VV	Course:	Software Verification and Validation, DAD404		
	Teacher:	Olle Lindeberg	Department:	IDE, University Of Karlskrona/Ronneby

Unit and System Testing Report

Prepared for

Olle Lindeberg
Olle.Lindeberg@ide.hk-r.se
IDE, University Of Karlskrona/Ronneby

Prepared by

Christian Bucanac
c.bucanac@computer.org
Software Engineering Student,
University Of Karlskrona/Ronneby

1998-12-05

Contents

Introduction

This is the second report in the Software Verification and Validation course, DAD 404 given at the University of Karlskrona/Ronneby. This report is about Unit and System Testing.

The report summarizes the main issues from the articles found in the reference section. The main issues are thereafter discussed to see if they support or contradict each other.

This report together with the other student's reports is used in the Unit and System Testing seminar held during the course.

Summarization of the main issues

Overview of Test Techniques

There is no best or a set of best methods for testing. Each testing technique tests differently and reveals different bugs. No method can guarantee to find all bugs. The objective in testing techniques is not which testing method is best. The objective is to use the testing technique that catches as many bugs as possible.

This article gives an overview of five different testing techniques:

- Path testing – The objective is to go through enough different paths to demonstrate that a routine's actual structure matches it's intended structure. Theoretically, it could mean that there is infinite number of test cases. The goal is to select a small but sufficient set of test paths.
  There is a special difficulty in testing loops, since they are the major cause of generating infinite number of test cases. Nestled loops can result in very large test execution times.
- Transaction flow testing – A transaction flowchart is usually used for representing the programs logical structure. This flowchart is used for transaction flow testing. It is a functional testing from the user's point of view. The testing technique is otherwise very similar to path testing.
- Input validation and Syntax testing – Input validation involves validating the input. No garbage input is allowed into the system. In syntax testing you are checking and validating the syntax of the input. The input strings are checked against a predefined input format. A string is either accepted or rejected.
- Logic based testing – A decision table is used in logic based testing. When a combination of decisions satisfies a rule, an action is taken. Decision tables are easy to implement in code, which makes logic based testing easy.
- State transition testing – A state transition graph consists of states and transitions. Inputs and outputs from a state are tested. The difficulty in this testing is that there might be a lot of states and improper states.

Partition Testing Does Not Inspire Confidence

Partition testing is not a good testing technique for confidence testing. Theoretical and statistical analysis of partition testing show that partition testing is most useful when you suspect some partition domains of having high failure probability. Partition testing is most valuable when there are partitions with narrow suspected areas where failures will most likely occur.

The partition testing method is compared with the random testing method. All comparisons made favored partition testing. The margin is very small. The random testing is at least 80% as effective as partition testing. Partition testing is better when classes have higher failure rate than the overall failure rate. The random testing is better when classes have lower failure rate than the overall failure rate.

Random testing is a good alternative for partition testing. It is much cheaper. Running more tests will outweigh its disadvantage compare to partition testing. The partition testing can be automated, which makes them more attractive than partition testing.

Markov Analysis of Software Specifications

Markov models can be used for understanding the software easier. The Markov models are based on specifications. From a model you can obtain any number of statistical test cases. These test cases are mainly used in the Cleanroom process model for generating random statistical test cases. A random number generator can easily do the generation of test cases. A test case starts in the start state, through a sequence of states and ends up in an end state.

The idea with a Markov model is to define probabilities for the usage of the software. There are two steps in constructing a Markov model. First in the structural step, you define the states and the arcs. In the next statistical step, you assign the transition probabilities. There are three approaches for the statistical step:

- Uninformed – Involves assigning uniform probability distribution across the exit arcs for each state.
- Informed – Used when there are some use-data available, for example a prototype or earlier version of the software.
- Intended – You hypothetically create runs of the software that you think will be performed by a careful and reasonable user.

One major advantage of using a Markov model is that analytical descriptions of the set of test cases can be generated in advance. You can in advance calculate how long the testing will take and what resources will be needed for performing the test cases.
Usage modeling, like a Markov model, focuses on understanding what the user will do. All cases are probabilistically modeled in the Markov model.

Integration

Integration testing is done to discover inconsistencies in the interfaces of two elements that are integrated. The aim is to discover both indirect and direct interface problems.

There are various tactics for integration testing. I present some of them that I think are the most important ones.
One problem is to determine which element calls another element. Using call trees can solve this problem. They model every call made by every routine until it reaches a routine that does not call any other routine.
A data dictionary can be used for determining and documenting data dependencies. It lists the properties and intended use of all data. Data dependency graphs can be used for illustrating the data dictionary.
It is very important to specify the interfaces between the elements. There are mainly three things that needs to be specified: number of parameters, the types of parameters and what data the parameters can hold. The last one is most important. It makes it possible to check and validate the inputs that come into the element. Rigid interfaces are important for limiting interface bugs.
Do not allow elements to have multi entry or exit paths. It is better to implement multi entry paths as a single entry path. The elements should internally direct the single entry path to multi paths. This makes the integration much easier and less complex.
Integration testing is very difficult in systems that have interrupts. Interrupts increase the number of possible paths, almost infinitely. Every instruction in every procedure, that is where an interrupt can occur, can be seen as a possible path. This can be partially solved by reentrant procedures. Reentrant procedures can be interrupted at any point and can be executed again. It is totally independent from other procedures.

The final goal of integration testing is to have fully functional system. This goal is reached by putting together elements, doing integration testing and add more elements to the newly integrated element. This is iterated until you have a fully functional system.

There are mainly two ways of integrating a system:

- Top down – Integration testing is started with the topmost element. Stubs simulate the sub elements. The stubs are replaced by the real sub elements after the topmost element has been tested. The procedure is repeated until the whole system is integrated.
- Bottom up – Integration testing is started at the bottommost element. Drivers simulate the upper elements. The upper elements are replaced by the real upper elements when the bottommost element has been tested. The procedure is repeated until the whole system is integrated.

Note that there can be more than one topmost or bottommost element.

PODS – A Project on Diverse Software

The purpose of the PODS project was to determine how different software development techniques affected the reliability of the produced software.
The main objectives were:

- To evaluate the merits of using diverse software (n-version)
- To evaluate the specification language X and its computerized tool SPEX
- To compare the productivity and reliability associated with high level and low level languages.

The second objective was to monitor the software development process with the focus on creation and detection of faults.

The three programs produced in the project were tested back-to-back. The same input data was given to all three programs. Thereafter the outputs were compared. If they were different, then there was something wrong. A majority vote was made for determining the result. When a fault was detected, the program was corrected and the tests were repeated from the beginning.

The test data was derived in different ways:

- Equivalence partition was used for divide the input data into a number of finite equivalence classes.
- Boundary values analysis was used to generate test cases for boundary values of the input and output equivalence classes.
- Decision tables were used to calculate possible combinations of input conditions, output conditions and internal program states.

The conclusions from the project are:

- Diverse implementation is effective in reducing failure rate.
- Diverse implementation provides an economic way of performing large number of tests. Diverse implementations make it possible to perform automatic testing with randomly generated inputs and to compare the outputs.
- Diverse implementation did not eliminate all common faults.
- The cost of developing three diverse programs was at least twice as much as developing a single program.

The FREE approach for System Testing

Uses cases are used for developing a system test plan. A use case is a sequence of external inputs to a system. This sequence accomplishes a task from a user's point of view. A use case is a dialog between the system and the actor, which may be a human being or another system.

The system test cases are derived from the functional specification of the system. It may be a user manual. There are four levels of system testing strategies that can be applied:

- Testing by poking around – There is no formal testing plan. The development team demonstrates the functionality of the system to a subjective degree of satisfaction.
- Functional compliance – A formal test plan is developed for all use cases. This testing level requires use case and event trace coverage. All use case tests must pass to achieve functional compliance.
- Reliability optimization – This level requires that the functional compliance testing has been done according to an operational profile. The use cases are ranked according to relative frequency. This maximizes the reliability.
- Integrity verification – This level requires that the reliability optimization testing has demonstrated thread coverage. This level establishes a measure of system test completeness.

An operational file specifies relative frequencies of a use case. It consists of the operations that compromise all the use cases in a system and the relative frequency of each operation (a use case consists of operations).

Discussion of the main issues

The article "Partition Testing Does Not Inspire Confidence" says "Our main point has been to call into question the common wisdom that confidence in software is obtained by vigorously seeking failures and when a variety of methods finds no more failures concluding that the software will prove reliable in use". This supports the article "Overview of Test Techniques". It says that there is no best or a set of best methods for testing. No method can guarantee to find all bugs. The objective in testing techniques is not which testing method is best. The objective is to use the testing technique that catches as many bugs as possible.

Transaction flow testing and use case testing are pretty much the same. Both techniques can use Markov models. These two testing techniques are more effective than for example path testing. The testing is done on a much higher level, which cover several paths. You get the same failures as a user would get. The problem is that these two testing techniques may be too effective. You might find several faults at the same time. It might be hard to trace the faults and find out what went wrong.
The difference is that use case testing is more focused on the frequency and primarily tests the use cases that are most frequently used. Transaction flow testing is more of a use case coverage testing.

Markov models are very useful in transaction flow testing, state transition testing and use case testing. The problem in transaction flow testing and state transition testing is that there are too many test cases to execute if you are to cover all transitions. It is impractical to execute all test cases. You would need to select a smaller amount of test cases that tests the key functionality. A Markov model can be used for this purpose.
The Markov model is mainly used for modeling usage of software. It focuses on understanding what the user will do. A Markov model can therefore be used in use case testing to generate operation profiles for software. The Markov model can model the frequency of operations in a use case. This is very useful when you generate operation profiles.

Specifying interfaces between elements is very important. We discovered this in the Humphrey project[7]. The head of the department in the project told us that this was the most important and critical part in the project. It was important to have a well-defined interface between each protocol layer. We listen to the advice and defined the interfaces in detail. Each interface had a detailed specification of the number of parameters, the types of parameters and the data that the parameters contained.

In integration testing it is important to check the input data to see if it is acceptable or not. The input validation and syntax testing technique can be used in the integration. This technique can be used to both check the data between large elements like in integration or in small single methods. I think it is impractical to use input validation and syntax testing to check each single parameter in a method.

The integration article says "An element cannot be considered integrated until every path in its real, dynamic, call graph has been explored under test". The article "Overview of Test Techniques" says the same thing about path testing. "The path testing is not complete until every element in the path tree has been called once and all possible callers have called every element".
In path testing it is not possible to test every path of execution, since they may be too many. There is the same problem in integration testing. The problem of executing a small amount of test cases is that you have not covered all paths of execution.

The PODS project used several of the testing techniques described in the other articles. For example:

- The systematic tests in the PODS project, the tests were generated to test the program for correct operation with both valid and invalid input data. The PODS project used the input validation and syntax testing technique as described in the article "Overview of Test Techniques".
- Boundary values analysis was used to generate test cases for boundary values of the input and output equivalence classes. Boundary value analysis is done by checking input values. The technique used here is the same as the syntax checking technique described in the article "Overview of Test Techniques".
- Decision tables were used to calculate possible combinations of input conditions, output conditions and internal program states. Using decision tables to calculate the possible combinations is a part of the logic based testing technique described in the "Overview of Test Techniques" article. This is a very easy technique, since decision tables are easy to implemented in code.
- Tests were defined to check that the values were correctly converted and that faulty input values were recognized. This garbage in technique is described in the "Overview of Testing Techniques" article. It is done by the syntax testing technique.
- The CERL group did not have any specification language like the other two groups. They tried to raise the customer requirement specification to a higher level of abstraction by using state transition diagrams. A Markov model does this easily. It is described in the article "Markov Analysis of Software Specifications".

If you compare the article "The FREE approach for System Testing" with "Markov Analysis of Software Specifications" you see many similarities:

- A Markov model is made out of specifications. The system test cases in the FREE approach are derived from the functional specification.
- The Markov model supports use case modeling by defining states and arcs. The FREE approach uses use cases for testing.
- The Markov model consists of a statistical step where probabilities are assigned to the transitions in the model. This is exactly what is needed for generating operational profiles in the FREE approach. An operational profile models different types of users by using different probabilities on the arcs and by choosing different ways/arcs from the start state to the end state.
- From a Markov model you can obtain any number of statistical test cases. Statistical testing on use cases is exactly what the FREE approach does. The article "The FREE approach for System Testing" says "We test most-likely use cases first because the most frequent used operations have the greatest impact on operational reliability. If a heavily used operation is buggy, the system will fail frequently. If a rarely used operation is buggy, the system will fail infrequently. Selecting tests according to usage frequency rapidly decreases failure frequency".
- A Markov model is usage modeling that focuses on understanding what the user will do. All cases are probabilistically modeled. It is exactly the same in the FREE approach.

Beizer contradicts himself. In the article "Overview of Test Techniques" he says that nestled loops are a nightmare. They produce infinitely number of paths that would be needed to test if coverage was to be achieved.
In the article "Integration" he says in the interrupts section "Every instruction in every routine can be considered as if it is followed by a jump to every possible routine that can interrupt it". Imagine of having a program with nestled loops in a system that has interrupts. In this case you will have more infinite paths to cover than in a system without interrupts.

Conclusion

I could not find many issues that contradict each other. The articles support each other pretty much. The difference between the articles is that the same basic technique, the state-transition trace net, is used on different levels of testing.

The techniques in the articles are based on the same foundation. The difference is on what level or in what way a technique is used. The common foundation is that all techniques rest on a state-transition trace net. For example:

- Path testing – A path can by an if or switch statement change state and go into another path.
- Logic based testing – Depending on which combination of decisions is satisfied the program uses a certain rule to execute an action and enter another state.
- The Markov model, state transition testing, transaction flow testing and use case testing all rely upon a net of states and arcs. A use case is a path of operations throughout such a net. It is the same for the transaction flow. For each transaction there are a number of operations that must be performed.

References

1. Overview of Test Techniques
   Beizer
   Software System Testing and Quality Assurance

2. Partition Testing Does Not Inspire Confidence
   Dick Hamlet, Ross Taylor
   IEEE Transactions On Software Engineering, Vol. 16, No. 12, December 1990

3. Markov Analysis of Software Specifications
   James A. Whittaker, J. H. Poore
   ACM Transactions on Software Engineering and Methodology, Vol. 2, No.1, January 1993

4. Integration
   Beizer
   Software System Testing and Quality Assurance

5. PODS – A Project on Diverse Software
   Peter G. Bishop, David G. Esp, Mel Barnes, Peter Humphreys, Gustav Dahll, Jaakko Lahti
   IEEE Transactions On Software Engineering, Vol. SE-12, No. 9, September 1986

6. The FREE approach for System Testing
   Robert V Binder
   Object Magazine, February 1996

7. Humphrey, Small Software Engineering Project, Spring 1998
   A course comprising 10 study points which at the time of writing is part of the software engineering program at the University of Karlskrona/Ronneby.
   http://apollo.rsn.hk-r.se/~humphrey