APPLICABILITY OF BET TO ELUSIVE BUGS IN DIVERSE APPLICATION AREAS

M. Chaudhary, B. Chen, P. Desai, R. Hemmatti, F. Lionetti, T. Hachisuka, W. Howden
CSE, UCSD, La Jolla, CA 92075
howden@cse.ucsd.edu

Keywords: Testing, Elusive Bugs, Defects, Bounded Exhaustive Testing, BET, Frameworks, Fault Model, Failure Model, Test Oracle, Inverse Oracle.

Abstract: The basic principles of Bounded Exhaustive Testing (BET) are reviewed, as well as the concept of an Elusive Bug (EB). Initial work on the application of BET to EB's previously indicated that it provides a new and promising approach to this problem. A four-part BET/EB oriented test framework involving: fault model development, BET test generation design, failure model identification and automated oracle design is introduced. The framework provides a systematic approach to BET/EB. It was applied to three very different areas of application. The research indicated the general applicability of BET and the BET/EB framework. It resulted in increased insight into BET/EB including the development of new techniques, such as the BET/EB "inverse oracle". The research illustrated how fault models can be used to put BET application bounding on a systematic basis. It also illustrated how failure models can be used to facilitate the development of automated oracles, and how they can be used, along with fault models, to systematically define the effectiveness scope of a BET testing strategy.

1 INTRODUCTION

1.1 Exhaustive Testing

BET is based on the observation that the set of possible inputs to a program can be classified according to the "size" of the application instance, and that defects will occur for small versions in the same way that they will occur for larger versions. For example, a sort routine can have input arrays of different sizes but many failures are as likely to occur for arrays of small size as easily as they are for larger arrays. This leads to the idea of exhaustively testing over "bounded" versions of an application.

Similar finite testing ideas also occur in model analysis and testing, in which abstraction may be used to create a finite version of a program or system. e.g. (Artho, 2008)

The BET approach was recently popularized as an automated class testing method, e.g. (Sullivan, 2004). More recent work has generalized BET, with more flexible test data generation mechanisms, e.g. (Howden, 2007), and the identification of BET oriented test oracles that may not be "complete" but are more powerful than simple crash detection. (Howden, 2008). In this paper we are specifically concerned with the use of BET as a testing method for attacking the Elusive Bug problem.

1.2 Elusive Bugs

An Elusive Bug has been characterized as one in which its manifestation depends on a combination of conditions (Howden, 2008). Elusive bugs often involve conditions which are individually meaningful, but for which a combination may have no meaning other than the post facto discovery that it causes a failure. Instead of trying to second guess which combinations might be useful, or how to define and select them, the idea in BET is to test them all, within application instance bounds.

1.3 Fault and Failure Models

In order to use BET to systematically test for elusive bugs we need a general framework for its application. It should be general enough to be useful in diverse application areas and at all levels of testing. The generic approach that we followed has four parts to it: fault model identification, BET test generator design, failure model identification and automated oracle design. We investigated this approach to BET in three application areas.

Fault models for software describe software defects, as opposed to *failure models* which describe invalid behavior. Fault models can be categorized as white or black box. *White box fault models*, as in the case of white box testing, are defined directly in terms of program constructs. Mutation testing is based on white box fault models, in which program faults are defined in terms of perturbations of program statements. *Black box fault models,* as in the case of black box testing, define faults indirectly in terms of classes of inputs.

In general, EB-oriented fault models associate faults with combinations of behaviour-affecting conditions. The conditions may originate from black or white box views of the software. One of the purposes of developing fault models is to characterize the defects for which a related testing method is effective. Another, for fault models associated with BET, is to help identify the size of the application instance which BET is going to have to test over to achieve fault model coverage.

Since BET can generate a large number of tests, it is desirable to develop an automated oracle for checking its results. It is not always possible to construct a "complete" oracle. An oracle is complete if it can compute a necessary and sufficient relation for the validity of all observed input-output behavior. An incomplete oracle computes a relation that is necessary or sufficient but not both. The simplest incomplete oracle is a robustness checker, e.g. (Miller, 2006), which determines whether or not a program has crashed or delivered an unexpected exception on a test. Although it may not be possible to develop a complete automated oracle, it is often possible to devise a necessity oracle that will do better than simple robustness checking. The first step in designing an oracle is to develop a failure model which characterizes the class of invalid behavior that can be detected. A framework for defining and developing incomplete oracles and failure models was previously described in (Howden, 2008).

When a BET testing strategy is designed, its effectiveness is circumscribed by the fault and failure model(s) that characterize the kinds of faults it will be able to detect. We note that the four parts of the approach are not always applied in order. The availability of a certain kind of automated oracle may determine the failure model, or the fault model and the BET test generator may be identical.

1.4 Diverse Application Areas

Programs from three application areas were considered: graphics, numerical simulation, and distributed systems. For each of these we describe the application of the four-part BET procedure and consider its applicability.

2 GRAPHICS

One of the more important contemporary graphics applications involves rendering. Rendering provides a visual representation of simulated objects illuminated by simulated lighting. Part of the rendering process involves the determination of light ray intersections with an object surface (Kensler, 2006).

One common way of representing surfaces is to use contiguous triangles. Both the triangle surfaces and a light ray can be considered to exist in a 3d grid, in which points represent vertices and line endpoints. A variety of algorithms have been developed. The testing problem is difficult because there has not been a programmable way of determining if the results of a test are correct, i.e. no automated oracle. Previous practice includes visual examination of the output from a rendering program. A failure to detect an intersection point may show up as a black dot on the screen (Woo, 1996).

The BET approach to this application area led to a way of automatically testing intersection programs using an automated oracle.

2.1 Fault Model

In this application, we considered both white and black box fault models. The first was used to motivate the second.

a) White box. There are three kinds of faults that can occur: computational expressions, precision, and program logic. The first can be covered by testing the computational expressions in isolation. The other two by considering their role in the generation of black box faults.

b) Black box. The program input consists of surfaces and light rays. Round-off errors may result in failures in which an intersection at an exact vertex or surface boundary is missed. If a boundary divides two surfaces, then a ray passing through a common boundary may be missed during calculations made for one surface but not the other, so a correct result could be determined. If the boundary is a surface edge, then the same calculations could produce an incorrect result, indicating that an input based fault model should include examples in which surface boundaries are both internal and external.

Precision faults may also occur when a ray passes very close to a surface without intersecting it. Intersection computations for the case where a ray intersects a surface at its interior may be less likely

to fail due to precision problems, and more likely to fail due to program logic errors that do not show up elsewhere.

These considerations lead to a fault model in which differently directed rays are combined with one or more surfaces. The elusive bug (EB) fault model, which consists of all combinations of relevant behavioral conditions, will include combinations such as a ray passing at a sharp angle through a three-way vertex. This is typical of the kinds of combinations that black box testing may not test for since there is no obvious cause-effect relationship here, and the combination is not semantically meaningful. Yet it is typical of the kind of unexpected combinations that can unexpectedly cause problems.

2.2 BET Test Data Generation

The above fault model can be covered with a BET testing approach having the following properties. First construct a cube containing possible intersection and vertex points that is large enough to include representative surfaces constructed from three contiguous triangles. The cube also includes rays which begin at locations in the cube and are directed at different possible angles. Ray generation can be simplified by considering the object surface triangle to be a subsurface of a larger super-surface. Rays are constructed that connect each possible originating location with a location on the surface. If the surface location is in the object subsurface, the algorithm should determine that an intersection exists. If outside the object subsurface, no object subsurface intersection should be identified.

2.3 Failure Model

In this example, we have a simple failure model. The program fails if it identifies a ray surface intersection when none exists, or fails to identify an intersection when it does exist.

2.4 Automated Oracle

One of the more interesting general concepts that we discovered during the investigation of this example was that of an *inversion oracle*. In this approach, the test generator starts with an output result and then constructs all inputs that should lead to that result. In this way it knows what the output should be for any test. In the surface intersection example, output can be True (intersection) or False (no intersection).

2.5 Evaluation

Previous experience with this problem domain provided examples of incorrect rendering in which an intersection was not observed by the graphics rendering tool. Standard test practice, which was both awkward and unreliable, was to choose random input data and to visually examine the output for problems. Failed intersection detections showed up as a black spot in the display. A sample BET testing system was constructed using the design described above and run on the examples defined by the EB fault model. The use of the inversion oracle made it possible to reliably and automatically detect intersection defects.

3 FINITE ELEMENT SIMULATION

A common method of simulating physical objects is to construct a grid or mesh. Each cell in the grid is associated with an equation that describes the properties of the cell. The cell takes inputs from one or more units surrounding it, and then produces outputs which become inputs to its neighbours. The properties of the corresponding physical entity are simulated over a sequence of steps, at the beginning of which inputs are produced at one or more "input" cells and then at the end outputs measured at one or more "output" cells.

The Continuity simulation program (Bioeng., 2008) can be used to simulate electrophysiology of living structures, and in particular to determine the transmitted effects of a stimulus. For example, a voltage could be supplied to one surface, and the model used to measure the transmitted effects arriving at some other location or surface.

3.1 Fault Model

This application involves a variety of conditions, corresponding to different kinds of inputs that will affect behavior. The EB-oriented fault model is associated with combinations of these conditions.

The Continuity program can accommodate a wide variety of problem applications. For our experiment, we used a generic simple version, in which a voltage is transmitted by "pure diffusion" through a cell, rather than through a means involving more complex equations. The input conditions that could vary were: mesh shape, stimulus location, mesh dimensionality, basis functions, derivative type, solution steps, rendering, and boundary condition constraints.

3.2 BET Test Generation

As in other BET applications, a central concern is the application bounding issue. For some of the conditions the values are binary, so both can be chosen. For others, the possibilities are large or unbounded. Examples of the latter are: model size, number of simulation steps, and location of stimulus.

The mesh size is related to the problem of solution convergence. The idea is to try models with different numbers of mesh elements and observe if important phenomena are observed. If not, then the model is too coarse. The size is increased until that behavior is observed and if the behavior stabilizes then the model size is fixed. In our experiments, a mesh with a stack of four elements was found to be sufficient.

The number of steps is also related to behavior convergence. Once a certain number of simulation cycles have been performed, if the behavior converges then that is considered adequate. In our experiments, between 6 and 10 steps was found sufficient so these two values were chosen for the tests.

Finally, the stimulus could be located in any location in the model. We chose the following set of six representative possibilities: top line, middle line, top left point, top middle point, point at the center of the mesh, and simultaneously at all points throughout the mesh.

The above choices, with a simple diffusion model, resulted in a set of 1536 tests.

3.3 Failure Model

Our failure models included a robustness model. An output value is referred to as NaN (not a number) if it is not numeric. The occurrence of this output indicates failure, since numeric output is a necessary condition for validity.

We were interested in detecting functional failures. Even if they corresponded to incomplete, necessity property, this would be stronger than simple robustness. To do this, we considered different output relationships that must occur for different stimulus conditions and mesh geometry. As an example of this, consider a simple 4-by-4, 2D mesh, with a stimulus that occurs simultaneously across the middle of the mesh. The "output" value at the middle of the top and bottom at the end of the simulation must be the same. These necessary relationships, together with the NaN robustness requirements, constituted our failure model.

3.4 Automated Oracle

Our test oracles paralleled the above failure models: a robustness oracle determined if outputs were NaN, and our (incomplete) functional oracle measured required relationships between output values.

Previous automated oracles for testing this application involved the use of regression tests. The dependency of a relation-based failure model, like the one we used, on a class of inputs requires more initial test planning than the use of a simple regression failure-model/oracle, but it is less fragile and more reliable in the following sense. If a set of regression tests fails, then new tests may have to be established as the gold standard. It may not be clear which set of results are the valid ones. In the case of a necessity relationship, it is always valid.

3.5 Evaluation

For this example, BET tests based on our failure model were run against the Continuity system. They revealed a known existing bug as well as many additional problems. The existing bug was of the EB type, in the sense that it only occurs when one of our fault model suggested combinations is used.

Other defects that were discovered included memory leaks and problems with certain kinds of boundary constraints. They also included a classic elusive bug in which some functionally meaningless combination of input types caused a design and coding defect to surface. BET provided the first systematic rigorous approach to testing that has been available for this application.

4 DISTRIBUTED SYSTEMS

We viewed distributed systems as those in which a set of processes communicates with each other to perform some function. In our investigation of distributed systems, instead of looking at a particular application we considered previous approaches to the testing problem, to see where BET might fit in, and whether or not it could be an improvement.

The first approach (Boy, 2004), which we will refer to as "Random", uses randomized testing. A set of clients is simulated, and for each client a set of random requests is sent to the server. The test driver keeps track of the sequence of calls that are made from the simulated clients, along with the received responses. Failures correspond to invalid sequences. A set of invariants is created which all sequences must satisfy. In the vocabulary of this paper, the test oracle is a necessity oracle.

There are several possible problems with this kind of approach. The first is characteristic of all automated testing methods that depend on random inputs. If there are a large number of possible sequences, random selection may easily miss the

small number of them that cause failures. Also, if the selection is truly random, then arbitrarily long sequences must be included, which will exacerbate this problem. Another problem, not connected to the randomness of test input, is a lack of control over the communications medium. Some faults may involve network transmission race conditions, which cannot be directly explored.

The first problem, random selection, is avoidable with BET, provided the bounding constraint does not exclude a fault. (Boy, 2004) describes a bug that was found by random testing but whose discovery seems to require luck when random testing is used. The same defect would have been found every time using a straightforward application of BET.

The second drawback to Random, the inability to explore network induced failures, can be overcome using a test strategy in which the network is simulated, as in Cesium. In this approach, the application processes run in the same network environment as a test driver. This enables the driver, at each moment of simulation time, to check process queues for required consistency relationships and other properties. It can also manipulate the queues to simulate failures such as lost messages and so on.

A drawback to Cesium is that the tester is still responsible for constructing all test scenarios. There may be combinations of events that could cause a failure that the test designer did not think of. The advantage of Random is that there is always some probability of these occurring, even if it is very small. BET combines the potential inclusiveness of Random, with the guarantee of execution of Cesium.

4.1 Fault Model

We considered faults that correspond to sequences of interactions between the processes in the system. For example, a race condition caused by a long delay in a message making its way through the system corresponds to a sequence of communications in which a request and its receipt are separated by a certain sequence of events that were made possible by that delay, and which causes consequent invalid behavior.

4.2 BET Test Generation

There are two principal issues here: test generation and the bounded application specification.

In a BET-oriented approach, a simulated environment like that used in Cesium, would be employed, but with a generic test driver and network simulator that would implement an all-combinations approach to test construction.

The first bounding consideration is the number of processes that we included in our test sequences. If our fault model is concerned with faults resulting from the simultaneous use of shared resources, then BET will have to generate tests in which up to three processes are involved.

A second consideration is a bound on "amount of time" (i.e. number of simulation cycles), for which a message to another process can be stalled inside the simulated network before delivery. In (Guillarmo, 2000), the authors incorporate lower and upper bound communication delays. These could be incorporated into the fault model. Communications that take longer than this are treated as network failures, and the corresponding messages left undelivered. This aspect of a fault model guides the construction of the corresponding aspect of the BET test generator.

Assuming that we can make the tests long enough to cover race condition message sequences, the length of a test is still an open issue. In the case where there are finite resources such as message queues, a state can be reached in which system behavior is altered due to the exhaustion of this resource. If we can test all combinations of messages that are long enough to cause this situation, then we would cover this aspect of the fault model. But if this only occurs after lengths of message sequences for which the consideration of all combinations is impractical, then it is necessary to have tests which set the system to the necessary intermediate state before the exhaustive all-combinations part of the test generation process begins.

4.3 Failure Model

Generic failure conditions for distributed systems include: failure to send a message and failure to receive a message. Automated oracles need to cover this simple failure model.

Failure models may also be more specific to the application such as the assignment of the same lock to two processes by a lock server. This is the kind of failure described below in the Evaluation Section.

4.4 Automated Oracle

The approach suggested in (Boy, 2004) involves the detection of illegal patterns in sequences of messages and responses. For example, certain requests will require a response, and a missing response failure will show up in the invoking sequence as an illegal pattern. This was the approach we followed.

4.5 Evaluation

The BET approach to this problem was analyzed with respect to the sample defect described in (Boy, 2004). In this example there are 2 or more clients, A and B, and a lock server. The code was apparently written so that if a client does not receive a requested lock after a certain amount of time, then it sends a release() for the lock to the server. Suppose that the grant() message had been issued by the server but not yet received by process A when it sends the release(). The logic written into the server is such that when it gets the release() it assumes the lock is free, so that it can allocate it to a subsequent request() from process B. As soon as A receives the tardy grant() we will have a situation in which both A and B have the same lock.

The random testing experiments described in (Boy 2004) resulted in the detection of this bug. However, a different set of tests, just as likely to be chosen, would have missed it. On the other hand, the more precise approach described in the Cesium paper would have repeatably and reliably found the defect, *provided* the tester had thought to construct such a test. BET would reliably generate a defect-revealing sequence every time.

5 CONCLUSIONS

The results of our investigation were positive in two ways. The four-part framework was an effective, generic approach to the application of BET, in which the essential concerns are identified and separated. In addition, BET was found to be an effective defect detection technique across the wide range of examples that were considered.

Fault models can be used to specifically document the defects for which a set of tests will be effective. They proved to be a convenient way to consider the application-bounding aspect of BET, in order to systematically define minimal bounds for reliable fault detection. In all three application examples, the requirement that the BET test generator "cover" the fault model facilitated the identification of necessary lower bounds on the "size" of the bounded application to be used in test construction.

Failure models were found to be useful in identifying a class of failures that can be observed. Taken together, the fault and failure methods circumscribe the defects for which a BET testing effort is guaranteed to be effective.

Our BET-oriented test procedure led to significant paradigm shifts in the way that two of our applications could be tested. For the graphics application it led to the creation of an inverse oracle. For the finite element simulation, it led to a failure model that was stronger than simple robustness and more consistent than regression testing. Both of these novelties were due to the very essence of BET – bounding the complete problem domain into meaningful subdomains by some central characteristics. By considering these central characteristics, stronger correctness oracles naturally become apparent. This is not likely to occur during simple random testing in which the test domain does not have the sort of structure that lends itself to the discovery of stronger failure models and automated oracles. While this kind of paradigm shift may not be unique to our testing methodology, and may not occur for every application, our experience indicated that our systematic testing framework facilitates this kind of change in thinking.

REFERENCES

Artho, C., Leungwattanakit, W., Hagiya, M., Tanabe, Y., Tools and Techniques for Model Checking Networked Programs. *SNPD*, 2008.

Bioeng. Dept., http://www.continuity.ucsd.edu, UCSD, 2008.

Boy, N., Casper, J., Pacheco, C., Williams, A., "Automated Testing of Distributed Systems, Final project report, MIT 6.824, May 2004.

Guillarmo A. A., and Cristian, F., Simulation-based Testing of Computer Protocols for Dependable Embedded Systems, *The Journal of SuperComputing,* 16(1-2), 2000.

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

Howden, W.E., Elusive Bugs, Exhaustive Testing and Incomplete Oracles, *ICSOFT*, Porto, Portugal, 2008.

Kensler, A.; Shirley, P. "Optimizing Ray-Triangle Intersection via Automated Search". *IEEE Symposium on Interactive Ray Tracing 2006*, Sept., 2006.

Miller, B. P., Cooksey, G, and Moore, F. "An Empirical Study of the Robustness of MacOS Applications Using Random Testing," *Proceedings of the 1st International workshop on Random testing*, 2006.

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

Woo, A., Pearce, A., and Ouellette, M. "It's really not a rendering bug, you see..." *IEEE Computer Graphics & Applications*, 16(5), September 1996.