Error Models and Software Certification

William E. Howden
CSE, UCSD
La Jolla, CA, 92013

Abstract- An error-based approach to certification is described. A classical theory of error is reviewed and a software interpretation of the theory is developed. The interpretation suggests a strategy for testing and analysis. The strategy was evaluated by comparing its potential effectiveness with that of certification standards based on individual methods.

Keywords- testing; analysis; error model; expertise; method; certification.

I. INTRODUCTION

The root causes of failures are viewed as errors that are made during one or more phases of development. If a general model of software errors could be developed, then it may feasible to develop a certification foundation based on the model.

The experimental approach that was followed was to apply a well-known, generic human error model [1] to software defects. The analysis was restricted to this model. It is beyond the scope of the paper to consider the extensive work on alternative error models, cognition, or related topics such as learning theory. Space also restricts the discussion of, and inclusion of references to, the large corpus of other work on certification, testing, and analysis.

The paper begins with an abbreviated description of the generic error model. This is followed by the introduction of a software-oriented interpretation of the model. The interpretation was used to develop an error-based strategy for testing and analysis that included an integration of both informal methods such as test and analysis checklists, as well as well as more "formal" methods such as coverage testing and static analysis.

The error-based approach was applied to a collection of known defects, examples of which are included. For each of the defects, the effectiveness of twelve well-known methods was evaluated. The methods consisted of: black-box, branch coverage, mutation, model-based, random and bounded exhaustive testing, and informal reviews, pair-programming, static analysis, formal verification and model-checking. Their effectiveness was compared to that of the error-based strategy.

In order to properly investigate the error-oriented strategy it was necessary to have examples of defects for which the originating errors could be determined. The 19 members of a graduate class in software testing and analysis, which included masters students from industry, as well as doctoral students engaged in advanced research, produced examples of errors that they personally experienced. This resulted in a set of 38 defects, from 36 programs, which ranged from student projects to industrial products such as cell-phone systems. For each of these defects, the students estimated the effectiveness of both the traditional methods and the error-oriented approach.

II. REASON'S ERROR MODEL

The experiment used the generic error model described by Reason in *Human Error* [1]. Reason describes a model that divides errors into three broad classes: *slips*, *rule errors*, and *knowledge errors*. Rule errors are further subdivided into *bad rules* and *rule misapplication*. Knowledge errors are subdivided into *inaccurate mental models*, and *limited-workspace*. Additional subcategories are described of which the *prospective memory* error subclass is included here.

A *slip* is an error similar to a slip of the tongue. It occurs when a correct "solution" to a required action has been formulated but a slip is made in its execution. In this context, the human is performing at the *skill* level, in which there is no conscious deliberation.

Rules are considered to be pieces of knowledge of the form "if condition then do action". They are established solutions that are repeatedly reused. *Bad rules* correspond to bad solution techniques that are wrong and need to be unlearned. *Rule misapplication* can occur in different ways, such as failure to satisfy all of the condition part, or incorrect application of the action part. It is noted that not all models include rules, a fact that is discussed by Reason in his book.

Knowledge-based errors are associated with the more laborious parts of problem solving, in which the solver has to resort to step-by-step reasoning from first principles. *Inaccurate or incomplete mental models* correspond to errors resulting from ignorance. *Limited-workspace* errors are caused by the limited "bandwidth" of the human brain, which can only simultaneously consider a small number of things at once. *Prospective memory* failures refer to the situation in which there had been a conscious intention to do something, but the resolution was lost. In addition, Reason discusses *violations*. These occur when the solver knows that some action may not be appropriate, but for various reasons, such as the press of time, does it anyway.

Space limits a more detailed description of the model.

III. SOFTWARE INTERPRETATION OF REASON'S MODEL

For each of the different kinds of errors in the model, a software interpretation was developed. Examples are

given for the error classes from the set of 38 defects. For each error class the application of appropriate test and analysis methods is briefly discussed. In some cases, an example could have been associated with more than one error class, and this is noted.

A. Slips

Simple manifestations of this kind of error include accidentally using the *wrong variable name*, or typos involving the *wrong arithmetic operator*. At a more abstract level, there may be very similar kinds of alternative actions and the wrong one is accidentally substituted. Associated methods include prevention techniques such as standards for naming variables that will make it more difficult to confuse them. Detection techniques include type-checking, which may detect the use of the wrong variable or operator. There was one example of a slip.

a) Robot hand: The example concerns a robot hand program. If a command is made to manipulate an object, the program is supposed to move the hand into a goal position. It does this by iterating a sequence of small moves, each followed by a measurement of the new position. The defect occurred because the programmer accidentally used the initial position variable rather than the updated position variable while doing the updates. It was recorded that the programmer simply forgot the meaning of the variable name, a little stronger than a slip in which the substitution is subconscious (and for which variable naming would have been an effective prevention technique), so this problem could also be classified as a limited-workspace error (see below). From a limited-workspace point of view, tests based on input domain modeling, as described below, were judged to have been an effective detection method. A type distinction between the immutable and updatable variables would also have worked

B. Rules

In the context of software development, a rule can be considered a piece of stored knowledge on how to accomplish a program construction task. For example, at the lower, programming/detailed design level, the programmer knows how to translate informal solution concepts such as "consider an entity to be a set of items and then repeat over the set", or the "means for stopping the repetition", into code constructs such as *arrays*, *loops*, and *loop headers*. At a slightly higher level of abstraction, rules include programming techniques, or *idioms* such as scanning a sequence until a special marker is seen. We refer to this example as the end-of-data-marker, or *caboose* rule/technique. Rules also include classic generic programming techniques such as *producer-consumer*, *protocol* and *reader-writer*.

1) Bad rules: Bad rule errors involve the use of bad rules, as opposed to misapplied good rules. One kind of bad rule involves the use of the wrong programming language construct. This occurs when the programmer has the wrong idea of what it accomplishes. A general method for dealing with errors like this is training. This includes the maintaining, by programmers, of their own *personal mistake checklists*. Periodic review may be necessary to avoid repeating the same error. The two examples of this kind of error, both of which could become documented learning experiences, included the following

a) Shallow array copy: It was necessary to create multiple instances of an array data structure. The programmer used the C+-language "=" operator to create new instances from old ones in order to save having to re-initialize it with common data. But this is a shallow copy operation, and the new copies are simply pointers to the original copy.

2) Misapplied rules: These errors occur when a rule-based idea for part of a solution is incorrectly translated into the corresponding code or design fragment. In some examples, several rules were simultaneously active and the problem involved an interaction between the rules. The dominant testing and analysis method in the suggested error-oriented approach for rule errors is the use of checklists. The lists contain the names of programming and design rules. They may also contain the names of risks associated with rules. The critical assumption is that the expert tester, focusing on individual rules, will be better able to construct tests that reveal possible misapplication. For example, experienced programmers know that the risks of using producer-consumer themes include the possibility of the producer overtaking the consumer, resulting in overwriting the buffer that is used to pass data. This focus on a producer-checklist item leads to the use of a review or of tests that address this associated risk. Alternatively, preventative run-time checks can be made in which the status of the buffer is monitored by the producer.

Errors in the application of rules may involve limited-workspace problems (discussed below) so that limited-workspace methods such as black-box testing may be effective. In this case, focus on the rule identifies the local domains over which black-box can be effectively applied. Rule errors were one of the more common kinds of errors in the sample set. Two examples are given here.

a) List of best-match digits: In this example, the *caboose* rule was misapplied. The problem occurred in a program which finds the best matches for a character from a set of samples. The matches are stored in a best-choices vector. The error occurs when the number of equal matches for a digit is equal to the length of the vector, so that no end-of-stream marker is inserted in the sequence of tied matches stored in the vector. It was overlooked by the programmer/designer when considering different possible outcomes. When no end-of-data marker is inserted, the program attempts to go beyond the end of the vector. The error-oriented tester is guided by the meaning of the rule

names in checklists to consider the common cases in which their instantiation could fail. In this case, it can be expected to include the occurrence of an empty sequence, a maximal sequence, or one in which the marker is missing. Tests of the second type reveal the problem.

b) Parser not working fast enough: This example involved a client process that accepts a command which it hands off to a parser, and then to a worker for processing. The worker does not copy the parsed command but works with its parsed source directly. This means that if the parser is given a subsequent command for processing, the new command could erase the previous parsed command before the worker has processed it. In addition, when the second command is parsed and a worker assigned, there could now be two workers working on the same parsed command, so the command could be performed twice. This could have also been classified as an inaccurate mental model error (see below) but the programmer indicated knowing about the possibility but overlooking it in the drive to produce an initial solution. It contains an example of the use of the *producer-consumer* design rule.

Focused consideration of the producer-consumer aspect, possibly in a post-construction review phase, will prompt the consideration of risks such as the producer overwriting the buffer before the consumer has finished with it. This could result in detection of the defect during informal program review. Because of the difficulties of constructing tests in a multithreaded application, it may not be possible to easily provoke this failure, but the consideration of its possibility, prompted by the checklist key word, was judged to effectively deal with the potential problem through the use of run-time checks in the code.

C. Knowledge-based Errors

1) Incorrect or inaccurate model of the problem space: These errors often involved a lack of correct information about some "other" component which was not constructed by the programmer/designer. Many of these errors can be described as false assumptions about an interface between two components.

The prominent method for this kind of error is to identify interfaces and then consider assumptions made about the interface. The next step is to examine the specifications "across" the interface, or test the assumptions when the specifications are not available. In the case where the testing is being done by the programmer, the explicit assumptions will be known. When it is being done by a tester who was not the programmer, and explicit assumptions are not documented (e.g. in comments), the tester needs to reconstruct potential assumptions.

This was also one of the more common error types seen in the sample. Three examples are given.

a) Too many filters: This example occurs in a program that sends an SQL request to a DB server. Unknown to the programmer, if a request contains more than 1000 DB filters they are simply ignored and all the data is returned. This is a discoverable assumption, based on the property "number of items". This is an interface that should be checked with specifications or, if these do not exist, by testing.

b) Commas not processed: In this example, one component calls another component to process some data and return the result. More specifically, the application calls a parser for CSV files. The programmer depended on an explicit assumption that the parser recognized quotes for embedded commas that should not be treated as separators. This led to mis-parsed files, leading to subsequent failures. The programmer indicated that this was an explicit (and as later found out) false assumption. In this case, if specifications were not available for post-construction examination, then black-box input domain testing would work.

c) Too many credit card items: In this example, a customer payment system uses a website that does authorizations. The programmer/designer did not know that the website limited transactions to 20 items. If more are sent, the site returns a confusing message saying that the submitted total does not match the sum of the individual items. This was a discoverable assumption, based on the number of purchased items limit. It could have been checked and confirmed with testing. Protection against unchecked values could be implemented with an invariant assertion at the point of the call to the third-party component.

2) Limited-workspace: This was the most common error. For all of the examples in this section, the programmer/designer reported that a limited-workspace error had occurred as opposed to, for example, an inaccurate mental model or a rule violation. There were just too many things going on to accurately keep track of them all. In a limited-workspace error, programmers typically reported that they overlooked something that, at some level, they "knew about". For example, in the case of a variable-overflow-limited-workspace error, the problem was not due to ignorance or complexity but (unavoidable) lack of focus. The suggested remedy is to use post-construction testing and analysis phases that can focus on each of these aspects individually, evaluating them for correctness. Simply focusing on potential overflow situations after construction when it can receive undivided attention may be enough to evoke the consideration of effective overflow-revealing methods. Two focus-facilitating approaches to the prevention or detection of limited-workspace errors are the use of *model-based testing* and *iconic errors*.

In the error-based paradigm, standard black-box testing is interpreted as an error-based method in which the separate, undivided focusing of attention on different program aspects is facilitated using an *input model*. A typical input model decomposes an input domain into

"functionally equivalent" subcases. This allows the tester to concentrate on each subcase in turn, to see if it is correct. The input model also more clearly identifies "edge" or "oddball" cases.

Depending on the application, different kinds of standard models, other than simple input equivalence partitioning, may be appropriate. Examples include *activity diagrams* (flow charts), and *decision, state* and *bounded-combinations models*. In each case, the goal is the same: to assist the programmer in identifying and then focusing on each of a set of different aspects of the code. If modeling is carried out before hand, errors may be identified during pre-construction phases. Post-construction, it leads to techniques such as model-based testing.

The goal in model-based testing is to assist the consideration of not only separate aspects of the code, but the way they work together. An alternative that is available in certain situations is the use of *iconic errors*, which capture the essence of certain kinds of limited-workspace errors in a reusable form. Examples include: *logical memory leaks*, *overflow*, *deadlock*, *bookends*, *round-off*, and *data race errors*. The idea is that the code is examined for the potential occurrence of the error, leading to the testing for the errors in the context of the given program. The following five examples describe a variety of limited-workspace errors. In each case the programmer/participants in the study judged that testing based on models and/or iconic errors would have been an effective detection method.

a) Two classes allowed at once: This example involved a system that allows students to register for classes. It incorrectly allows the situation where a student registers for a class that is held at the same as a class that has been previously registered for by that student. In this error, a special subcase was excluded from consideration. The programmer reported overlooking this possibility.

Error-directed testing in this case might best be helped with an *input model* specification. In this case, inputs include the current system state plus the new request. Standard invalid input tests based on an explicit input model (as opposed to the incomplete mental model of the programmer) would reveal the defect.

b) Bad rendering: In this example, a 3-D rendering routine failed to work for a certain kind of object. For this kind of object, a shading routine was called with the wrong parameters for that subcase. An *activity diagram* model constructed either during design or after code construction would have laid out all the different aspects of the code for more focused analysis or testing. The application of standard black-box testing to the different subcases in the model would have detected the problem.

c) No profile updates allowed: In this case a job search data base was being constructed which was accessed by email addressee keys. The system correctly allowed the insertion of a new searcher record, but incorrectly disallowed updating of existing records. The problem was the programmer had failed to distinguish the insertion from the update case, leading to attempts to update an entry with an INSERT rather than an UPDATE command. The programmer reported knowing about the alternative, but simply overlooking it. Several models would have allowed a retrospective consideration of alternative flows in the solution, such as an abstract program activity diagram or a user *state model*. The models would have included the ability to update as well as insert, and model-based testing would have led to the discovery of the program defect.

d) Remember scripts forever: A stock-trading system recorded scripts of system usage. The system did not delete the scripts, neither after some period of time nor after a max limit was reached. The system ran fine until certain state parameters made their existence known in a failure. This was a latent *logical memory leakage error*. In this case, focus on the possibility of this iconic error will involve the consideration of places where memory is allocated. Analysis (or associated testing) will reveal the overlooked parts of the program solution corresponding to the missing code.

e) Rest of the value missing: This was a word search program which required the use of a merge-sort routine. A division of n by 2 was needed. It was implemented with an integer n/2 division, resulting in the loss of an element each time n was odd. In this error, certain computational subcases were ignored, Tests based on a *roundoff* iconic error checklist item would reveal the error. The tester would look for potential occurrences of roundoff, and then using knowledge of corresponding expressions or code in the given program cause the appropriate tests to be executed.

3) Prospective memory: Two examples are given here. Both are errors of omission. In prospective memory errors the programmer/designers were aware of something that they consciously intended to do, but then forgot. Forgetting to do something in the sense that it never occurred to the programmer in the first place is different, and could be a limited-workspace error. Prospective memory error-oriented methods include *mementos* and *bookends*. In mementos, a programmer uses comments to document intentions when they occur, which are then later checked for satisfaction. Bookends-focused testing and analysis, described above in connection with logical memory errors, involves expected pairs of operations or events in which one part of the pair has been left out. Bookends testing and analysis is carried out to confirm that for each bookend, there is a matching pair.

a) Missing table unlocks: This example involves a database server which has a thread pool for answering data base requests. The threads put a lock on a table when accessing it for a delete, but the programmer forgot to put in the intended unlock. The missing unlock causes the

system to grind to a halt. Mementos are the most direct technique that could have helped. In the case of bookends analysis and testing, the programmer judged that testing and analysis which focused attention on this overlooked action is likely to reveal the problem. Analysis and testing for the problem is simplified by the fact that the unmatched locks are not followed by unlocks on *any* path, avoiding false positive problems due to infeasible paths.

b) Ignoring return codes: In the second example, a cell phone has an upload function that it can use for uploading diagnostic files to the server. If a voice call is received while this transmission is in progress, the upload is lost. The cell phone programmer made a mental note to check the return code of the uploading function, which would have revealed the problem, but forgot. The *memento* approach could be used here. A bookends approach would try to match variable value definitions with uses. In this case, it would reveal the there is *no* use of the defined return code in the program on *any* path.

D. Violations

Violations, in which the programmer knew to do something but did not do it, or did something known to be wrong, could theoretically involve anything. However, in practice, it appears to involve programmers acting as though they had an incorrect mental model or a limited-workspace problem. This makes violation errors approachable using the methods associated with those other error classes. The samples included two violations.

a) Cell phone logical channel degradation: In this example, a cell phone communications program was written in which it was assumed that if initial program strength was strong enough, and it was possible to decode the first of a potential sequence of logical channels, then further communications actions could continue unchecked. However, the programmer knew that physical strength could deteriorate, and that channel decoding is influenced by other factors such as volume of data. The programmer reported acting as though holding an incorrect mental model of the signal strength problem. In this case, there is an interface between the receiver and transmitter of the signals. Examination of assumptions across this interface, resulting in their evaluation during testing and analysis would have lead to discovery of the error.

b) Web page button annotations too long: The second example involves a web-based application. A web page is displayed with buttons having annotations. The annotations are dynamic, being supplied to the page-rendering code at run time. The annotations are displayed before their associated buttons. As a consequence, a long annotation may push a button right off the page, making it inaccessible to a user. In this case, the programmers simply ignored what the web site designers had done, including the problem that the design might have with longer annotations. They could be interpreted as behaving as though they had a limited-workspace problem. Limited-workspace error testing would include the construction of an input model for the page rendering code, with its annotation text. Testing this over extreme values, e.g. annotation length, would reveal the defect.

IV. Effectiveness

The 19 programmers who supplied the 38 examples documented the errors that led to the defects and evaluated the potential effectiveness of both the error-based approach, described in the previous section, and the 12 standard testing and analysis methods listed in the introduction. For each of their defects, and for each method, they asked themselves if the method would have found that defect. Positive answers included both "yes" and "probably." Consider, for example, white-box branch coverage testing. If a program has the property that it would fail whenever a branch was executed, then branch coverage would force exposure and the answer would be "yes". If the program was such that a failure would occur whenever the branch was executed for some data, but not others, but the data was the most likely to be chosen, then the answer would be "probably". Descriptions of the programs and the raw data for the experimental summary are contained in an associated 182 page report. The results were edited for consistency by the author.

The results are less scientific than formal experiments on a small set of programs over a small set of methods, such as the classic work described in [2], but the approach was necessitated by the scope of the investigation in which a large set of methods is evaluated over a very wide range of applications. Compensatory testing is defined to be the combined application of all 12 methods. The idea is that a weakness in one method may be compensated for, in some undefined way, by another method. The reported effectiveness of the 12 standard methods, of compensatory testing, and of the error-based approach for the 38 defects in the study is summarized in Table 1.

The Table indicates that error-based testing and analysis was deemed effective or probably effective for 35 of the 38 errors. This was followed by compensatory at 29, black-box at 18 and BET (Bounded Exhaustive Testing) at 19. Random and model-based testing were good for 11 and 10 defects, and the rest were in the single digits. Error-based testing was found to be a way of positioning methods in the development process, providing a rationale for their use, and improving their effectiveness by directing and focusing their application. Consider black-box testing. A standard approach in black-box testing is to decompose the input domain into "functionally equivalent" classes and to then test each of these, plus the oddball/edge-cases. The error-based viewpoint improves its effectiveness in the following way. Black-box is generally applied to whole programs or program components. The checklist items associated with rule or workspace errors direct the application of black-box testing to other aspects of the program that are not immediately clear by looking at program interfaces, such

as implementation programming constructs.

There were six examples where only an error-based approach was judged to be potentially effective. A common way in which error-based was effective on its own was in situations where it would focus attention on the possibility of rule misapplication or on the occurrence of iconic errors. In these cases, the focus that is gained from the associated keywords was judged by the programmers to be necessary and sufficient for probable defect discovery.

TABLE 1
METHOD EFFECTIVENESS

Method	Yes	Probably	Total
Black-box	11	7	18
Branch Coverage	3	3	6
Mutation	4	5	9
Model Based	8	3	11
BET	16	3	19
Random	4	6	10
Informal Review	1	5	6
Pair Programming		6	6
Static Analysis	2	2	4
Dynamic Analysis	2	3	5
Model Checking		2	2
Formal Verification	4	1	5
Compensatory			29
Error-based	27	8	35

There were three cases where no method was judged to be potentially effective. In one of these, there was significant missing information. Two others involved obscure properties of the code that were related to peculiarities of the programming language.

V. CONCLUSIONS

An informal evaluation of a broad range of defects indicated that an error-based approach will provide a stronger level of certification than one based on a single, standard method. The strongest effect of the error-based approach was found to be its focused attention on aspects of a program that would otherwise have only been analyzed indirectly or as parts of larger entities. The two principal kinds of focusing mechanisms were *checklists* and *program models*. Two kinds of checklists were described. The first kind focuses on program constructs such as array references and on design idioms such as producer-consumer. The tester is expected to identify potential risks associated with instantiations of these entities and to construct corresponding tests or analyses. The second kind of checklist focuses on risks such as overflow or deadlock. In this case the tester is expected to identify aspects of the program that have the potential to produce such a risk, and to then construct pertinent tests and analyses of those aspects.

The checklist items in the study are generic, whereas examples of the second kind of focusing mechanism, program models, are application-specific. Program models may be developed during specifications and design, but could also be developed by the tester/analyst. Program model components are not simply "covered", as in traditional model-based testing, but are focal objects to be individually tested and analyzed. As part of their focusing role, the components of program models lead to more precise test generators and validation oracles.

The error-oriented approach suggests a certification framework in which it is required that tests and analyses be carried out for each of the different kinds of errors in the error model. Because the program correctness problem is undecidable, there can be no foolproof general certification standard. In the case of the error-based approach, checklists and program models may be incomplete, and their instantiation and application may be imperfect. Certification that is based on error-based testing and analysis should, consequently, include the identification of checklists and program models used in the evaluation.

Because the error-based framework is informal, and because the effectiveness of the included methods depends on experience in their application, the approach would benefit from relevant, systematic theory of expertise. Work on natural decision making, such as that described by Klein in [3], provides such a foundation. Decision-making *critical cues*, for example, can be compared to error-model checklist items. They provide a bridge between the particulars of a program and the knowledge-base of the expert. In addition, Klein's work incorporates the possibility of improvement through experience and variability based on context. It also contains guidelines for the development and maintenance of expertise.

Current work involves further refinement of the approach, including its application to the testing of an industrial real-time system.

VI. ACKNOWLEDGEMENTS

Descriptions of the programs and the raw data of the effectiveness analyses are contained in an associated UCSD/CSE technical report. The authors of that report, and the participants in the course and the experiments, were: W. Howden, G. Cheang, M.G. Chin, X. Ding, B. Emde, M.A. Fedder, S.R. Foster, R.M. Gehrer, L. Geng, A.P. King, J, Lee, C. M. Louie, J.A. Meister, M. Mubin, I. Mulic, A. Ravinagarajan, R. Valentin, H. Sabnani, A. Vekataraman, and E.C. Yip.

REFERENCES

[1] James Reason, Human Error. Cambridge UK: Cambridge University Press, 1990.

[2] V. Basili and R. Selby, Comparing the Effectiveness of Software Testing Strategies, IEEE TSE, vol. 13(12): 1278-1296, December 1987.

[3] Gary Klein, Sources of Power. Cambridge MA: MIT Press, 1999.