly arrive at state S for which the transition to W is correct, but for which the

guard evaluates to Unknown, since the guard is only sufficient.

The reason why we might need to settle for guards that are sufficient but not necessary is the potential difficulty of computing the guard g. In general, g will be a function of both the input entered at S, and the inputs entered on the subpath p that led up to S during an execution of P. The steps on the path p leading up to S provide the *context* for the computation step that occurs at S. If that step is *context free*, then the guard will only depend on the input entered at S. If the step is not context free g will have to evaluate the effects of the inputs along the path p leading to S. It cannot do this by simply observing the state of P at S, since then we would be using information about the state of P in order to validate the behaviour of P, which is circular reasoning. There are several possible approaches to this problem.

For the guards in our test model M, we constructed simple path pattern recognition routines that could be used to define simple sufficiency guards. These routines are capable of examining a path p leading to a state S to see if a sufficient pattern of steps has occurred.

For example, the state deleteMember in the model M has a transition to a next state noSuchMember. Assume that, during testing, the program always starts out with an empty membership data base. Examples of possible sufficiency guards that could be used on the transition include the following path patterns: (no addMember), or (for each addMember(x) there is a deleteMember(x)). If an execution path p arrives at S = deleteMember, and one of these simple guards is satisfied for p and the data entered at S, then we know that for that execution the program should transition to the model state noSuchMember. The transition to a different next state constitutes incorrect behaviour. There are more complex paths to S that do not satisfy these simple patterns, for which the correct next state is also noSuchMember, and along which the program also fails. For these tests we would not be able to determine failure using the guard, since the guard for the transition to noSuchMember would return Unknown, and not True. But all we need is one path for which the guard evaluates to True.

An alternative to the use of guard path patterns is the "second program" approach. An oracle program would be used to determine the correct program states and transitions. This approach may be both feasible and necessary for some applications, but it seems that the path pattern approach will often be simpler.

There are several ways to use test models. One involves a test harness that traverses paths in the model, keeping track of path coverage, while simultaneously executing the program under test on the input derived from the domain generators in the model states occurring along a followed path. As it follows a path, the traverser evaluates program behaviour and checks guards to see if an observed transition to an observed screen is valid.

The second way of using a test model is to have a separate test specifications generator that traverses the model, generating model paths that are then handed off to a test harness that works with one path at a time. In our testing tool we used this approach. The model traverser generates a test specification in the form of a FIT test table, which is handed off to a FIT test runner (Mugridge, 2005). This was done for several reasons. One was to make use of an already existing FIT tool. The other was related to the desirability of the basic approach to systems testing that is followed in FIT. FIT uses *test fixture* classes to map from easily readable FIT test tables to the underlying application code that has to be executed for the specified steps in the table. We used similar test fixtures to map from the test model to the steps that should appear in a FIT test table. A more detailed description of this use of the FIT testing strategy is contained in (Barzin, 2008).

6 CONCLUSIONS

The necessity/sufficiency framework, introduced in this paper, was found to be useful way to characterize both individual oracles and the construction of more general oracles from less general components. It has been used to analyze both hybrid and non-hybrid oracles, such as the two examples included here. The elusive bug hypothesis (EBH), also introduced here, is consistent with our analysis of elusive defects and establishes a basis for further research into the elusive bug problem. The BET hypothesis seems intuitively reasonable on its own, but it is the EBH that may be one of the principal reasons why BET is effective: limited tests are adequate because they generally contain instances of elusive bug combinations, and these combinations cause a failure whenever they appear. In the case of interactive programs, the combinations of conditions causing a failure occur on short paths, which are associated with simple sufficiency guards.

The use of incomplete necessity and sufficiency oracles in automated testing can be justified *per se* because they will be least as effective as robustness testing, which uses a kind of minimal necessity oracle. The addition of more general sufficiency

oracles, or wider-based necessity oracles, can only improve the effectiveness of automated testing beyond the default robustness level.

Continuing research is investigating the foundations for Elusive Bug and BET automated testing, the algebra of incomplete oracles, and the application of these ideas to additional classes of programs.

REFERENCES

Barzin, R., Fukushima, S., Howden, W., Sharifi, S., SuperFIT Combinational Elusive Bug Detection, *Proceedings, COMPSAC 2008,* IEEE, 2008.

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

Cheon, Y., Leavens, G. 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, 2002.

Howden, William E., Functional Program Testing, *IEEE-TSE*, 6-2, 1980.

Howden, W.E. Introduction to the Theory of Testing, in *Software Testing and Validation Techniques,* E. Miller and William E. Howden, IEEE, 1978.

Howden, W.E., Rhyne, C., Test Frameworks for Elusive Bug Testing, *Proceedings ICSOFT 07, Barcelona,* 2007.

Howden, William E., Symbolic Evaluation and the DISSECT Symbolic Evaluation System, *IEEE TSE,* SE-3, 4, July, 1977.

Memon A., Banerjee I., A. Ngarajan, A., What Test Oracles Should I use for Effective GUI Testing? *IEEE TSE*, 31-10, Oct 2005.

Miller, B., Forrester J.E., and Miller, B.P., An empirical study of the robustness of Windows NT applications using random testing, *Proc. 4th Usenix Windows System Symposium*, 2000.

Mugridge, R., Cunningham W., *FIT for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005.

Richardson, D.J., TAOS: Testing with analysis and oracle support, *ISSTA: Proceedings of the International Symposium on Software Testing and Analysis*, ACM, 1994.

Sullivan, K, J., Yang, J., Coppit, D., Khurshid, S., Jackson, D., Software Assurance by Bounded Exhaustive Testing, *Proc. ISSTA*, 2004.

Weyuker, E.J. On testing non-testable programs, *The Computing Journal*, 25-4, 1982.