ISSN (Online): 2230-7893
www.IJCEM.org

# Validating and Testing Software System using Use Case Based Approach

**[1]Suman Kasnia, [2]Deepa Mehta**

**[1]Department of Computer Science, S.S.G.Collage**
**Sirsa, Haryana, India**
***sumankasnia@gmail.com***

**[2]Department of Technical Education, G.P.N. C.**
**Sirsa, Haryana, India**
***deepa.mehta15@gmail.com***

**Abstract**

Use cases are a means to capture a system's functionality and behavior in a user-centered perspective. Thus they are used in most modern object-oriented software development methods to help elicit and document user requirements. Scenarios (Use Cases) also form a kind of abstract level test cases for the system under testing yet they are seldom used to derive concrete system test cases. Here we will find a procedure to use scenarios in a defined way to systematically derive test cases for system test. This is done by formalization of natural language scenarios into state charts, annotation of state charts with helpful information for test case creation/generation and by path traversal in the state charts to determine concrete test cases.

**Keywords:** *Scenario, Performance UCB Testing, SuD (System under Discussion), State Chart*

## 1. Introduction

Testing plays an important role in validating and verifying systems. Yet test preparation and the development of test cases is often done only just before testing starts, at the end of the development process, even though analysis as well as design would greatly profit from the insight gained by developers in creating test cases and preparing tests. Moreover, testing is often done in an ad-hoc manner, and test cases are quite often developed in an unstructured, non-systematic way. This is mainly due to the reality of commercial software development (only limited resources are available and only sparse resources are allocated to testing) and less to lack in available methods or lacking problem understanding. Any testing strategy has to address this practical issue if it is to be successfully applied. To improve testing in practice, systematic test case development and integration of test development methods with 'normal' system development methods is central. Test cases are only developed in a systematic way if clearly defined methods are applied. Test development methods will only be used if they are easy to apply, blend into existing development methods and do not impose an inappropriate overhead or intolerable cost. Many strategies and approaches to testing exist. Besides established techniques like control and data flow testing or boundary analysis/domain testing, formal languages for specification and specialized testing languages are gaining increased attention. Yet a gap is opening between the state of the art and the state of practice. The gap in-between what theoretically could be done and what really is done in practice, is mainly due to the following reasons

### 1.1 Unrealistic expectations

Our industry is known for latching onto any new technical solution and thinking it will solve all of our current problems. Testing tools are no exception. There is a tendency to be optimistic about what can be achieved by a new tool. It is human nature to hope that this solution will at last solve all of the problems we are currently experiencing.

### 1.2 Poor testing practice

If testing practice is poor, with poorly organized tests, little or inconsistent documentation, and tests that are not very good at finding defects, automating testing is not a good idea. It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing.

### 1.3 Expectation that automated tests will find a lot of new defects

A test is most likely to find a defect the first time it is run. If a test has already run and passed, running the same test again is much less likely to find a new defect, unless the test is exercising code that has been changed or could be affected by a change made in a

different part of the software, or is being run in a different environment. Test execution tools are 'replay' tools, i.e. regression testing tools. Their use is in repeating tests that have already run. This is a very useful thing to do, but it is not likely to find a large number of new defects, particularly when run in the same hardware and software environment as before. Tests that do not find defects are not worthless, even though good test design should be directed at trying to find defects. Knowing that a set of tests has passed again gives confidence that the software is still working as well as it was before, and that changes elsewhere have not had unforeseen effects.

### 1.4 False sense of security

Just because a test suite runs without finding any defects, it does not mean that there are no defects in the software. The tests may be incomplete, or may contain defects themselves. If the expected outcomes are incorrect, automated tests will simply preserve those defective results indefinitely.

### 1.5 Maintenance of automated tests

When software is changed it is often necessary to update some, or even all, of the tests so they can be re-run successfully. This is particularly true for automated tests. Test maintenance effort has been the death of many test automation initiatives. When it takes more effort to update the tests than it would take to re-run those tests manually, test automation will be abandoned.

### 1.6 Technical problems

Interoperability of the tool with other software, either your own applications or third-party products, can be a serious problem. The technological environment changes so rapidly that it is hard for the vendors to keep up. Many tools have looked ideal on paper, but have simply failed to work in some environments.

### 1.7 Organizational issues

Automating testing is not a trivial exercise, and it needs to be well supported by management and implemented into the culture of the organization. Time must be allocated for choosing tools, for training, for experimenting and learning what works best, and for promoting tool use within the organization. An automation effort is unlikely to be successful unless there is one person who is the focal point for the use of the tool, the tool 'champion'. Whenever a new tool (or indeed any new process) is implemented, there are inevitably adjustments that need to be made to adapt to new ways of working, which must be managed.

## 2. Testing performance

Performance is (1) the act of performing, for instance, execution, accomplishment, fulfillment, and so on; and (2) operation or functioning, usually with regard to effectiveness.

Finally, since testers are the first expert users of a product, their questions, working notes, and instructions usually form the foundation of the user guide and seed for frequently asked questions documentation. It is a sad waste when the documentation creation process does not take advantage of this resource.

## 3. Testing Metrics

There are well-defined quantified metrics and metrics systems available today in software testing. Unfortunately, many of these metrics are not fundamental measures; they are complex and often obscure. Function points [Jones 1995] and McCabe's Complexity [McCabe 1989] are examples. It is not immediately apparent from these names how to use these metrics, what they measure, or what benefits they offer. Special knowledge is required. Acquiring this knowledge requires an expenditure of time and resources. Fundamental testing metrics are the ones that can be used to answer the following questions.

How big is it?
How long will it take to test it?
How much will it cost to test it?
How much will it cost to fix it?

The question "How big is it?" is usually answered in terms of how long it will take and how much it will cost. These are the two most common attributes of it. We would normally estimate answers to these questions during the planning stages of the project. These estimates are critical in sizing the test effort and negotiating for resources and budget.

We have heard the following fundamental metrics discounted because they are so simple, but in my experience, they are the most useful:

- Time
- Cost
- Tests
- Bugs found by testing

We quantify "How big it is" with these metrics. These are probably the most fundamental metrics specific to software testing. They are listed here in order of decreasing certainty. Only time and cost are clearly defined using standard units. Tests and bugs are complex and varied, having many properties. They can be measured using many different units.

### 3.1 Time

Units of time are used in several test metrics, for example, the time required to run a test and the time available for the best test effort

ISSN (Online): 2230-7893
www.IJCEM.org

### 3.1.1 The Time Required to Run a Test

This measurement is absolutely required to estimate how long a test effort will need in order to perform the tests planned. It is one of the fundamental metrics used in the test inventory and the sizing estimate for the test effort. The time required to conduct test setup and cleanup activities must also be considered. Setup and cleanup activities can be estimated as part of the time required to run a test or as separate items. Theoretically, the sum of the time required to run all the planned tests is important in estimating the overall length of the test effort, but it must be tempered by the number of times a test will have to be attempted before it runs successfully and reliably.

### 3.1.2 The Time Available for the Test Effort

This is usually the most firmly established and most published metric in the test effort. It is also usually the only measurement that is consistently decreasing.

Sample Units: Generally estimated in minutes or hours per test. Also important are the number of hours required to complete a suite of tests. In time available it is generally estimated in weeks and measured in minutes.

### 3.2 The Cost of Testing

The cost of testing usually includes the cost of the testers' salaries, the equipment, systems, software, and other tools. It may be quantified in terms of the cost to run a test or a test suite.

Sample Units**:** Currency, such as dollars; can also be measured in units of time.

### 3.3 Tests

We do not have an invariant, precise, internationally accepted standard unit that measures the size of a test, but that should not stop from benefiting from identifying and counting tests. There are many types of tests, and they all need to be counted if the test effort is going to be measured.

Sample Units:

- A keystroke or mouse action
- An SQL query
- A single transaction
- A complete function path traversal through the system
- A function-dependent data set

### 3.4 Bugs

Many people claim that finding bugs is the main purpose of testing. Even though they are fairly discrete events, bugs are often debated because there is no absolute standard in place for measuring them.

Sample Units: Severity, quantity, type, duration, distribution, and cost to find and fix.

## 4. Plan the Test Environment

The test environment comprises all of the elements that support the physical testing effort, such as test data, hardware, software, networks, and facilities. Test-environment plans must identify the number and types of individuals who require access to the test environment, and specify a sufficient number of computers to accommodate these individuals. Consideration should be given to the number and kinds of environment-setup scripts and test-bed scripts that will be required.

After gathering and documenting the facts as described above, the test team must compile the following information and resources preparatory to designing a test environment: Obtain descriptions of sample customer environments, including a listing of support software, computer hardware and operating systems. Determine whether the test environment requires an archive mechanism.

Identify network characteristics of the customer environment. In the case of a client-server or Web-based system, identify the required server operating systems, databases, and other components. Identify the number of automated test tool-licenses required by the test team. Identify other software needed to execute certain test procedures, consider test at requirements including their size.

## 5. Risks During Testing

1.1.1 Test-program assumptions, prerequisites, and risks must be understood before an effective testing strategy can be developed. This includes any events, actions, or circumstances that may prevent the test program from being implemented or executed according to schedule, such as late budget approvals, delayed arrival of test equipment, or late availability of the software application.

Test strategies generally must incorporate ways to minimize the risk of cost overruns, schedule slippage, critical software errors, and other failures. During test-strategy design, constraints on the task at hand, including risks, resources, time limits, and budget restrictions, must be considered.

A test strategy is best determined by narrowing down the testing tasks as follows:

- Understand the system architecture.
- Determine whether to apply GUI testing, back-end testing, or both.
- Select test-design techniques.
- Select testing tools.
- Develop in-house test harnesses or scripts.
- Determine test personnel and expertise required.
- Establish release criteria. Stating the testing coverage is closely related to defining release or exit criteria.
- Set the testing schedule.

ISSN (Online): 2230-7893
www.IJCEM.org

## 6. UCB Testing Approach

### 6.1 Test case

"A test case has components that describe an input, action or event and an expected response, to determine if a feature of an application is working correctly."
There are levels in which each test case will fall in order to avoid duplication efforts.

Level 1: write the basic test cases from the available specification and user documentation.

Level 2: This is the practical stage in which writing test cases depend on actual functional and system flow of the application.

Level 3: This is the stage in which you will group some test cases and write a test procedure. Test procedure is nothing but a group of small test cases maximum of 10.

Level 4: Automation of the project. This will minimize human interaction with system and thus Quality Assurance can focus on current updated functionalities to test rather than remaining busy with regression testing.

So, the basic objective of writing test cases is to validate the testing coverage of the application.

### 6.2 Use case

Use cases are a software modeling technique that helps developers determine which features to implement and how to gracefully resolve errors. Within systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals.

#### 6.2.1 Use cases and the development process

The specific way use cases are used within the development process will depend on which development methodology is being used. In certain development methodologies a brief use case survey is all that is required. In other development methodologies use cases evolve in complexity and change in character as the development process proceeds. Use cases can be a valuable source of usage information and usage testing ideas. In some methodologies, they may begin as brief business use cases, evolve into more detailed system use cases, and then eventually develop into highly detailed and exhaustive test cases...

#### 6.2.2 Use case focus

"Each use case focuses on describing how to achieve a goal or a task. For most software projects, this means that multiple, perhaps dozens of use cases are needed to define the scope of the new system. The degree of formality of a particular software project and the stage of the project will influence the level of detail required in each use case.

Use cases should not be confused with the features of the system. One or more features describe the functionality needed to meet a stakeholder request or user need each feature can be analyzed into one or more use cases.

Use cases treat the system as a black box, and the interactions with the system, including system responses, are perceived as from outside the system. This is a deliberate policy, because it forces the author to focus on what the system must do, not how it is to be done, and avoids making assumptions about how the functionality will be accomplished.

A use case should:

- Describe what the system shall do for the actor to achieve a particular goal.
- Include no implementation-specific language.
- Be at the appropriate level of detail.
- Not include detail regarding user interfaces and screens. This is done in user-interface design, which references the use case and its business rules.

Use cases are mostly text documents, and use case modeling is primarily an act of writing text and not drawing diagrams. Use case diagrams are secondary in use case work.

#### 6.2.3 Degree of detail

Three levels of detail in writing use cases:

- Brief use case -- consists of a few sentences summarizing the use case. It can be easily inserted in a spreadsheet cell, and allows the other columns in the spreadsheet to record priority, duration, a method of estimating duration, technical complexity, release number, and so on.
- Casual use case -- consists of a few paragraphs of text, summarizing the use case.
- Fully dressed use case -- a formal document based on a detailed template with fields for various sections; and it is the most common understanding of the meaning of a use case. Fully dressed use cases are discussed in detail in the next section on use case templates.

#### 6.2.4 Actor

An actor specifies a role played by a person or thing when interacting with the system. The same person using the system may be represented as different actors because they are playing different roles. The actors in the system are the passenger, the counter clerk and the reservation system consisting of form processing, reservation, fare computation, ticket processing, ticket printing, collection of fare amount and posting as sub-systems.

#### Types of Actors

- Primary Actor - Fulfills the user goals by using the services of the SuD (System Under Discussion). Ex: 'Booking Clerk' in Railway Reservation System.
- Secondary Actor - Provides a service to the SuD. Ex: 'user of database system'.

ISSN (Online): 2230-7893
www.IJCEM.org

- Offstage Actor - Has an interest in the behavior of the system but is not primary or secondary. Ex: A government tax agency.

### 6.2.5 Limitations of use Cases

Use case flows are not well suited to easily capturing non-interaction based requirements of a system (such as algorithm or mathematical requirements) or non-functional requirements (such as platform, performance, timing, or safety-critical aspects). These are better specified declaratively elsewhere.

- Use case templates do not automatically ensure clarity. Clarity depends on the skill of the writer(s).
- There is a learning curve involved in interpreting use cases correctly, for both end users and developers. As there are no fully standard definitions of use cases, each group must gradually evolve its own interpretation. Some of the relations, such as *extends*, are ambiguous in interpretation and can be difficult for stakeholders to understand.
- Proponents of Extreme Programming often consider use cases too needlessly document-centric, preferring to use the simpler approach of a user story.
- Use case developers often find it difficult to determine the level of user interface (UI) dependency to incorporate in a use case.
- Use cases can be over-emphasized. In Object Oriented Software.
- Use cases have received some interest as a starting point for test design. Some use case literature, however, states that use case pre- and post conditions should apply to all scenarios of a use case (i.e., to all possible paths through a use case) which is limiting from a test design standpoint
- Some systems are better described in an information/data-driven approach than in a the functionality-driven approach of use cases. A good example of this kind of system is data-mining systems used for Business Intelligence.

## 7. State Chart Diagram

A state diagram is used to describe the behavior of systems. State chart diagrams represent the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event in stances. Typically, it is used for describing the behavior of classes, but state charts may also de scribe the behavior of other model entities such as us e-cases, actors, subsystems, operations, or methods.

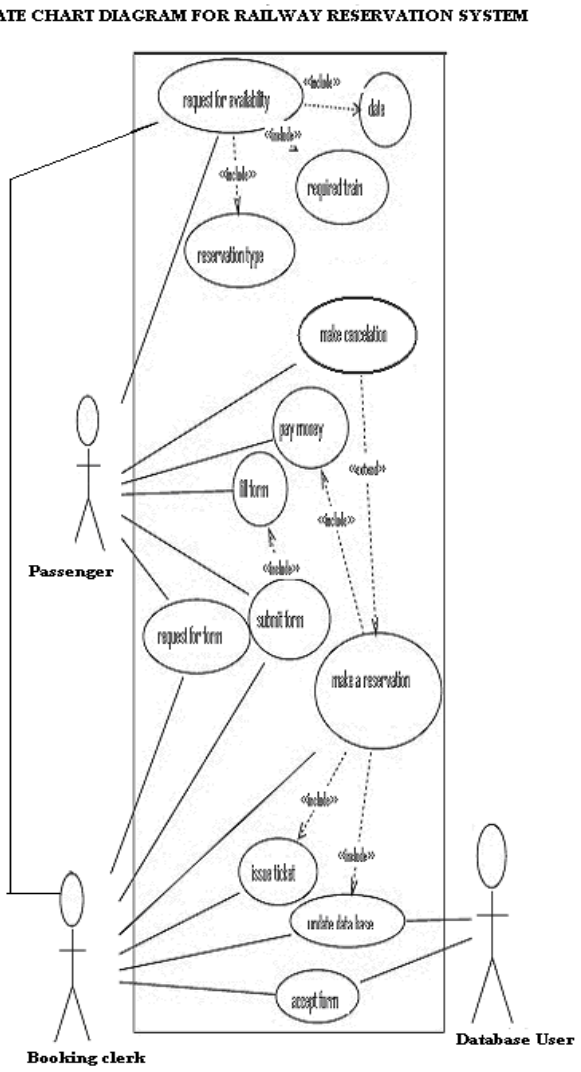


Fig. 1 State Chart Diagram of RRS

| *User* | *Role* | *Use case* |
|---|---|---|
| • Passenger | • Enquiry<br>• Reservation and ticketing<br>• Cancellation | • Enquire ticket availability and other details.<br>• Reserve seats and berths, tickets.<br>• Cancel tickets. |
| • Booking clerk | • Form data entry<br>• Requisition processor<br>• Ticket processor<br>• Data manager | • Enter Reservation Requisition Form.<br>• Process requisition for booking.<br>• Process ticket to print.<br>• Submits ticket data for updation |
| • Reservation and ticketing system (Database user). | • System server | • Process reservation data, process.<br>• Ticketing process cancellation.<br>• Update the status by date, train, etc. |

Table 1: User roles and use Cases for Fig. 1

## 8. Use Case specifications for RRS

8.1 Request for availability –

8.1.1 Brief Description
The main purpose of using this use case is to know details of particular trains available or not .Along with reservation type, date, particular train details are also known.

Flow of Events
8.1.2 Generic flow
- User enquires booking clerk.
- Booking clerk check the database.
- On success Passenger make reservation

8.1.3 Alternate Flow
If in the basic flow, particular trains are not available then he choose any others trains go to destination place.

8.1.4 Pre Conditions
The Passenger should have a train for destination place.

8.1.5 Post Conditions
The reservation database is modified after reservation

8.2. Reservation type

8.2.1 Brief Description
The main purpose of using this use case is to know details of particular reservation type i.e. sleeper class, AC class, general etc.

Flow of Events
8.2.2 Generic flow
- User enquires booking clerk.
- Booking clerk checks the database.
- On success Passenger chooses required reservation type.

8.2.3 Alternate Flow
If in the basic flow, required reservation type is not present then he chooses any other reservation type.

8.2.4 Pre Conditions
The Passenger should know reservation type.

8.2.5 Post Conditions
The reservation database is modified after choosing reservation type.

8.3. Date

8.3.1 Brief Description
The main purpose of using this use case is to know details of particular trains available or not on a particular date.

Flow of Events
8.3.2 Generic flow
- User enquires booking clerk.
- Booking clerk check the database.
- On success Passenger make reservation.

8.3.3 Alternate Flow
If in the basic flow.

8.3.4 Pre Conditions
The Passenger should have a train for destination place.

8.3.5 Post Conditions
The reservation database is modified after reservation.

8.4.Request for availability

8.4.1 Brief Description

The main purpose of using this use case is to know details particular trains available or not .

Flow of Events

8.4.2 Basic flow

- User enquires booking clerk.
- Booking clerk check the database.
- On success Passenger make reservation.

8.4.3 Alternate Flow

If in the basic flow.

8.4.4 Pre Conditions

The Passenger should have a train for destination place.

8.4.5 Post Conditions

The reservation database is modified after reservation

## 9. Conclusion

Firstly, we have discussed some problems of software testing. Here, we tried to reduce the gap between the state of the art and the state of practice for this work we have presented the UCB Method, a Use case-based (Scenario) based approach to support the tester of a software system in systematically developing test cases. The narrative scenarios then are formalized using state charts as a notation. The state charts are annotated with information important for testing. Finally, test cases are derived by path traversal of state charts. The method presented here is novel with respect to the synthesis of the aforementioned factors into one single process. We thus supply a method that supports the tester in systematic test case derivation, that uses artifacts of the early phases of the development process in testing again and that handily integrates with existing development methods.

This method has been applied in practice to Railway Reservation System. First experiences are quite promising as the main goal of the method, namely to supply test developers with a practical and systematical way to derive test cases, has been reached. The use of scenarios was perceived by the developers as helpful and valuable in modeling user interaction with the system. On the other hand, scenario management was a major problem throughout the development process. A narrative scenario transformed into a state chart by one developer may differ significantly from a state chart developed from the same scenario by another developer.

Test case creation was unproblematic as the chosen link coverage in state charts is simple, yet powerful.

## 10. References

[1] T. S. Chow: Testing Software Design Modeled by Finite-State Machines; IEEE Transactions on Software Engineering, vol. 4, n¡ 3, pp. 178-187, 1978.

[2] D. C. Firesmith: Modeling the Dynamic Behavior of Systems, Mechanisms and Classes with Scenarios; Report on Object Analysis and Design, vol. 1, n¡ 2, pp. 32-36,47, 1994.

[3] N. E. Fuchs, U. Schwertel, R. Schwitter: Attempto Controlled English - Not Just Another Logic Specification Language; Logic-Based Program Synthesis and Transformation, Eighth International Workshop LOPSTR'98, Manchester, UK, 1999.

[4] G. Gonenc: A Method for the Design of Fault-detection Experiments; IEEE Transactions on Computers, vol. C-19, pp. 551-558, 1970.

[5] D. Harel: State charts: A Visual Formalism for Complex Systems; Science of Computer Programming, vol. 8, pp. 231-274, 1987.

[6] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen: Formal Approach to Scenario Analysis; IEEE Software, vol. 11, n¡ 2, pp. 33-41, 1994.

[7] R. Itschner, C. Pommerell, M. Rutishauser: GLASS: Remote Monitoring of Embedded Systems in Power Engineering; IEEE Internet Computing, vol. 2, n¡ 3, 1998.

[8] Jacobson, M. Christerson, P. Jonsson, G. .vergaard: Object Oriented Software Engineering: A Use Case Driven Approach. Amsterdam: Addison-Wesley, 1992.

[9] Jacobson: Basic Use Case Modeling; Report on Object Analysis and Design, vol. 1, n¡ 2, pp. 15-19, 1994.

[10] S. Pimont, J.C. Rault: A Software Reliability Assessment Based on a Structural Behavioral Analysis of Programs; Proceedings 2nd International Conference on Software Engineering, San Francisco, CA, 1976.

[11] J. Ryser, S. Berner, M. Glinz: On the State of the Art in Requirements-based Validation and Test of Software; University of Zurich, Institut f.r Informatik, Z.rich, Berichte des Instituts f.r Informatik 98.12, Nov 1998.

[12] J. Ryser, M. Glinz: SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test; to appear as a technical report at University of Zurich, Institut f.r Informatik, Z.rich,1999.
[13] J. Ryser, M. Glinz: A Practical Approach to Validating and Testing Software Systems Using Scenarios; Quality Week Europe '99, Brussels, 1999.
[14] S. Som., R. Dssouli, J. Vaucher: Toward an Automation of Requirements Engineering using Scenarios; Journal of Computing and Information, Special issue: ICCI'96, 8th International Conference of Computing and Information, pp. 1110-1132, 1996.
[15] Spence, C. Meudec: Generation of Software Tests from Specifications; SQM'94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, 1994.
[16] M. Andersson, J. Bergstrand, "Formalizing Use Cases with Message Sequence Charts," in Lund Institute of Technology: Lund, 1995.
[17] M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt, "Survey on the Scenario Use in Twelve Selected Industrial Projects," GI Fachgruppe 2.1.6 Requirements Engineering, Aachener Informatik-Berichte 98-07, June 1998.
[18] B. Beizer, Software Testing Techniques, Second Edition ed. New York: Van Nostrand Reinhold, 1990.
[19] B. Beizer, Black-Box Testing, Techniques for Functional Testing of Software and Systems. New York: John Wiley & Sons, 1995.
[20] B. Boehm, Software Engineering Economics. Englewood Cliffs, N.J.: Prentice Hall, 1981.
[21] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," IEEE Transactions on Software Engineering, vol. 4, # 3, pp. 178-187, 1978.
[22] D. C. Firesmith, "Modeling the Dynamic Behavior of Systems, Mechanisms and Classes with Scenarios," Report on Object Analysis and Design, vol. 1, # 2, pp. 32-36,47, 1994.
[23] M. Glinz, "An Integrated Formal Model of Scenarios Based on State charts", in W.Schäfer, and P.Botella, (eds.) Software Engineering - ESEC '95. Proceedings of the 5th European Software Engineering Conference, Sitges, Spain. Springer, Berlin (Lecture Notes in Computer Science 989), pp. 254 - 271, 1995.
[24] G. Gonenc, "A Method for the Design of Fault-detection Experiments," IEEE Transactions on Computers, vol. C-19, pp. 551-558, 1970.
[25] D. Harel, "State charts: A Visual Formalism for Complex Systems," Science of Computer Programming, vol. 8, pp. 231-274, 1987.
[26] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen, "Formal Approach to Scenario Analysis," IEEE Software, vol. 11, # 2, pp. 33-41, 1994.
[27] R. Itschner, C. Pommerell, M. Rutishauser, "GLASS: Remote Monitoring of Embedded Systems in Power Engineering," IEEE Internet Computing, vol. 2, # 3, 1998.
[28] I.Jacobson, M. Christerson, P. Jonsson, G. Övergaard, Object Oriented Software Engineering: A Use Case Driven Approach. Amsterdam: Addison-Wesley, 1992.
[29] Jacobson, "Basic Use Case Modeling," Report on Object Analysis and Design, vol. 1, # 2, pp. 15-19, 1994.
[30] Jacobson, "Basic Use Case Modeling (cont.)," Report on Object Analysis and Design, vol. 1, # 3, pp. 7-9, 1994.
[31] W. J. Lee, S.D. Cha, Y.R. Kwon, "Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering," IEEE Transactions on Software Engineering, vol. 24, # 12, pp. 1115-1130, 1998.
[32] G. J. Myers, The Art of Software Testing. New York: John Wiley & Sons, 1979.
[33] S. Pimont, J.C. Rault, "A Software Reliability Assessment Based on a Structural Behavioral Analysis of Programs," Proceedings 2nd International Conference on Software Engineering, San Francisco, CA, 1976.
[34] C. Potts, K. Takahashi, A.I. Anton, "Inquiry-based Requirements Analysis," IEEE Software, vol. 11, # 2, pp. 21-32, 1994.
[35] B. Regnell, K. Kimbler, A. Wesslén, "Improving the Use Case Driven Approach to Requirements Engineering," Proceedings 2nd International Symposium on Requirements Engineering,York, England, 1995.
[36] Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8:231{274 – Harel – 1987.
[37] Software Engineering Economics – Boehm – 1981.
[38] The Art of Software Testing – Myers – 1979.
[39] Inquiry-based Requirements Analysis – Potts, Takahashi, et al. – 1994.
[40] Testing software design modeled by finite-state machines – Chow – 1978.
[41] Guiding Goal Modeling Using Scenarios – Rolland, Souveyet, et al. – 1998.
[42] A formal approach to scenario analysis – Hsia, Samuel, et al. – 1994.

[43] Object Oriented Software Engineering: a Use Case Driven Approach – Jacobson, Christerson, et al. – 1992.
[44] Supporting scenariobased requirements engineering – Sutcliffe, Maiden, et al. – 1998.
[45] Black-Box Testing : Techniques for Functional Testing of Software and Systems – Beizer.
[46] An integrated formal model of scenarios based on statecharts – Glinz – 1995.
[47] A Hierarchical Use Case Model with Graphical Representation – Regnell, Bergstrand – 1996.
[48] Improving the Use Case Driven Approach to Requirements Engineering – Regnell, Kimbler, et al. – 1995.
[49] A method for the design of fault detection experiments – Gonenc – 1970.
[50] Toward an Automation of Requirements Engineering using Scenarios – Somé, Dssouli, et al. – 1996.
[51] Formalizing use cases with Message Sequence Charts – Andersson, Bergstrand – 1995.
[52] SCENT: A Method Employing Scenarios to Systematically Derive TestCases for System Test – Ryser, Glinz – 2000.
[53] Integration and analysis of use cases using modular Petri nets in requirements engineering – Lee, Cha, et al. – 1998.
[54] 7 GLASS: Remote Monitoring of Embedded Systems in Power Engineering – Itschner, Pommerell, et al. - 1998
[55] Generation of Software Tests from Specifications – Spence, Meudec - 1994
[56] How to design practical test cases – Yamaura – 1998.
[57] A software reliability assessment based on a structural and behavioral analysis of programs – Pimont, Rault – 1976.
[58] 3 Modeling the Dynamic Behavior of Systems, Mechanisms and Classes with Scenarios," Report on Object Analysis and Design – Firesmith – 1994.
[59] Software Testing Techniques, Second Edition ed – Beizer – 1990.
[60] Basic Use Case Modeling; Report on Object Analysis – Jacobson – 1994.
[61] Basic Use Case Modeling (cont.)," Report on Object Analysis – Jacobson – 1994.
[62] On the State of the Art in Requirements-based Validation and Test – Ryser, Berner, et al. – 1998.
[63] http://www.ifi.uzh.ch/groups/req/ftp/.../QWE99_Scenario BasedTesting.pdf.
[64] http://www.softwaretestinglab.co.uk/...testing/common-problems-of-test- automation.
[65] http://www.flylib.com/books/en/1.329.1.35/1.
[66] http://www.softwaretestinglab.co.uk/...testing/fundamental-testing-metrics-bugs.
[67] http://www.softwaretestinglab.co.uk/test-planning/plan-the-test-environment.
[68] http://www.softwaretestinglab.co.uk/category/test-planning.
[69] http://www.asknumbers.com/QualityAssuranceandTesting.aspx.
[70] http://www.softwaretestinghelp.com/how-to-write-effective-test-cases-test-cases- procedures-and-definitions.
[71] http://www .en.wikipedia.org/wiki/Use_case.
[72] http://www.springerlink.com/index/ghm05rmdbrv041k5.pdf.
[73] http://www.scribd.com/doc/.../2909460-Uml-Diagrams.
[74] http://www.karthikk.net.
[75] Bertolino, A. Software Testing Research and Practice, In Proc. of the 10th Int. Wksp on Abstract State Machines (ASM 2003) (Taormina, Italy, March 2003). LNCS 2589, Springer, 2003, 1-21.
[76] Craggs I., Sardis M., and Heuillard T. AGEDIS Case Studies: Model-based Testing in Industry. Proc. 1st European Conf. on Model Driven Softw. Eng. (Nuremberg, Germany, Dec. 2003), imbus AG, 106-117.
[77] Duran, J.W. and Ntafos, S.C. An Evaluation of Random Testing. IEEE Trans. Software Engineering. 10, 4 (1984) 438-444.
[78] Gamma, E., et al., Design Patterns Elements of reusable Object-Oriented Software, Addison-Wesley, Reading, Mass., 1996.
[79] Hutchins, M., et al. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Proc. of the 16th Int. Conf. on Software Eng. (ICSE '94), (Sorrento, Italy, May 1994), 191-200.
[80] ISSTA2002 Panel: Is ISSTA Research Relevant to Industrial Users?. In ACM Proc. of ISSTA 2002 (Roma, Italy, July 2002), 201-209. Juristo, N., Moreno, A.M., and Vegas, S. Reviewing 25 Years of Testing Technique Experiments, Empirical Softw. Eng. J., 9, 1/2 (March 2004), 7-44.
[81] Littlewood, B., et al. Modeling the Effects of Combining Diverse Software Fault Detection Techniques. IEEE Trans. Software Eng., 26, 12, 2000, 1157—1167.
[82] Marick, B. Software Testing Patterns. On line at
[83] http://www.testing.com/test-patterns/index.html (accessed April 21, 2004).
[84] Mohagheghi, P., et al. An Empirical Study of Software Reuse vs. Reliability and Stability. . In Proc. of the 26th Int. Conf. on Softw. Eng.

(ICSE'04) (Edinburgh, Scotland, UK, 23-28 May 2004).

[85] Morasca, S. and Serra-Capizzano, S. On the Analytical Comparison of Testing Techniques. In Proc. of ACM ISSTA 2004, (Boston, Ma, USA, July 2004). ACM Press (to appear).

[86] Ostrand, T.J., Weyuker, E.J. and Bell, R.M. Where the Bugs Are. In Proc. of ACM ISSTA 2004, (Boston, Ma, USA, July 2004). ACM Press (to appear).

[87] Schmidt, D.C., Fayad, M. and Johnson, R.E., Software Patterns. Communications of the ACM, 39, 10 (Oct. 1996), 37-39.

[88] Tichy, W.F. Hints for Reviewing Empirical Work in Software Engineering, Empirical Softw. Eng. J., 5, 4 (Dec. 2000), 309-312.

[89] http://www.sce.carleton.ca/squall/WERST2004/ ((accessed April 21, 2004).

[90] Wood, M., et al. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. In Proc. of ESEC/FSE (Zurich, Switzerland, 1997), LNCS 1301, Springer, 262-277.