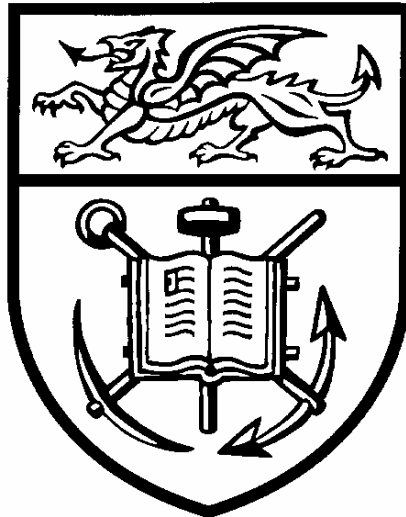


UNIVERSITY OF WALES, SWANSEA



Computer Science Department

Session 2006/ 2007

System Testing

Name: Ayodeji, Kehinde A.

Student Number: 304271

Abstract

There is more to do about testing a system than just seeing that it conforms to its specification. A system can conform to its specification and still do what it isn't supposed to do. To test a software system thoroughly so that its reliability after delivery could be worth its cost, faults lurking in the software should be detected. A look at the problems that make thorough testing difficult and strategies that have been laid down from time to time to face this daunting task is what this paper will go through.

Contents

1. Introduction.....	4
2. Practical Problems.....	5
3. Identifying Test Cases.....	7
3.1 Requirement Constructs	
3.2 Model Based Testing	
4. Functional Strategy.....	14
4.1 Data-Based	
4.2 Event-Based	
4.3 Port-Based	
5. Structural Strategy.....	19
5.1 Node and Edge Metrics	
5.2 Structural Use Cases	
6. Systematic Guidelines.....	22
7. Conclusion and Future Work.....	23

1. Introduction

Every system is tested; at least if it isn't to see if the system does what it is not suppose to do, it will be to see that it does what it is suppose to do. The first execution of a system after it has been implemented puts it under a test. Testing a software system has always been known in the time past to just be making sure that it conforms to its specification. However, as time went by, the increase in the cost of software development along with the demand for the reliability of software systems has pushed testing software systems from just been perfunctory to a process that needs practice and strategy. Software system testing is now the approach that is taken to make sure that a software system meets the demand for more reliable, qualified, verified and valid software.

Software contains faults; the faults can be realized when the software is in execution and the fault executed. With what the software can currently be used for, the software tester has to try as much as possible to detect faults lurking in the software system. Executing faults in a software system when it is actually in use could cause annoyance – in the case of a personal computer crash, or cause disaster – plane crash due to the misbehaviour of a flight control system. Faults that cause this kind of failures are either as a result of specification, design, implementation or verification [Omar and Mohammed 1991]. To detect these faults under the limited resources available for software development is what makes the testing of software system a daunting task. To alleviate the task of testing software systems, a life-cycle model for system testing should run in parallel to the software's development cycle. Though faults of whichever kind are mostly realized when the implementation of the system is under test, earlier stages like planning, preparation and design for testing could begin immediately the specification of the software system is laid down.

Many techniques have been applied to carry out effective testing of software systems, however, these techniques are divided into two broad classes – functional and structural. The classification is based on the source of information for the test case requirements as well as the nature of the fault that is to be detected. These two classes are

complements to each other, making software system testing a process that requires the use of techniques from both classes. At least, if the software testers are to test the software system from the customers view (functional view), what is the essence of the edge they have over the customer that they know the material they are testing and have access to the code that implements it? They can also test it from a structural view – how the software is implemented. This is what this paper will be based on. The aim of this paper will be to mainly talk about the sources, in the phases of software development, from which test cases can be identified and also the different techniques, both structural and functional, that can be applied to make testing software systems an effective and satisfying process.

The remainder of the paper is organised as follows. Section 2 will discuss the different factors that make testing software systems a difficult process when in practice. The source for identifying software test data (test cases) that can be effective in software testing will be discussed in Section 3. Different strategies that have been applied to testing a software system from a functional approach is what Section 4 will be about. Section 5 will be similar to section 4, but from the structural approach. Section 6 will give some guidelines to be applied to testing software systems. Finally, Section 7 will conclude and give prospects.

2. Practical Problems

Testing a software system is not an easy task; even when compared to systems of other kind (mechanical, electrical etc). The intangible and dynamic nature of the software system makes testing it difficult. Software faults are also as a result of the design process and are buried in the software, only to manifest as a result of failure. To add to this difficulty, resources available to carry out this process, which would have been an unlimited process if carried out exhaustively, in practice is limited. In this section, some of the practical problems faced in testing software systems are looked at.

❖ Resource Constraints

Software systems are built under the pressure of a scheduled time and budget. The development takes place according to what is available for use. Placing the system under test, which is one of the last process carried out in the development process is likely to be carried out under pressure. For instance if a fault has been realized after testing a software system that is to be delivered soon and also the cost of fixing this fault will make the cost for the development process surpass the budget, should the software be delivered as it is with fingers crossed?

❖ Cost of test development

In software organisations where testing is taken as a practice, development for testing has a cost of its own. Over 30 - 40% of the cost of software development is about testing it [Veenendal and Pol 1997]. When faults are detected during testing, components of the system that are faulty are sent back to the developers for repair. An iterative loop sets in there. So time is spent in this iterative process and also in the usage of other resources. The cost of test development is also raised by the issue of training the average software developer to understand formal languages used to model the behaviour of the software system under development.

❖ Immeasurable effectiveness

How effective a test plan used to testing a software system is cannot be quantitatively measured. The impact the plan has on the software's quality, reliability and correctness cannot be quantified [Gittens et al 2002]. Being able to know how effective a test plan is for a software system of any kind will help decide if a change or improvement is needed. Most software organisations only tend to claim the effectiveness of their ad-hoc software test plan.

❖ Testers vs. Developers

A standard software organisation is meant to have a team that develops the software system and another one that tests it. The joy of developers is to see that the part of the software system they develop is completely correct. However, for testers, it is to see that they detect faults. In other words, the tester is happy when a fault is detected. With this contradiction, are developers supposed to enjoy working with testers? [Petschenik 1985] Considering the fact that software testing is now becoming a significant part of software development.

❖ Fatigue and Boredom

The main aim of a good software test plan is to provide a systematic approach for generating test cases. Due to its systematic nature, many of the procedures are being automated. The tester just get into place the test suite, test harnesses, test objects and any other thing needed to run the test and just sits there watching the process in execution. If the system under test is a large and complex one, the process used in testing the system take a long time and more use of resources which are done automatically. The automatic approach just let the tester to just spend a while watching the system, leading to monotony, fatigue and boredom.

3. Identifying Test Cases

The main task in test planning and development is to be able to intelligently select test cases that will be used to test the software system's conformance to its specification as well as have the potential to detect errors. However, before one can select adequate test cases to run a test with, there must be choices from which this selection is to be made. The choices are the test cases that need to be identified. In this section, avenues through which test cases that can stand as candidates for testing a software system will be discussed. The constructs for requirement as a means for identifying test case will be

visited. Following this will be the use the idea of model-based testing as an approach for testers to identify test cases.

3.1 Requirement Constructs

People that want to describe the requirement of a software system, especially when they are not in the field of the system development is requested for, is bound to describe the system in an informal way. Examples of statement beginnings that will be made are “When I want to save a file...”, “When this button is clicked...”, “When this number is entered....”, “When this storage device is inserted...”, “After the Enter button is clicked, Screen 5 should appear and” A look at all these clauses will make you agree with these: the first statement refers to an event, the second an action, the third a data, the fourth a device and the fifth a thread (sequence of actions). Jorgensen 1998 [cited in Jorgensen 2002] stated that the five basic constructs used to describe the requirement of a software system are data, actions, events, device and threads. A software system tester who needs to know how the system should behave in terms of how the client describes it needs to have an idea of these requirements and use these constructs as metrics with which test cases are identified. Each of these constructs will be taken one after the other and how each could help the system tester to identify test cases will be examined.

3.1.1 Data

If one of the requirement constructs mentioned above were to be chosen as the only one to describe software, I don't see any reason why the data doesn't almost get that position if it actually doesn't. In the world of the testers, the data is one thing that causes the combinatorial explosion of test cases especially when testing from the functional approach. The various forms a data could take in a software application give the data the potential of being able to make test cases visible. In an application like e-commerce applications, one could talk of datasets, data files, data streams, checkpoint data, metadata and other form of data representations [Merzky et al]. A data could be persistent (relational database), could be volatile (Random Access Memory) and could be corrupted as a result of a fault in the software not

detected during testing – probably because a test was not run using a test case that was to be identified using the data construct. For example, in the Entity-Relationship modelling of the relational database, the relationship between entities may be of a one-to-one, many-to-one or many-to-many cardinality, test cases can be drawn out to test that these cardinalities among the appropriate data entities exist.

3.1.2 Events

The Oxford English dictionary defines an event as anything that happens or takes place. In the process of testing software systems, test cases can be drawn out from anything that can happen in the real environment in which the software will operate. For instance, if a coin is dropped in the part of a vending machine that only accepts notes, how should the system respond? Should a software application delete data from your PC without asking for a confirmation when the delete button is clicked? Test cases realized from events are usually executed to test the software system for non-functional requirements of the system (robustness, safety, stress, etc) and is also performed with the intention to break the system. Ron Patton will call it a “*test-to-fail*” [Patton 2000].

3.1.3 Actions

Whenever a user interacts with a system, there are things expected to be seen as a result of the expected behaviour of the system. Actions are physical happenings on the system that can be used to judge if the system is functioning as expected. For instance, a coffee machine not performing the action of returning your change after payment is made or not dropping out your coffee cup as requested shows that the system’s misbehaviour. Actions such as these should be tested for and test cases can be identified this way. Low level actions such as addition, subtraction, conditional decision to determine the flow of control and many others are not exempted.

3.1.4 Devices

Software systems have a means of interacting with their environment. Devices and human are actors that will interact with the system. Test cases can be realized to test for ways interactions will take place. In an operating system for example,

when the interaction is with a file storage device (e.g. USB drive), test cases that can test for actions like moving files, renaming files, deleting files and storing files. These are actions that happen between the operating system and the file storage system that manages the USB device. So taking a look at all the media through which a system will interact with its environment, one can identify cases that are to be necessarily tested for.

3.1.5 Threads

Threads at the system level are defined to be a sequence of atomic system functions – functions of the system that at the view of the customer will look uninterrupted [Jorgensen 2002]. The thread usually consist of a sequence of the other four constructs and test cases of a thread nature (sequence of actions, events or data) are identified when an artefact that models the system's behaviour is put down. Section 3.2 will go through that.

In brief, requirement constructs are the basics for identifying test cases as beginning to plan a test by identifying test cases as early as possible using the requirement of the software is of an advantage as time is used properly and the cost of fixing an error found at an early stage of software development is less.

3.2. Model-Based Approach

Artefacts made by the requirement engineers, designer tester, or developer to describe the mental understanding of the software system under development at any phase of development can be of great use to the tester. The system to be tested will be tested finally at its most concrete level – after complete implementation. The system as a whole could be a composition of many components; hence information extracted from artefacts created for each component at each phase of development is useful to the system tester. Model based testing is an approach that has brought out the ease in this area. Right from the phase of requirement analysis to coding, structured, reproducible and motivating models could be created at each phase and will be useful to the tester a great deal as it is to the developer. Modelling a software system has eased the automaton of software testing by just needing to identify the test cases looking at the model. In this subsection,

different kinds of behavioural model will be briefly discussed and an example will be given with the FSM model.

Finite State Machines: The finite state machine model can model any system with a finite number of states. Actions carried out when in a particular state cause transitions between states. The behaviour of the system to be modelled determines the number of states needed, which state is transitioned to when an action is carried out in a specific state. FSM model has its theory based on Automata Theory. Using this theory in addition to the graph theory, test cases are identified making use of the nodes and arcs in the graphical correspondence of an FSM model as metrics. [Friedman et al 2002] describes an extensive use of the FSM model in identifying test cases as well as selecting adequate ones. Software systems that are modelled using the FSM model are mainly menu-driven systems like ATM machine, vending machine etc. To illustrate, we identify test cases in the model below.

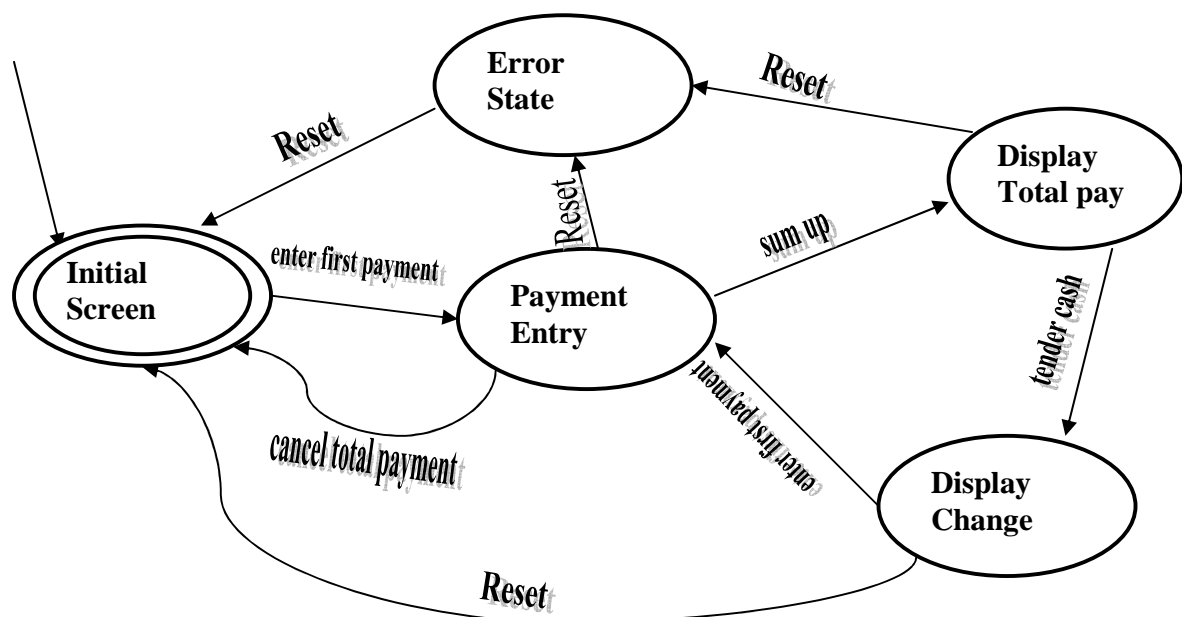


Figure 3.1: A Finite State Model of my Cash register at work

From the figure 3.1, just taking a look at the figure, what one can test for is easily identified even if they are not adequate to cover the test requirement, they will be

candidates for selections. The different paths in the graph from the **Initial Screen** back to itself are sequences of actions that can be tested for. Examples are:

- *enter first payment -> Reset -> Reset*
- *enter first payment -> sum up -> tender cash -> Reset*
- *enter first payment -> cancel total payment*

Combining these sequences in different orders make the amount of test cases that this cash register could be tested with astronomical. Imagine the amount of test cases that could be identified when the system under test is more complex and sophisticated.

StateCharts: The statechart model is an extension of the FSM model. It is the FSM model with some annotations and restriction value in each state in the FSM. The use of hierarchy, external conditions determine if transition takes place when the system is in a given state. The statechart is intuitively equivalent to the most powerful form of Automata (Turing Machines). Consequently, it mainly addresses modelling complex and real-time systems.

Markov Chains: The Markov chain model is a stochastic model are usually used to model software systems with discrete parameters and finite-state time. It is similar to the FSM but has its basics in probabilistic automata. Instead of having events on the edges in the graph, they are labelled with probabilities that sum up to one for each outgoing edges. Besides identifying or generating test cases, Markov chains are used for systems that gather and analyses failure data to estimate measures such as reliability and the mean time of failure.[Yücesan 1990] contains more details on Markov chains.

Grammars: Grammars are used to describe different syntax of programming languages. They are equivalent to different forms of the state machine. For example Context-free grammars (Finite State Machines), Context-sensitive grammars (Turing Machines), Regular grammar (Push down automaton). Software systems or aspects of software

systems are that have to do with lexical analysis or parsing is best modelled with the grammar model. Example of a grammatical model of the cash register in Figure 2.1 is

INITIAL SCREEN ^{*def*} = *reset ERROR STATE, reset DISPLAY CHANGE,*
cancel total payment PAYMENT ENTRY
PAYMENT ENTRY ^{*def*} = *enter first payment DISPLAY SCREEN,*
enter first payment INITIAL SCREEN
DISPLAY CHANGE ^{*def*} = *tender cash DISPLAY TOTAL PAY*
DISPLAY TOTAL PAY ^{*def*} = *sum up PAYMENT ENTRY*
ERROR STATE ^{*def*} = *reset PAYMENT ENTRY, reset DISPLAY TOTAL PAY*

Petri-nets: Petri-nets are bipartite directed graphs -directed graphs that have edges only between nodes that are in two different sets (place and transitions). The idea that a place can still be enabled after firing a transition makes this model able to model asynchronous systems, the idea that two different transitions can be fired simultaneously makes it able to model concurrent and distributed systems. The tokens that determine if the place nodes are enabled means a variety in different system applications just as how it could mean the number of process that the UNIX operating system is allowed to let alive, it could mean the number of resources a process is allowed to be allocated (modelling resource allocation). Deadlock, a common concept with operating system could also be modelled with Petri-nets. The nature of the Operating system even makes it to want to be taught in universities with Petri-net models [Jeffrey 1991]. An incremental approach to modelling concurrent systems using a model other than Petri-nets is found in [Koppol and Tai 1996].

Different models are used for different software systems. The nature of the system, the aspect of the system to be modelled, the amount of knowledge about the model and the entities that will be using the model are factors to be considered before selecting a model.

After identifying test cases, the next approach is to make selection. The next section deals with that from a functional approach.

4. Functional Strategy

The functional approach to software testing is the common approach to testing software at the system level. This approach is made up of the inputs and outputs of the software as well as knowledge of how the software is expected to behave. The software is tested with certain inputs, and from the appraisal of the corresponding output, one can infer if the software is behaving or misbehaving. A calculator that returns a 7 for the sum of 3 and 6 where a 9 is expected definitely has a fault. To find all the faults with a software system using this approach could involve testing all the inputs that the software will be passed throughout its lifetime; this approach is practically impossible as there isn't time for that and we are yet to be able to see the future. In this section, different methodologies that are used to currently test data inputs (test cases) that would leave one with a high level of confidence that the software system is behaving appropriately are visited. These methodologies are discussed in categories of the basic concepts that are involved during this approach of testing – data, events and port [Jorgensen 2002].

4.1 Data-based

Sophisticated data-driven systems are mostly known to work with persistent data (database). However, some software system carry out their function with a little bit of data passed to them as parameters. Hence, even if concentration in this subsection will be on the functional strategy to select test cases that are basically necessary to test a data-driven system from the view of persistent data, it is worth mentioning few methodologies used to derive test cases from systems that could be data-driven by a parametric, volatile data.

Equivalence Partitioning: The data's input domain is partitioned into equivalence classes and a representative (a random member of the class) of each class stands for its class in the test execution. Any misbehaviour of the system as a result of the

execution of a class' representative is the response the system will give to any other member of the class.

Boundary-Value Analysis: A specification of the equivalence partitioning. This method makes sure that the representative of each class is chosen so that they are values at the boundary of each class.

Cause-effect graph: A logical Boolean graph that shows the relationship between the causes and effect (input and output conditions drawn out from a careful observation of the system's specification). The nodes represents the causes and effects; the thickness of the edges between a cause node and an effect node depicts how much of an effect will result from a cause. [Nursimulu and Probert 1995] is in detail about this method.

Category-Partition Method: A method that has an edge over the equivalence partitioning method and as well the boundary-value analysis method. This method also considers input combinations of system functions with more than one parameter. The strategy is as follows

- Each functional unit in the system that can function separately is identified.
- For each of this functional unit, identify the parameters, constraints and environmental effects.
- Identify the characteristics of each parameter (known as its category).
- Classify each category into distinct classes (known as choices) by equivalence partitioning.
- Combine the representative of each choice, taking the constraints external factors from the environment into consideration e.g. a parameter might not be allowed to be passed in combination with another.

More details on this approach are found in [Ostrand and Balcer 1988].

Another elegant approach similar to this is a method tailored to applications involving multiple inputs and outputs. This approach is based on the principle that not all input parameters will affect a given output. On performing an analysis on the input and output values for a given set of data, the relationship existing between an output and a set of

inputs that affect the given output helps limit the amount of unnecessary test cases selected [Schroeder and Korel 2000]. Reduced test case algorithm [Harold et al 1993] is also a brilliant approach.

More sophisticated, large and complex systems that are data-driven, model the database with the entity-relationship model. To be concrete in this subsection, explanation will go on side by side with the figure below.

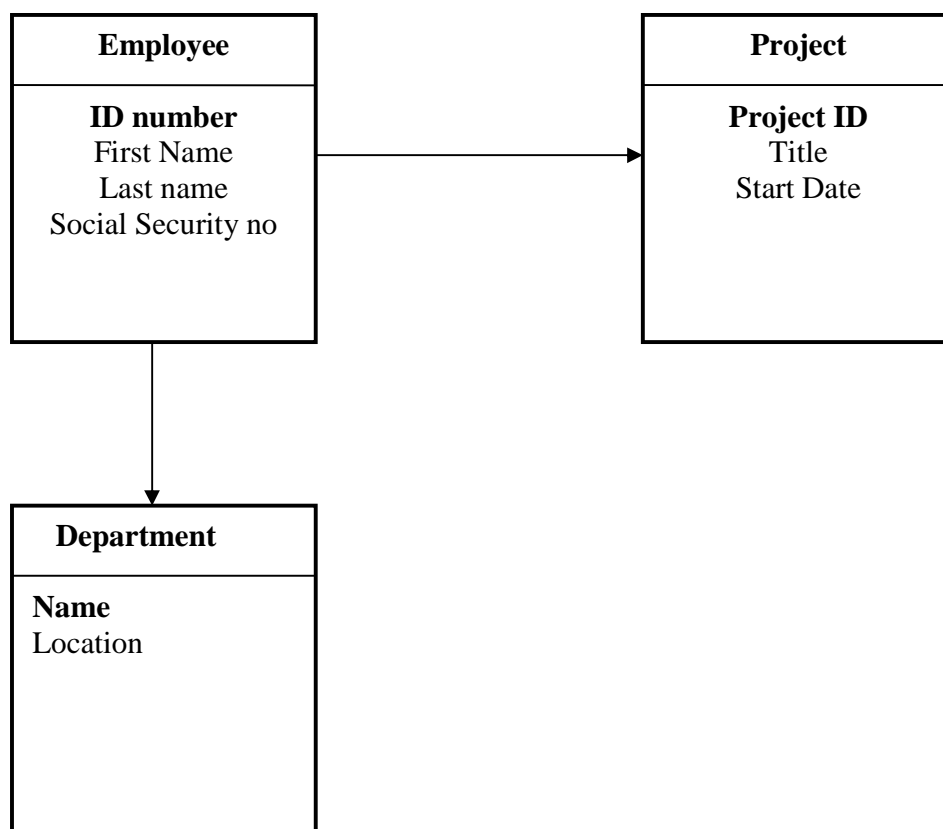


Figure 4.1: An Entity-Relationship model of employee, project and department

The relationships existing among entities in this diagram is

Each employee must to only one department; a department must contain a maximum of 50 employees

A department must hold at least an employee.

Each project must be ran by at least 2 employees and at most 10 employees

An employee can partake in running at most 2 projects and at least none.

The coverage metrics in this case are the cardinality – maximum number of entities partaking in a relationship, participation – minimum number of entities allowed to partake in a relationship and the functional dependencies – does a relationship depend on another?

From the figure above, the coverage under the metric of cardinality is meant to elicit test cases to test that only a maximum of 10 employees runs a project, a department can only contain 50 employees, a 51st employee is looking for trouble (stress testing), a department cannot exist if it doesn't hold at least 1 employee and only a maximum of 2 projects can be run by an employee.

The coverage under the metric of participation is meant to elicit test cases to test that the least amount of employees running a project is 2, every employee must belong to a department, an employee must belong to only one department and there could exist an employee that isn't running any project.

The coverage under the metric of functional dependencies is meant to elicit test cases to test that every employee that runs a project must belong to a department as the employee's number in the department is what is used for identification in the project assigned.

4.2 Event-based

Testing the functionality of a software system has a lot to do with the events that takes place at the point of interaction one has with the system (port). 5 coverage metrics are considered when the event is an input and 2 coverage metrics when it is an output. A brief discussion of each of them will follow them as they are mentioned.

The first coverage metric is that the system be tested for each input event that occurs at a given port. This metric is to see that the system functions as expected when in normal

use. Hence, given a port, test it for all the input events that it expects in order to act normally. For instance, when a system expects an Enter key or a Cancel key, test that point of interaction with those keys.

The second coverage metric is that the system be tested for common inputs at a given port. This metric is difficult to quantify. It could depend on the environment in which this system will be put to use. For example, a hot drink vending machine that is in use by night-shift workers should be tested more with the coffee or espresso vending button – talking from experience there.

The third coverage metric is that the system be tested for events that are relevant to a given context. The function of a Select button on the display of a particular screen could differ from its function on the display of another screen.

The fourth coverage metric is that the system be tested to see how it will respond to inappropriate events occurring in a given context. What will the system do when a Select button is pressed while it expects an enter button. Test cases elicited using coverage of this metric is usually intended to test the system to break, in cases of real-time systems, to miss their deadlines.

The last coverage metric on input events is to test the system for all possible events that the system could experience given a particular input. This time around one is set to break the system a great deal – the system developer isn't our friend in this case.

For port output events that are meant to occur at a given port is tested for. This is a test for the normal and expected performance of the system. For example, to make sure that an ATM machine returns a screen listing choices for service when your PIN number is verified. The second coverage metric for output events is to test that each port output events occurs for each cause. It won't be nice for a file management system to tell one that a file is deleted when it has just been moved.

4.3 Port-based

Testing based on the port of a system is usually as a complement to that of the event-based testing. The process involves considering all the events that can occur at a given port of the system and selecting test cases that will be used to run a test to see that each

event at a given port is carried out appropriately. SCENT (Scenario Based Validation Testing) [Ryser and Glinz 1999] is a method that puts this into practice. Scenarios are created for each port, formalized into statecharts and then test cases are generated from the statechart model.

5. Structural Strategy

The structural approach to system testing is about being able to select adequate test cases with information about the structure of the system. In this section, the node and edge metrics will be looked at and the hierarchical relationship relating Figure 5.1 to Figure 3.1. The section will be closed up by looking at structural testing with use cases.

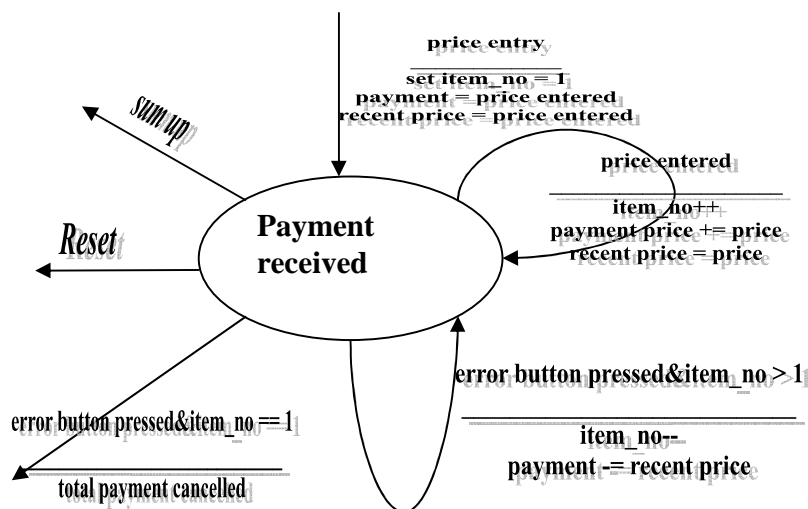


Figure 5.1: A detail of Payment Entry state in Figure 3.1 at a lower level in hierarchy

Testing at the system level involves running test cases to test functions that tend to show their behaviour or misbehaviour with an action visible at the system level. For instance, an error button clicked in the cash register during payment entry will show the “ERR”

notation on the cash screen. Based on the belief that system testers should know how the system is implemented, they can derive a sequence of input events that will provoke that action. Different actions could be provoked using test cases derived from the knowledge of the system's implementation at any level. These test cases can then be selected as adequate ones with which the system test will be executed.

5.1 Node and Edge Metrics

The nodes and edges in the graphical representation of finite state machines are the metrics for measuring how adequate a set of paths are in testing the system for its requirement. The coverage criterion for the nodes in the graph is:

Given a graph G , with node N , a test suite T , test case t , T is adequate for test executions if for every $N \in G$ there exist a $t \in T$ that causes a transition through N .

From Figure 3.1, $T1 = \{\text{enter first payment} \rightarrow \text{sum up} \rightarrow \text{Reset} \rightarrow \text{Reset}, \text{enter first payment} \rightarrow \text{tender cash} \rightarrow \text{Reset}\}$ satisfies this criterion.

The coverage criterion through edges in the graph is:

Given a graph G , edges E , a test suite T , test case t , T is adequate for test executions if for every $E \in G$ there exist a $t \in T$ that causes a transition through E

From Figure 3.1, $T2 = \{\text{enter first payment} \rightarrow \text{sum up} \rightarrow \text{reset} \rightarrow \text{reset}, \text{enter first payment} \rightarrow \text{sum up} \rightarrow \text{tender cash} \rightarrow \text{reset}, \text{enter first payment} \rightarrow \text{reset} \rightarrow \text{reset}, \text{enter first payment} \rightarrow \text{sum up} \rightarrow \text{tender cash} \rightarrow \text{enter first payment} \rightarrow \text{cancel total payment}\}$ satisfies this criterion.

A hierarchical model of a system, down to the lowest level where an atomic system function can be noticed, is an adequate model from which test cases can be selected. A sequence of input or actions that cause transitions between inter-level states would lead to state explosion and an unsystematic approach to selecting test cases. Figure 3.1 is a

detailed form of the **Payment Entry** state in Figure 3.1. Test cases that cover the test requirements of the system under test are first selected from Figure 5.1. After this, moving to the upper level in hierarchy is next (Figure 3.1). In Figure 5.1 the node coverage is done immediately by the **price entry** action on entering that level, realizing that the node coverage is not really as standard as transition coverage.

As noticed, there are loops in Figure 5.1; considering only the first loop for simplicity, the first transition path in the coverage above doesn't give through the loop, in analogy to testing for the loop's condition for execution being false before control gets to it. The second goes through the loop, in analogy to testing for the persistence of the value of a loop invariant and the post-condition value of the loop approaching its value iteration.

The three transition paths in the transition coverage satisfies the transition coverage criteria that every path be transmitted at least once. There are limitations to executing the loop as this is one of the graph construct that brings about combinatorial explosion.

Combinatorial explosion is still bound to pop out a face as the size of the system that needs to be placed under a system test gets more complex and larger. As the states and edges increase, the test cases will also increase. Test constraint is an approach that is used to attack this problem. A state that cannot be reached because it is forbidden for the testing objective forbids other states only reachable from itself even if there is nothing forbidden about those states in the first place. The project state machine coverage [Friedman et al 2002] is a method that uses this strategy. It makes use of the form model, equivalence partitioning and input applicability and constraints.

5.2. Structural Use Cases

Use cases, known to be an element used in the functional approach to testing systems also have information on how they are made to interact with each other internally and externally. With this structural information a structural approach to testing systems using an element for the functional approach is possible [Carniello et al].

The metrics used in this case are the associations between the use cases and the actor and use case. They are: association – relationship between a use case and an actor,

include – use case carrying out a function with another’s help, extend – use case being a specialization of another and generalisation – use case being more abstract of another.

The coverage criteria are:

All-association-inclusion-extension: *Given a test suite T, and a use case diagram D, T must cause each association, inclusion and extension to be exercised at least once.*

All extended combination criterion: *Given a test suite T and a use case diagram D, for each use case in D extended by at least two other use cases, T must cause all of the combinations of exercising and non-exercising extend relationships to be exercised at least once.*

Hence, the functional and structural approach to testing systems goes beyond being complements of each other but can mimic each other.

6. Systematic Guidelines

Here are some systematic guidelines to carrying out system testing.

- System testing is an abstract level of testing, hence, it should be seen that it has a lifecycle of its own. One that begins immediately the requirement of the system to be developed is laid down. TMap [Veenendal and Pol 1997] elaborates on this.
- An appropriate model should be chosen for the kind of system to be modelled or the aspect of the system to be modelled. The developer and testers working with a similar artefact makes one involved in the other’s work.
- A good and clear understanding of the systems requirement should be available to the tester as it is to the developer. Testers should be involved in the requirement elicitation and analysis.
- Manage the test development process as the software development process is managed. Remove obsolete and redundant test cases [Harrold et al 1993], ones that can arise as a result of the modification of software requirements and test requirements that will demand regression testing.

-
- The requirement and design phases of development are good sources for test requirement. Since the selection of abstract test cases using a functional strategy is possible at these stages, it should be done.
 - Inspection and walkthrough, a case where artefacts are inspected for defects, has an effect on system testing [Laitenberger]. It should be practised.
 - Model based testing, an opportunity to automate most of the processes involved in testing development

7. Conclusion and Future Work

Testing at the system level is not an easy task. Its practice even adds to this difficulty. Identifying test cases and selecting the ones adequate based on given criteria are more of an ad-hoc process mainly aimed at detecting the faults for the system under test. Any method that proves to be adequate to make system testing effective so as to improve the system's reliability and worth for its cost of development (including testing) should be applied.

Quantifying the effect an approach to system testing has on the software been developed is also a problem [Gittens et al, 2002]. Hence, standardizing the method that is good enough for a given kind of system is still a stagger. This is the part of system testing I am interested in and aim at a quantified measurement of system testing methods in order to standardize appropriate testing methods to each kind of systems.

References and Bibliographies

- Carniello, A. , Jino, M., Chaim, M. L., Structural Testing with Use Cases *National Institute for Space Research, Applied Computing Program*. [online]. Available from: wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER04/Adriana_Carniello.pdf [January 11, 2007].

-
- Chillarege, R. 1999. Software Testing Best Practices. *Centre of Software Engineering IBM Research*[online]. Available from: www.chillarege.com/authwork/papers1990s/TestingBestPractice.pdf [January 11, 2007]
 - Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. 1999. Model-based testing in practice. In *Proceedings of the 21st international Conference on Software Engineering* (Los Angeles, California, United States, May 16 - 22, 1999). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 285-294.
 - Friedman, G., Hartman, A., Nagin, K., and Shiran, T. 2002. Projected state machine coverage for software testing. In *Proceedings of the 2002 ACM SIGSOFT international Symposium on Software Testing and Analysis* (Roma, Italy, July 22 - 24, 2002). ISSTA '02. ACM Press, New York, NY, 134-143. DOI= <http://doi.acm.org/10.1145/566172.566192>
 - Gittens, M., Lutfiyya, H., Bauer, M., Godwin, D., Kim, Y. W., and Gupta, P. 2002. An empirical evaluation of system and regression testing. In *Proceedings of the 2002 Conference of the Centre For Advanced Studies on Collaborative Research* (Toronto, Ontario, Canada, September 30 - October 03, 2002). D. A. Stewart and J. H. Johnson, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 3.
 - Harrold, M. J., Gupta, R., and Soffa, M. L. 1993. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (Jul. 1993), 270-285. DOI= <http://doi.acm.org/10.1145/152388.152391>
 - Jeffrey, J. M. 1991. Using Petri nets to introduce operating system concepts. In *Proceedings of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education* (San Antonio, Texas, United States, March 07 - 08, 1991). SIGCSE '91. ACM Press, New York, NY, 324-329. DOI= <http://doi.acm.org/10.1145/107004.107074>
 - Jorgensen, P. C. 2002. *Software testing – A Craftsman’s Approach*. (2nd ed). Grand Valley State University, Michigan, USA.
 - Kit, E. 1995. *Software Testing in the real world: improving the process*. Wokingham: Addison-Wesley Pub Co,1995
 - Koppol, P. V. and Tai, K. 1996. An incremental approach to structural testing of concurrent software. In *Proceedings of the 1996 ACM SIGSOFT international Symposium on Software Testing and Analysis* (San Diego,

California, United States, January 08 - 10, 1996). S. J. Zeil and W. Tracz, Eds. ISSTA '96. ACM Press, New York, NY, 14-23. DOI=<http://doi.acm.org/10.1145/229000.226298>

- Laitenberger, O. Studying the effect of Code Inspection and Structural Testing on Software quality. *Fraunhofer Institute for Experimental Software Engineering*. Available from: [January 12, 2007]
- Larsen, K. G., Mikucionis, M., Nielsen, B., and Skou, A. 2005. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM international Conference on Embedded Software* (Jersey City, NJ, USA, September 18 - 22, 2005). EMSOFT '05. ACM Press, New York, NY, 299-306. DOI= <http://doi.acm.org/10.1145/1086228.1086283>
- Merzky, A., Schintke, F., Schutt, T. Requirement Analysis. *Data management and Visualisation*. [online]. Available from: www.gridlab.org/Resources/Deliverables/D8.1b.pdf [January 12, 2007].
- Nursimulu, K. and Probert, R. L. 1995. Cause-effect graphing analysis and validation of requirements. In *Proceedings of the 1995 Conference of the Centre For Advanced Studies on Collaborative Research* (Toronto, Ontario, Canada, November 07 - 09, 1995). K. Bennet, M. Gentleman, H. Johnson, and E. Kidd, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 46.
- Omar, A. A. and Mohammed, F. A. 1991. A survey of software functional testing methods. *SIGSOFT Softw. Eng. Notes* 16, 2 (Apr. 1991), 75-82. DOI= <http://doi.acm.org/10.1145/122538.122551>
- Ostrand, T. J. and Balcer, M. J. 1988. The category-partition method for specifying and generating functional tests. *Commun. ACM* 31, 6 (Jun. 1988), 676-686. DOI= <http://doi.acm.org/10.1145/62959.62964>
- Patton, R. 2000. *Software Testing* Hemel Hempstead: Prentice Hall 2000
- Petschenik, N. H. 1985. Building awareness of system testing issues. In *Proceedings of the 8th international Conference on Software Engineering* (London, England, August 28 - 30, 1985). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 182-188.
- Ryser, J., Glinz, M. 1999. A Practical Approach to Validating and Testing Software Systems Using Scenarios *Department of Computer Science, University of Zurich*. [online] Available from: www.ifi.unizh.ch/groups/req/ftp/papers/QWE99_ScenarioBasedTesting.pdf [January 11, 2007].

-
- Schroeder, P. J. and Korel, B. 2000. Black-box test reduction using input-output analysis. In *Proceedings of the 2000 ACM SIGSOFT international Symposium on Software Testing and Analysis* (Portland, Oregon, United States, August 21 - 24, 2000). M. J. Harold, Ed. ISSTA '00. ACM Press, New York, NY, 173-177. DOI= <http://doi.acm.org/10.1145/347324.349042>
 - Veenendal, V. V., Pol, M. 1997. A Test Management Approach for Structured Testing. *Software Product Quality*. [online]. Available from: www.improveqs.nl/pdf/tmap.pdf [January 12, 2007].
 - Whalen, M. W., Rajan, A., Heimdahl, M. P., and Miller, S. P. 2006. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 international Symposium on Software Testing and Analysis* (Portland, Maine, USA, July 17 - 20, 2006). ISSTA '06. ACM Press, New York, NY, 25-36. DOI= <http://doi.acm.org/10.1145/1146238.1146242>
 - Yücesan, E. 1990. Analysis of Markov chains using simulation graph models. In *Proceedings of the 22nd Conference on Winter Simulation* (New Orleans, Louisiana, United States, December 09 - 12, 1990). O. Balci, Ed. Winter Simulation Conference. IEEE Press, Piscataway, NJ, 468-471.