STRATGEY AND DOMAINS ADOPTED IN DATA ANALYSIS

1Amit Singh and 2Dr. Pardeep Goel

1Research Scholar, CMJ University, Shillong, Meghalaya

2Associate Professor, Fatehabad, Haryana

Abstract:

Less rigorous strategies are practiced widely in testing. Here we refer to such old favorites as boundary testing, testing zero, one, and many occurrences of some particular phenomenon, and other standard practices given some knowledge of the system specifics, data types, and operators. These adapt to the specification level very easily the only transition required is working with the notation of the specification rather than that of the implementation as is usually done.

Conclusion

Our experiments using testing strategies at the specification level led us to develop two new specification-based testing strategies. The first, domain propagation, is an extension of partition testing. The second, specification mutation, is an adaptation of the existing implementation-based mutation testing technique.

Key words: phenomenon, strategies, specification mutation, adaptation.

INTRODUCTION

This paper examines applications of formal methods to software testing. Which offers many advantages for testing. The formal specification of a software product can be used as a guide for designing functional tests for product. The specification precisely defines fundamental aspects of the software, while more detailed and structure information is omitted. Thus, the tester has the important information about the product's functionality without having to extract it from necessary detail. Testing from formal specification offers simpler, structured, and more rigorous approach to the development of functional tests than standard testing techniques. The strong relationship between specification and tests facilitates error pin pointing and can simplify regression testing. An important application of specifications in testing is providing test oracles. The specification is an authoritative description of system behavior and can be used to derive expected results for test data. Review The computation of the success/failure verdict of test execution tools follows from the comparison between the outputs given by the system under test and the expected ones defined by the formal specification. Besides the possibility of computing verdicts for a test case execution, using formal specifications allows one to properly define the conformance relation, which states what it means for a system to conform to its specification. Such a conformance relation depends on both test hypotheses on the system, which allow to consider it as a formal model, and observability restrictions on the system. These observability restrictions are used to select test cases which can be interpreted as successful or not when performed by the system under test.

We informally argued that software testing is difficult. DeMillo et al., Morell, and Voas have separately proposed a very similar fault/failure model that describes the conditions under which a fault will manifest itself as a failure. Using the fault/failure model proposed by Voas and the Kinetic example initially created by Paul, we can define a simple test suite to provide anecdotal evidence of some of the difficulties that are commonly associated with writing a test case that reveals a program fault. As stated in the **PIE model** proposed by Voas, a fault will only manifest itself in a failure if a test case T_f executes the fault, causes the fault to infect the data state of the program, and finally, propagates to the output. That is, the necessary and sufficient conditions for the isolation of a fault in P are the execution, infection, and propagation of the fault [DeMillo and Offutt, 1991, Morell, 1990, Voas, 1992].An oracle is a means to judge the success or failure of a test, that is, to judge the correctness of the system for some test. The simplest oracle is comparing actual results with expected results by hand. This can be very time consuming, so automated oracles are sought. Test case A test is useless if no expectations of behavior are held. Hence, a test case must contain both test data and a test oracle for the data.

Oracle partitioning is a method of breaking up a very large table and/or its associated indexes into smaller pieces. Each piece, in essence, is either a table or an index although they are referred to as ‘partitions’ since together, they make up a larger object. Although indexes belonging to a given table are generally partitioned along with the table, Oracle does support the ability to partition tables and indexes independently such that you could have a regular, non-partitioned table but its associated indexes are partitioned. Each partition will be in its own segment and potentially, and for greatest flexibility, in its own table space (will allow independent backup and recovery). The primary purpose of partitioning is faster query access. This is accomplished via partition pruning (elimination), a method where Oracle can query the data dictionary and determine the content or definition of a given partition without having to query that partition’s data, as it otherwise would in a non-partitioned table. In this way, Oracle can very quickly exclude large portions of data before the query search begins and not have to search through certain partitions at all in order to resolve a query. Rather, very focused subsets of data can be quickly isolated to be further refined.

Material and method

Some discussion of specification-based testing strategies is in order. Though strategies aren't central to this thesis, we use quite a range in demonstrating the framework. This chapter discusses using existing strategies with the framework (essentially, using strategies at the specification level), and two new strategies we developed.

We do not need to invent a gamut of new testing strategies for specification-based testing. Most existing strategies already use either generally applicable selection criteria or specification-level criteria. We can use these strategies with little or no adaptation to the specification-level. There are two issues in adapting strategies for specification-based testing:

- how differences between the implementation and the specification affect the strategy, and
- how the strategy can make full use of the specification.

Dealing with a specification can affect a strategy due to the abstract nature of the specification, certain elements of specification style, or features of the particular specification language used. Certain implementation concepts are alien in a specification. For example, the concept of a path through an implementation does not transfer well to an abstract specification where the detailed steps in transforming input to output are not defined. So, a strategy like path testing does not adapt well to specification-based testing. Specification languages commonly use different standard data structures such as sets. Testing involving data types can only be concerned with a conceptual understanding of the data type, rather than some implementation representation such as linked lists. However, there may be little or no adaptation required. Input partitioning, for example, is a concept perhaps more applicable at the specification level than at the implementation level. Using partitioning strategies on implementations usually requires deriving abstract expressions for conditions over the input. Such expressions are specifications, and if they are not already explicit in the specification they should be easier to derive from a specification.

Clearly, knowledge of the specification notation is required to extract relevant information such as condition expressions. Some strategies may be able to make use of details of the notation in the specification, particularly any pre-defined operators in the language. We consider adapting some popular strategies to give the flavor of using strategies at the specification level.

Partitioning strategies divide the input space into domains according to some criteria. The most commonly used criteria are branch conditions using variables of the input space. Domains of such a partition are determined by reducing the conditions in the input expression to disjunctive normal form such that each disjunctive is disjoint. Each disjunction is a constraint over the input which defines an input domain. Other partitions can be just as easily defined, though not necessarily so easily derived. Partitioning the input space based only on the input expression can be a pitfall in specification-based testing. Some partitioning strategies partition the input space using more information than contained in the input expression. An example is cause-effect mapping. With the cause-effect strategy, input `causes' are mapped to output effects. In terms of partitioning, this requires an output partition to be determined, and then the input partition is based on the input domains that map to the identified output domains. This output partition is determined by reducing the output expression to disjunctive normal form.

Domain testing [WC80] uses the control flow of a program to partition its input space. The path predicates form boundaries of the various input domains in the program's input space. The strategy tests for domain errors by checking whether the domain borders are in the correct position. A major pre-requisite of domain testing is that the path predicates have a linear representation in the program's input space, i.e., if a graph of the input space is constructed, the path predicates define domains with linear structures. The dimension of these structures depends on the number of variables in the path predicate. Domain testing also assumes that there is no coincidental correctness, there are no missing path errors, adjacent domains compute different functions, the correct border is also linear, the input space is continuous, and there are no loops in the code as this greatly increases the complexity of the path predicates.

The path predicates are easily determined from a specification by reducing the input expression to disjunctive normal form. The disjunctive are the path predicates and represent the domain boundaries. A much more significant problem with adapting domain testing to the specification level is finding linear representations for the path predicates. That is, finding a way to represent the predicate so that it forms a linear structure in the input space. It is common for path predicates to be high level expressions involving complex data types. In some cases, a linear representation suggests itself, but there is no guarantee that a linear representation exists. For example, sets and set operations defy linear representation1. It is probably more likely that a linear representation does not exist it depends largely on the problem specified. If a linear representation can be found, however, domain testing is a very appealing strategy to use. Another consideration is that specifications commonly use discrete spaces. Numerically, the naturals and integers serve in most specifications, and data types are likely to be represented by discrete spaces if a representation can be found at all. This is not a problem per se; continuous input spaces are advantageous because they allow arbitrarily accurate testing. Testing with discrete spaces has limitations on accuracy.

Less rigorous strategies are practiced widely in testing. Here we refer to such old favorites as boundary testing, testing zero, one, and many occurrences of some particular phenomenon, and other standard practices given some knowledge of the system specifics, data types, and operators. These adapt to the specification level very easily the only transition required is working with the notation of the specification rather than that of the implementation as is usually done.

Conclusion

Our experiments using testing strategies at the specification level led us to develop two new specification-based testing strategies. The first, domain propagation, is an extension of partition testing. The second, specification mutation, is an adaptation of the existing implementation-based mutation testing technique.

Reference

1. Man-yee Chan and Shing-chi Cheung. Applying white box testing to database applications. Technical Report HKUST-CS9901, Hong Kong University of Science and Technology, Department of Computer Science, February 1999a.
2. Man-yee Chan and Shing-chi Cheung. Testing database applications with SQL semantics. In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, March 1999b.
3. David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weyuker. A framework for testing database applications. In *Proceedings of the 7th International Symposium on Software Testing and Analysis*, August 2000.
4. David Chays and Yuetang Deng. Demonstration of AGENDA tool set for testing relational database applications. In *Proceedings of the International Conference on Software Engineering*, pages 802–803, May 2003.
5. David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. AGENDA:
6. A test generator for relational database applications. Technical Report TR-CIS-2002-04, Department of Computer and Information Sciences, Polytechnic University, Brooklyn, NY, August 2002.
7. B. Choi, A. Mathur, and B. Pattison. PMothra: Scheduling mutants for execution on a hypercube. In *Proceedings of the Third ACM SIGSOFT Symposium on Software Testing, Analysis, and Verficiation*, pages 58–65, December1989.
8. L.A. Clarke. A system to generate test data symbolically. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.
9. Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the 8th International Conference on Software Engineering*, pages 244–251. IEEE Computer Society Press, 1985.

10. David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–87, September 1996.