Adequacy of Bounded Exhaustive Testing and Incomplete Oracles for Elusive Bug Detection

William E. Howden
CSE, UCSD, La Jolla, CA, 92093

Abstract

Elusive bugs (EB) are not reliably discovered using standard methods such as black and white box testing. Bounded exhaustive testing (BET) is a promising approach for their detection. A generic EB fault model is introduced which is used to consider the adequacy of BET for EB detection. BET requires the use of an automatic oracle. Situations in which automation may only be practical with the use of an incomplete oracle are considered. Failure models are used to evaluate effectiveness of incomplete oracles.

1. Introduction

An elusive bug is one that depends on a combination of conditions that may occur infrequently, and are unlikely to have been tested for. The combination of conditions may not have any corresponding application functionality. In BET, a "bounded" version of the system is tested over all inputs. This paper explores the use of a Bounded Exhaustive testing approach to elusive bug detection.

In order to carry out a set of BET tests, it is necessary to have an automated oracle that can evaluate the validity of the behavior or output for a test. A "complete" oracle can determine if an output is valid. An "incomplete oracle" is weaker. It may only be able to evaluate if the output is valid under certain circumstances, or if it posses certain necessary properties. It may be necessary because it is not possible to construct a complete automated oracle. The effectiveness of incomplete oracles is analyzed using failure models.

2. Sample programs

In the following discussion three sample bugs are used, one from each of three sample programs. The first two are data processing programs. The third is a simple interactive program. All of the bugs were naturally occurring.

BET test generation tools and incomplete automated oracles were built that were applied to each of the examples. This not only served as proof of concept, but helped to clarify a number of foundational issues.

2.1. General ledger accounting

The first data processing program reads in a file of records that are sorted by account number, causing them to appear in account groups. Records can be financial or non-financial. If financial, they have a transaction amount. For each group the program prepares a final financial amounts total, which it outputs. For each non-financial record, the program outputs a report. The program fails to check for a change in account numbers when the last record in an account group is a non-financial record. Consequently, it adds the financial record total for that group (if the group has at least one financial record) on to the total for the next group.

2.2. Field validation

This program reads in a file of transaction records having 6 fields, which the program is supposed to check for validity (correct range, type, etc.). If a record has one or more bad fields, this information is supposed to be reported in a printed report. The report for a record has a header line for the record, plus one line for each invalid field. The report lines for a record cannot be split between pages, each having 50 lines. Under certain circumstances, the program fails to properly ensure that the output lines for a group are not split between pages.

2.3. Dating System

This was a simple dating system program that was written to experiment with different testing methods. The user starts the system, resulting in a start/end screen. If start is chosen a logon screen appears, and the user types in a name in a logon box. If a special name is typed, the user is assumed to be an administrator. This results in a screen that allows a choice between adding a new member or deleting an old one. After performing such an action, a result message is printed and the system returns to the start/end screen. If the user is not the administrator and is a member of the system (i.e. has an entry in the data base) an options screen appears that allows a choice between asking for a date or (re)setting personal data. If a date is requested, the user is presented with a form for entering preferences. The system then tries to find a match, and prints an appropriate reply message. After

looking for a date or setting user data, the system returns to the start/end screen. Choosing end terminates a session.

3. Elusive bug fault models and the effectiveness of BET

3.1. Elusive bugs

As described in the introduction, an elusive bug is associated with an unusual or unexpected combination of "conditions". This basic fault model is elaborated by further characterizing such bugs as having the following properties:

i) *Characteristic combinations*. The bug is associated with a characteristic set of combinations of conditions.

ii) I*nvariant invalidity*. An elusive bug results in invalid behavior whenever a combination in its characteristic set of combinations occurs.

iii) *Condition relevance*. The individual conditions in the combinations that cause failures are oriented towards program specifications rather than implementation, and are relevant to the discussion of a program's expected behavior.

iv) *Coincidental combinations*. The combinations that cause a program failure may or may not have any expected functional relevance.

Conditions may be associated directly with properties and relationships in input data, or with states that may arise during application execution. In all cases they are specification oriented in the sense that they are associated with knowledge of how the program is supposed to behave rather than how it is designed or implemented. The use of conditions as the basis of testing is a very old idea, appearing in, for example, [1] and [2].

In the sections below, conditions are identified for two classes of applications: stream based data processing programs and interactive data base systems.

3.2. BET

The classical approach to testing for condition combinations is to consider all "functionally meaningful" combinations [e.g. 1]. Techniques for systematically considering combinations include cause-effect graphs, introduced in [3]. Later work includes ways for indirectly describing combinations and tools for generating them, such as [4].

In the case of elusive bugs, we assume that the conditions are functionally meaningful, and hence discoverable, but that the combination may not be. Consequently, techniques like cause-effect graphing are not directly useful. Alternatively, Bounded Exhaustive Testing considers all combinations of inputs for a "bounded version of an application" which will allow those odd combinations to "emerge" and be tested for.

The idea that bugs can be reliably detected by exhaustively testing over a limited version of a program has occurred in different forms e.g. [5], [6]. Modern approaches that use BET for class testing are described in [7] and [8]. The term BET appears to have achieved common usage starting with [9].

In the following sections, the effectiveness of BET for detecting elusive bugs is considered by examining its use for two classes of programs. The success of BET in "inducing" the combinations needed to detect elusive bugs is analyzed, including refinements to the basic method.

4. Stream based data processing

4.1. Typical conditions

Stream processing programs have one or more input streams and one or more output streams. Only simple computations are involved, wherein input data is combined to produce an item in an output stream. The items in the stream are normally transactions with one or more fields. The transactions may occur in groups, which are either explicitly or implicitly identified in the streams. Streams may be files, output traces to data bases, or items sent to a printer.

Possible conditions include:

i) properties of data items in record fields

ii) categories of records

ii) relationships between records or data items

iii) location in a stream such as first or last item

iv) properties of groups of records in a stream

v) relationships between groups in streams

4.2. General ledger accounting example

In this example, we can identify the following conditions.

i) record is financial or non-financial

ii) account amount is a mathematically special value such as zero or one

iii) record is first or last in the stream

iv) account number in one record is equal to or less than account number in the next

v) account group has at least one financial record, or not

vi) account group is first or last in the stream.

The EB fault model for this example would include all combinations of these conditions. The question for BET is whether it would be effective in generating tests that would cover this fault model.

In order to examine the effectiveness of BET, we will first consider the types of tests we would generate. The first step is to identify the finite value domains that will

be used for the record entries. One field indicates if the record is financial or non-financial. Another has the account number, for which we might choose 3 representative numbers. In the case of a non-financial record, we can construct a small sample of entries. In the case of a financial record, we can construct representative account transaction amounts, including the special zero value which may result in alternative behavior.

The second step is to consider the lengths of the streams. In this case we will consider streams with from 1 to 3 account groups, and account groups with from 1 to 3 records. This will allow the generation of streams with the following conditions: groups and records will be in or not be in the special stream locations (first and last).

The third step is to construct a test generator that will generate all possible input sequences of the specified lengths form the finite data domain subsets. This step will have to incorporate the restriction that input streams be sorted by account number. Our experiments used the BET variation of JUnit which was previously developed and which is described in [10].

The BET generated data sets will cover all the EB combinations and is a natural fit. EB condition combinations that do not occur directly in the BET generation process will occur indirectly as the BET combinations are generated. For example, the EB fault model combination in which a record group with at least one nonfinancial record occurs as the first element in the stream will occur when the generation of all combinations of lower level items is carried out. Other combinations, and in particular the combination in which an account group which is not last, has a nonfinancial record as its last record, and is followed by an account group that ends with a financial record, will also occur. This combination is an example of the characteristic combinations set for the "failure to check for account break for nonfinancial records" defect, included in the description of this example in the introduction.

4.3. Field validation example

In this example, the following conditions are readily identified:

i) valid and invalid fields for each field

ii) output for current record will or will not fit on current output report page.

The second condition is different from the conditions in the previous example. In that case they are all defined as input stream properties. In this case the condition relates input to output stream properties. Alternatively, it is a property of the intermediate state of the program during processing. It is, however, still a property that would occur as part of the specifications.

The consideration of all combinations of different records with different combinations of good or bad fields will give us a complete EB fault model. In order to apply BET, we need to choose finite data domains for record fields that cover both good and bad fields and allow both to occur. We also need to consider the lengths of the input streams. In the previous example, bounding input stream length was easy - just choose lengths up to 3. In this example, BET would not reveal the defect if we followed this simple bounding technique, and we have to introduce a modification to BET that allows "state initialization".

The size of a report page is 50 lines. In order to test over the combinations that cause page ejects, we would have to have all possible input streams up to 50 or 60 records which would be too many, even for automated testing. An alternative is to do the following:

i) identify the values of an application abstract state variable that will cause the condition to occur

ii) construct initial streams for each of these values

iii) combine all inputs with all initializing streams.

In this case, we need to consider streams at the end of which the current output page has from 45 to 50 lines. This will cover both the page turning condition and non-page turning condition. We then combine these partial input streams with an input record with all possible combinations of valid and invalid conditions for its field values.

BET generated test data, with the state initialization modification, will cover the EB fault model for this application and will include tests that cause the report formatting problem to occur.

5. Interactive applications and the Dating System example

In this kind of application the user enters data on a screen, presses some kind of Enter button which causes processing to occur, resulting in new screen. The user will start a session, perform a sequence of interactions and then terminate the session. Typically, the associated program has a data base and performs relatively simple computing, often limited to comparisons. The Dating System example described in the introduction is a very simple instance of this kind of program.

Typical conditions for this kind of application involve properties of data entered by the user, or relationships between data and the data base.

In the dating system example the following conditions are readily identified:

i) a member(x) is or is not in the system

ii) a member(x) is the administrator

iii) input data item left blank or not blank

iv) date preferences p have/do not have a match in the data base *db*

v) first and last interactions in a session.

The application of BET involves the generation of possible interaction sequences up to a predetermined

length. For each possible input, a finite set of possibilities would be chosen. If we assume that the data base is initially empty, then the construction of all possibilities up to the chosen path lengths that allow all screens to occur will include all possible combinations of the first iv) of the conditions. Condition v) will be covered if sequences of 1-3 sessions are considered.

The implementation of BET for testing interactive systems can follow a standard model testing approach, with the additional feature that finite domains are defined for the inputs that can occur in each screen/state. One of the major hurdles is how to determine which transition(s) can be followed when there are multiple transitions from a state to the states after it. For simple examples, the transition may correspond to an exact input entered in the previous state, and the system is "Markovian" in the sense that earlier inputs do not affect the choice of a transition. For others, especially those for which the model is abstract, transition conditions or "guards" are needed to disambiguate model non-determinism.

BET testing of interactive systems appears to be effective in that it will cover the EB fault models. The elusive defect described for the DS example will be revealed by any tests in which no member is successfully deleted before an attempt is made to delete a member not currently in the data base.

6. Elusive Bug Fault Models and Incomplete Oracles

The BET approach results in the generation of large numbers of tests, requiring the use of an automated oracle. This may result in the use of incomplete oracles, which do not always give a definitive answer in regards to the validity or invalidity of a program's output. In this section the effectiveness of incomplete oracles is considered for BET generated tests. First, a general framework for the consideration of incomplete oracles is presented.

6.1. Incomplete Oracles

The concept of a *test oracle* appears to have been first introduced in [11]. Additional articles on oracles considered the general problem of their implementation, such as [12], where it was suggested that it may be necessary to construct a second test version of a program in order to validate the behavior of the production version. Other papers discussed the idea that oracles may not be exact, such as in Taos system [13], where an oracle may be restricted to checking if output is in range. Perhaps the least demanding kind of automated oracle occurs in "robustness testing" [e.g. 14] in which a program is run for large numbers of randomly generated tests, and the program is monitored to see if it crashes or returns an unexpected exception. Both the range check and robustness testing examples use *incomplete oracles*.

Two general classes of incomplete oracles will be discussed: necessity and sufficiency. A *necessity oracle* is able to determine if output or behavior has a necessary property for validity. A *sufficiency oracle* is able to determine if output or behavior has a sufficient property for validity. A necessity oracle is incomplete because in the case where a necessity property is satisfied, the validity of the output is unknown or undefined. It is only in the case where it is not satisfied that its validity is known, where it is invalid. A sufficiency oracle is incomplete because in the case where it is not satisfied validity is unknown or undefined.

An oracle Q(X.Y), which determines if output Y is valid for input X, can be thought of as being based on a relationship $Q_r(X,Y)$ which has the following properties. In the case of a necessity oracle, the oracle returns invalid if and only if the relationship evaluates to False. In the case of a sufficiency oracle, it returns Valid if and only if the oracle returns True. In other cases the oracle returns "undefined".

We define one oracle to be *more general* than another if it is defined for a broader range of inputs, (i.e. if the set of inputs over which it is defined contains the set of inputs for which the second oracle is defined). New, more general necessity oracles can be created by taking the intersection of the base relationships for two existing oracles. New, more general, sufficiency oracles can be created by taking the union of the base relationships of two existing sufficiency oracles. It is also possible to obtain increased generality by combining sufficiency and necessity oracles. For example, we may use a sufficiency oracle for the easy cases, and a necessity oracle for the rest.

Robustness testing is used to refer to automated testing efforts in which application behavior evaluation is limited to the detection of crashes and unexpected exceptions. We considered this to be a kind of lower bound for automated necessity oracles. Any oracle which is more general than a robustness oracle is, per se, potentially more effective, motivating our general study of incomplete automated oracles.

As in the discussion of BET adequacy for elusive bug test generation, two general classes of programs are considered, stream based data processing and interactive programs. Failure models are used to analyze the effectiveness of associated incomplete oracles.

7. Stream-based data processing

As described above, programs like these have input and output streams. The computations, in which items in the input stream are transformed into items in the output stream, are often fairly simple. Two general classes of

failures are identified: *item function failures* and *structure failures*. An item function failure results in an incorrectly computed item in an output stream. A structure failure corresponds to failing to produce a necessary item, duplicating an item, or misordering items in an output stream.

An automated incomplete oracle for a stream based application might have both a sufficiency and a necessity aspect. A hand computed set of results, acting as an incomplete, sufficiency oracle, could be used to check the item function computations. A set of consistency invariants, describing structural relationships between inputs and outputs, might implement a structural necessity oracle. The structural oracle could rely on input and output *metadata*. An automated test generator could generate properties of input test streams, such as stream length, along with the test streams. Output properties might be generated along with the output. A *structural necessity* oracle could then be based on the meta data rather than on the actual test input and output.

7.1. General Ledger Accounting

The defect given for this example results in several structural output failures. It fails to include a financial output report for one or more accounts for some classes of input. It also fails to generate a non-financial report for some of the inputs. Suppose that x is the number of nonfinancial records that appear in the input stream. This could be determined by the test generator as meta-data. Let y be the number of records in the non-financial items output report. This could be determined by the oracle or from output metadata produced by the program. Then $x=y$ is a structural necessity oracle. A variety of other necessity conditions can also be established.

For the class of structural failures associated with this example, a straightforward use of a structural necessity oracle will be adequate. The testing of the remaining functionality of the program - computing the correct entries for items in the output streams - could be done with more localized black box testing.

7.2. Field validation

This example is also a stream processing program. The above example involved missing or duplicate stream output steps. In this example, there are different output steps can occur for the same output stream. Possible modes of failed behavior include incorrect interleaving of output steps. In particular, suppose that we consider the output actions in which a field error report line, a record field errors header, and a page boundary output step are performed. If these are out of order, a page straddling field report failure may occur.

The same structural necessity oracles that were used in the general ledger accounting example could be used here. In addition we could add a necessity oracle for detecting output report formatting violations. Suppose that the following metadata is either generated by the program, or constructed from the output stream. The metadata consists of a sequence of tokens *fl*, *hl*, or *pb*, standing for field line, header line and page boundary. The oracle looks for the necessity relationship between successive items: not(*hl* followed by *fl*). If this is violated the program is invalid.

For both this and the above general ledger accounting example, incomplete structural necessity oracles are adequate for the detection of stream based output structure failures. In addition, the remaining functionality that needs to be tested is amenable to standard functional testing methods.

8. Interactive systems

In the approach used for this class of programs, a system specification state model is used both for (BET) test generation and for (incomplete) automated oracle implementation.

8.1. State models and incomplete oracles

In general, state models are often abstract, and only certain unique identifying properties of the states/screens are given. We expect all models to be complete in the sense that any (legal) state that the program can get into must match one of the state descriptions in the model. Guard conditions may appear on the transitions when there are multiple transitions from an abstract state S. Suppose that a partial program execution corresponds to a partial path P from the initial state up to the state S in the state model. The correct transition from S is a function of what the state of the program should be at S, if the path P had been followed, and of the inputs entered at S.

A test runner that is based on a state model traverses paths through the model, generating different possible inputs in each state. When the test runner is in a state S, and there are multiple transitions from S to the next states, then the following is required of the model:

i) if the program is in a state that matches the description for S, then the set of states that follow S in the model should contain the (description of) the next state to which the program should transition.

ii) if the program is in a state S, and the transition guard on a transition to a next state V evaluates to True, then the correct next program state must match the model state description V.

These two properties indicate the way in which a model is used as an oracle. Condition i) is a necessity condition. If the program under test transitions to a state other than one of the next states in the model, then the behavior is invalid. Condition ii) is a sufficiency property. If a guard is satisfied, and the next state of the program does not

match the next state in the model associated with that guard, then the program behavior is invalid. If condition ii) is satisfied, and the next state of the application matches the next state in the state model, then the behavior at that point is valid.

The use of this kind of state model for test generation and oracle validation requires a computational mechanism for computing the guard transitions. One approach that has been suggested is to use a parallel gold standard program that is run along with the application under test. This is the approach used by [6]. This approach has the obvious potential problems: the cost of constructing another program and the possibility that the oracle version of the program will have the same defects as the original program under test.

When the above approaches are not practical, it may be necessary to consider incomplete guards (oracles). This means that the program could be in a state that matches model state S, and none of the guards on the model transitions evaluate to True. Provided that the next state necessity condition is satisfied (i.e. the next state of the program matches the description of one of the next states in the model) then no violation has been detected. In situation such as this, testing and oracle evaluation can continue with the next state, but a defect may have been undetected.

As in the general discussion of incomplete oracles given above, we can increase the generality of our state based oracles by constructing more precise necessity conditions, or more general sufficiency conditions.

8.2. Dating system

There are many kinds of conditions and condition combinations in this example that are fault model relevant, such as deleting a non-existent member, or restarting the system after adding a new member without setting the member's data. All of these will arise during a BET oriented traversal of a test generation state model. BET will also allow the consideration of combinations in which one kind of condition is combined with the absence of another kind.

Experiments were carried out with several kinds of model based automated oracles for the dating system example. One of these involved the use of *path pattern recognition*. A rationale for investigating this approach is that when it is used in conjunction with BET, it is only necessary to consider patterns in paths of limited lengths, so that pattern complexity will be limited. Suppose the user is attempting to delete a non existent member *x*. The transition from S to the model state V that reports the there is no such member *x* could have any of the following sufficiency guards. All are related to the pattern of user actions on paths through the model from the initial state to S.

i) there are no add() states

ii) there are no add(*x*) states/screens, where *x* is a possible member name

iii) every add(*x*) state is followed by a delete(*x*) state without an intervening add(*x*)

iv) every add(*x*) state that is not followed by another add(*x*) state without an intervening delete(*x*) state is followed by a delete(*x*) state.

In the case of oracle guard i), no value is defined for the user who is being added, and the computation is quite easy, but execution paths for which it provides an answer are very limited. Guard ii) is more general in the sense that it will evaluate to True (i.e. will not evaluate to undefined) for a broader class of situations, which will make it possible to evaluate the validity of a broader range of behavior. The other guards provide even more general oracle capabilities.

For the sample bug that was given in the description of the Dating System, a very simple sufficiency oracle is adequate. If every test starts with an empty data base then there will be a path to the state S along which a transition to an incorrect state V occurs instead of to a state W for which sufficiency oracle guard ii) is satisfied, revealing the defect. Actually, in this particular case, the bug would also have been caught by the necessity oracle associated with the set of next states after S in the model, because the program transitions to a state V that is not in that set.

9. Conclusions and Future Work

The goal in this research was to consider the following two questions. The first was "is BET adequate for the testing of elusive bugs, and what are some useful guidelines for its application?" This question was approached by constructing a generic EB fault model for elusive bugs. A more specific fault model for an application is constructed by considering its conditions and their combinations. The more specific model can be used to evaluate the potential effectiveness of BET and to adjust it for the application context. For example, in the case of the field validation program, it was necessary to include a context setting capability in the BET test generator.

The second question was "can adequate automated oracles be built for validating BET generated program behavior?" For a particular application, the idea was to construct the most general automated, possibly incomplete, oracle that could be devised, and to characterize its effectiveness using a failure model. A general framework, based on necessity and sufficiency oracles, was introduced that can be used to systematically describe and analyze the generality of incomplete oracles.

With respect to the first question, the conclusion was positive. The results of the analysis indicated that BET, adjusted to the application context, could be an effective testing method for elusive bugs. In addition, fault models

can be used to both characterize BET efficiency and to tailor BET to a particular application context.

With respect to the second question, the conclusion was positive but not as definitive. Two kinds of programs were analyzed: simple stream processing programs and an interactive system. For stream based data processing programs, the conclusion is positive. The consideration of failure models allowed us to characterize the effectiveness of the incomplete oracles that were used. In this case they were more powerful than simple robustness necessity oracles which would not have been general enough to detect the sample defects.

For the second kind of program, interactive systems with model based oracles, several model-based failure modes were described. The analysis that was presented indicated that transition guard sufficiency oracles were adequate for the class of failures seen in the example, but that simple necessity oracles that were able to determine whether or not the result of a transition was in a set of legal possible resulting states were also adequate. In other words, the extra complication of sufficiency guards was not needed, at least in this case, because the faults resulted in failures that generated invalid next states.

The research literature on testing contains many examples of different methods which are described along with statistics on their effectiveness in the detection of defects. The work described here differs from this in that it attempts to characterize the kinds of defects for which an approach will be effective. It does this first in terms of a specific fault model, derived from the generic EB model, and secondly in terms of an oracle failure model.

Planned future research involves the application of the approach described here to a more extensive set of examples. This could take the form of both more specific EB fault models, and the identification of different kinds of oracle related failure models. In the case of model-based testing, for example, it may be possible to characterize various graph based failure modes that correspond to the situations in which simple classes of incomplete sufficiency guards are adequate for their detection.

10. References

[1] Myers, Glenford, *The Art of Software Testing*, Wiley, 1979/2004.

[2] Richardson, D.J., Clarke, L.A., A partition analysis method to increase program reliability, *Proceedings ICSE*, IEEE, 1981.

[3] Elmendorf, W. R., "Cause-Effect Graphs in Functional Testing", TR_00.2487, IBM, Poughkeepsie, N.Y., Nov. 1973.

[4] Ostrand, T.J., Blacer, M.J., The category-partition method for specifying and generating functional tests. *CACM*, 31-6, June, 1988.

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

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

[7] 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.

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

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

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

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

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

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

[14] 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.