Software Testing Techniques

Technology Maturation and Research Strategy

Class Report for 17-939A

Lu Luo
Institute for Software Research International
Carnegie Mellon University
Pittsburgh, PA15232
USA

Software Testing Techniques

Technology Maturation and Research Strategies

Lu Luo
School of Computer Science
Carnegie Mellon University

1 Introduction[1]

Software testing is as old as the hills in the history of digital computers. The testing of software is an important means of assessing the software to determine its quality. Since testing typically consumes 40~50% of development efforts, and consumes more effort for systems that require higher levels of reliability, it is a significant part of the software engineering. With the development of Fourth generation languages (4GL), which speeds up the implementation process, the proportion of time devoted to testing increased. As the amount of maintenance and upgrade of existing systems grow, significant amount of testing will also be needed to verify systems after changes are made [12]. Despite advances in formal methods and verification techniques, a system still needs to be tested before it is used. Testing remains the truly effective means to assure the quality of a software system of non-trivial complexity [13], as well as one of the most intricate and least understood areas in software engineering [19]. Testing, an important research area within computer science is likely to become even more important in the future.

This retrospective on a fifty-year of software testing technique research examines the maturation of the software testing technique research by tracing the major research results that have contributed to the growth of this area. It also assesses the change of research paradigms over time by tracing the types of research questions and strategies used at various stages. We employ the technology maturation model given by Redwine and Riddle [15] as the framework of our studies of how the techniques of software testing first get the idea formulated, preliminarily used, developed, and then extended into a broader solution. Shaw gives a very good framework of software engineering research paradigms in [17], which classifies the research settings, research approaches, methods, and research validations that have been done by software researchers. Shaw's model is used to evaluate the research strategies for testing techniques used in our paper.

2 The Taxonomy of Testing Techniques

Software testing is a very broad area, which involves many other technical and non-technical areas, such as specification, design and implementation, maintenance, process and management issues in software engineering. Our study focuses on the state of the art in testing techniques, as well as the latest techniques which representing the future direction of this area. Before stepping into any detail of the maturation study of these techniques, let us have a brief look at some technical concepts that are relative to our research.

2.1 The Goal of Testing

In different publications, the definition of testing varies according to the purpose, process, and level of testing described. Miller gives a good description of testing in [13]:

> The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.

Miller's description of testing views most software quality assurances activities as testing. He contends that testing should have the major intent of finding errors. A good test is one that has a high probability of finding an as yet undiscovered error, and a successful test is one that uncovers an as yet undiscovered error. This general category of software testing activities can be further divided. For purposes of this paper,

[1] Jointly written by Paul Li

testing is the dynamic analysis of a piece of software, requiring execution of the system to produce results, which are then compared to expected outputs.

2.2 The Testing Spectrum

Testing is involved in every stage of software life cycle, but the testing done at each level of software development is different in nature and has different objectives.

Unit Testing is done at the lowest level. It tests the basic unit of software, which is the smallest testable piece of software, and is often called "unit", "module", or "component" interchangeably.

Integration Testing is performed when two or more tested units are combined into a larger structure. The test is often done on both the interfaces between the components and the larger structure being constructed, if its quality property cannot be assessed from its components.

System Testing tends to affirm the end-to-end quality of the entire system. System test is often based on the functional/requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability, are also checked.

Acceptance Testing is done when the completed system is handed over from the developers to the customers or users. The purpose of acceptance testing is rather to give confidence that the system is working than to find errors.

2.3 Static Analysis and Dynamic Analysis

Based on whether the actual execution of software under evaluation is needed or not, there are two major categories of quality assurance activities:

Static Analysis focuses on the range of methods that are used to determine or estimate software quality without reference to actual executions. Techniques in this area include code inspection, program analysis, symbolic analysis, and model checking.

Dynamic Analysis deals with specific methods for ascertaining and/or approximating software quality through actual executions, i.e., with real data and under real (or simulated) circumstances. Techniques in this area include synthesis of inputs, the use of structurally dictated testing procedures, and the automation of testing environment generation.

Generally the static and dynamic methods are sometimes inseparable, but can almost always discussed separately. In this paper, we mean dynamic analysis when we say testing, since most of the testing activities (thus all the techniques studied in this paper) require the execution of the software.

2.4 Functional Technique and Structural Technique

The information flow of testing is shown in Figure 1. As we can see, testing involves the configuration of proper inputs, execution of the software over the input, and the analysis of the output. The "Software Configuration" includes requirements specification, design specification, source code, and so on. The "Test Configuration" includes test cases, test plan and procedures, and testing tools.

Based on the testing information flow, a **testing technique** specifies the strategy used in testing to select input test cases and analyze test results. Different techniques reveal different quality aspects of a software system, and there are two major categories of testing techniques, functional and structural.

Functional Testing: the software program or system under test is viewed as a "**black box**". The selection of test cases for functional testing is based on the **requirement** or **design specification** of the software entity under test. Examples of expected results, some times are called **test oracles**, include

requirement/design specifications, hand calculated values, and simulated results. Functional testing emphasizes on the **external behavior** of the software entity.

Structural Testing: the software entity is viewed as a "**white box**". The selection of test cases is based on the **implementation** of the software entity. The goal of selecting such test cases is to cause the execution of specific spots in the software entity, such as specific statements, program branches or paths. The expected results are evaluated on a set of **coverage criteria**. Examples of coverage criteria include path coverage, branch coverage, and data-flow coverage. Structural testing emphasizes on the **internal structure** of the software entity.

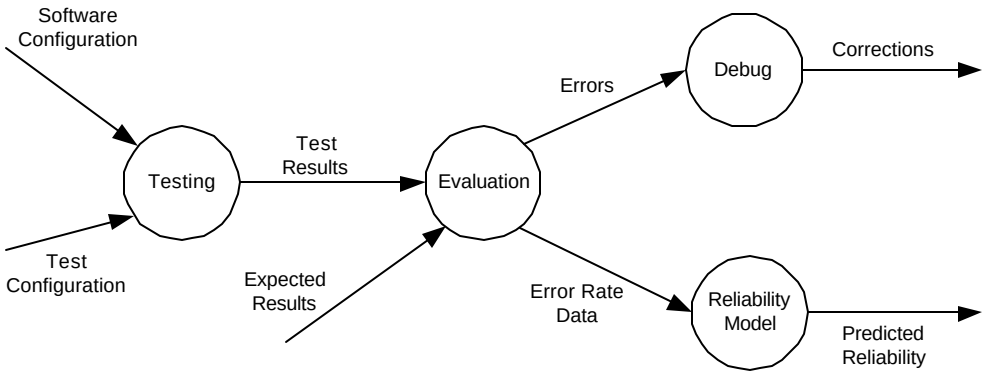

Figure 1. Testing Information Flow

3 Scope of the Study

3.1 Technical Scope

In this paper, we focus on the technology maturation of testing techniques, including these functional and structural techniques that have been influential in the academic world and widely used in practice. We are going to examine the growth and propagation of the most established strategy and methodology used to select test cases and analyze test results. Research in software testing techniques can be roughly divided into two branches: theoretical and methodological, and the growth in both branches push the growth of testing technology together. Inhibitors of maturation, which explains why the in-depth research hasn't brought revolutionary advantage in industry testing practice, are also within our scope of interest.

There are many other interesting areas in software testing. We limit the scope of our study within the range of testing techniques, although some of the areas maybe inseparable from our study. Specifically, we are ***not*** going to discuss:

- How testing is involved in the software development cycle
- How different levels of testing are performed
- Testing process models
- Testing policy and management responsibilities, and
- Stop criteria of testing and software testability

3.2 Goal and standard of progress

The ultimate goal of software testing is to help designers, developers, and managers construct systems with high quality. Thus research and development on testing aim at efficiently performing effective testing – to find more errors in requirement, design and implementation, and to increase confidence that the software has various qualities. Testing technique research leads to the destination of practical testing methods and

tools. Progress toward this destination requires fundamental research, and the creation, refinement, extension, and popularization of better methods.

The standard of progress for the research of testing techniques include:

- Degree of acceptance of the technology inside and outside the research community
- Degree of dependability on other areas of software engineering
- Change of research paradigms in response to the maturation of software development technologies
- Feasibility of techniques being used in a widespread practical scope, and
- Spread of technology – classes, trainings, management attention

4. The History of Testing Techniques

4.1 Concept Evolution

Software has been tested as early as software has been written. The concept of testing itself evolved with time. The evolution of definition and targets of software testing has directed the research on testing techniques. Let's briefly review the concept evolution of testing using the testing process model proposed by Gelperin and Hetzel [6] before we begin study the history of testing techniques.

Phase I. Before 1956: The Debugging-Oriented Period
– Testing was not separated from debugging

In 1950, Turing wrote the famous article that is considered to be the first on program testing. The article addresses the question "How would we know that a program exhibits intelligence?" Stated in another way, if the requirement is to build such a program, then this question is a special case of "How would we know that a program satisfies its requirements?" The operational test Turing defined required the behavior of the program and a reference system (a human) to be indistinguishable to an interrogator (tester). This could be considered the embryotic form of functional testing. The concepts of program checkout, debugging and testing were not clearly differentiated by that time.

Phase II. 1957~78: The Demonstration-Oriented Period
– Testing to make sure that the software satisfies its specification

It was not until 1957 was testing, which was called program checkout by that time, distinguished from debugging. In 1957, Charles Baker pointed out that "program checkout" was seen to have two goals: "Make sure the program runs" and "Make sure the program solves the problem." The latter goal was viewed as the focus of testing, since "make sure" was often translated into the testing goal of satisfying requirements. As we've seen in Figure 1, debugging and testing are actually two different phases. The distinction between testing and debugging rested on the definition of success. During this period definitions stress the purpose of testing is to demonstrate correctness: "An ideal test, therefore, succeeds only when a program contains no errors." [5]

The 1970s also saw the widespread idea that software could be tested exhaustively. This led to a series of research emphasis on path coverage testing. As is said in Goodenough and Gerhart's 1975 paper "exhaustive testing defined either in terms of program paths or a program's input domain." [5]

Phase III. 1979~82: The Destruction-Oriented Period
– Testing to detect implementation faults

In 1979, Myers wrote the book *The Art of Software Testing*, which provided the foundation for more effective test technique design. For the first time software testing was described as "the process of executing a program with the intent of finding errors." The important point was made that the value of test cases is much greater if an error is found. As in the demonstration-oriented period, one might unconsciously select test data that has a low probability of causing program failures. If testing intends

to show that a program has faults, then the test cases selected will have a higher probability of detecting them and the testing is more successful. This shift in emphasis led to early association of testing and other verification/validation activities.

Phase IV. 1983~87: The Evaluation-Oriented Period
– Testing to detect faults in requirements and design as well as in implementation

The Institute for Computer Sciences and Technology of the National Bureau of Standards published *Guideline for Lifecycle Validation, Verification, and Testing of Computer Software* in 1983, in which a methodology that integrates analysis, review, and test activities to provide product evaluation during the software lifecycle was described. The guideline gives the belief that a carefully chosen set of VV&T techniques can help to ensure the development and maintenance of quality software.

Phase V. Since 1988: The Prevention-Oriented Period
– Testing to prevent faults in requirements, design, and implementation

In the significant book *Software Testing Techniques* [2], which contains the most complete catalog of testing techniques, Beizer stated that "the act of designing tests is one of the most effective bug preventers known," which extended the definition of testing to error prevention as well as error detection activities. This led to a classic insight into the power of early testing.

In 1991, Hetzel gave the definition that "Testing is planning, designing, building, maintaining and executing tests and test environments." A year before this, Beizer gave four stages of thinking about testing: 1. to make software work, 2, to break the software, 3, to reduce risk, and 4, a state of mind, i.e. a total life-cycle concern with testability. These ideas led software testing to view emphasize the importance of early test design throughout the software life cycle.

The prevention-oriented period is distinguished from the evaluation-oriented by the mechanism, although both focus on software requirements and design in order to avoid implementation errors. The prevention model sees test planning, test analysis, and test design activities playing a major role, while the evaluation model mainly relies on analysis and reviewing techniques other than testing.

4.2 Major Technical Contributions

In general, the research on testing techniques can be roughly divided into two categories: *theoretical* and *methodological*. Software testing techniques are based in an amalgam of methods drawn from graph theory, programming language, reliability assessment, reliable-testing theory, etc. In this paper, we focus on those significant theoretical research results, such as test data adequacy and testing criteria, which provide a sound basis for creating and refining methodologies in a rational and effective manner. Given a solid theoretical basis, a systematic methodology seeks to employ rational techniques to force sequences of actions that, in aggregate, accomplish some desired testing-oriented effect.

We are going to start with the major technical contributions of theoretical researches as well as milestone methodologies of testing techniques. In next section, the Redwine/Riddle maturity model will be used to illustrate how testing techniques have matured from an intuitive, ad hoc collection of methods into an integrated, systematic discipline. Figure 2 shows the concept formation of testing, milestone technical contributions on testing techniques, including the most influential theoretical and method research. Note that the principle for paper selection is significance. A research is chosen and its idea shown in Figure 2 because it defines, influences, or changes the technology fundamentally in its period. It does not necessarily have to be the first published paper on similar topics.

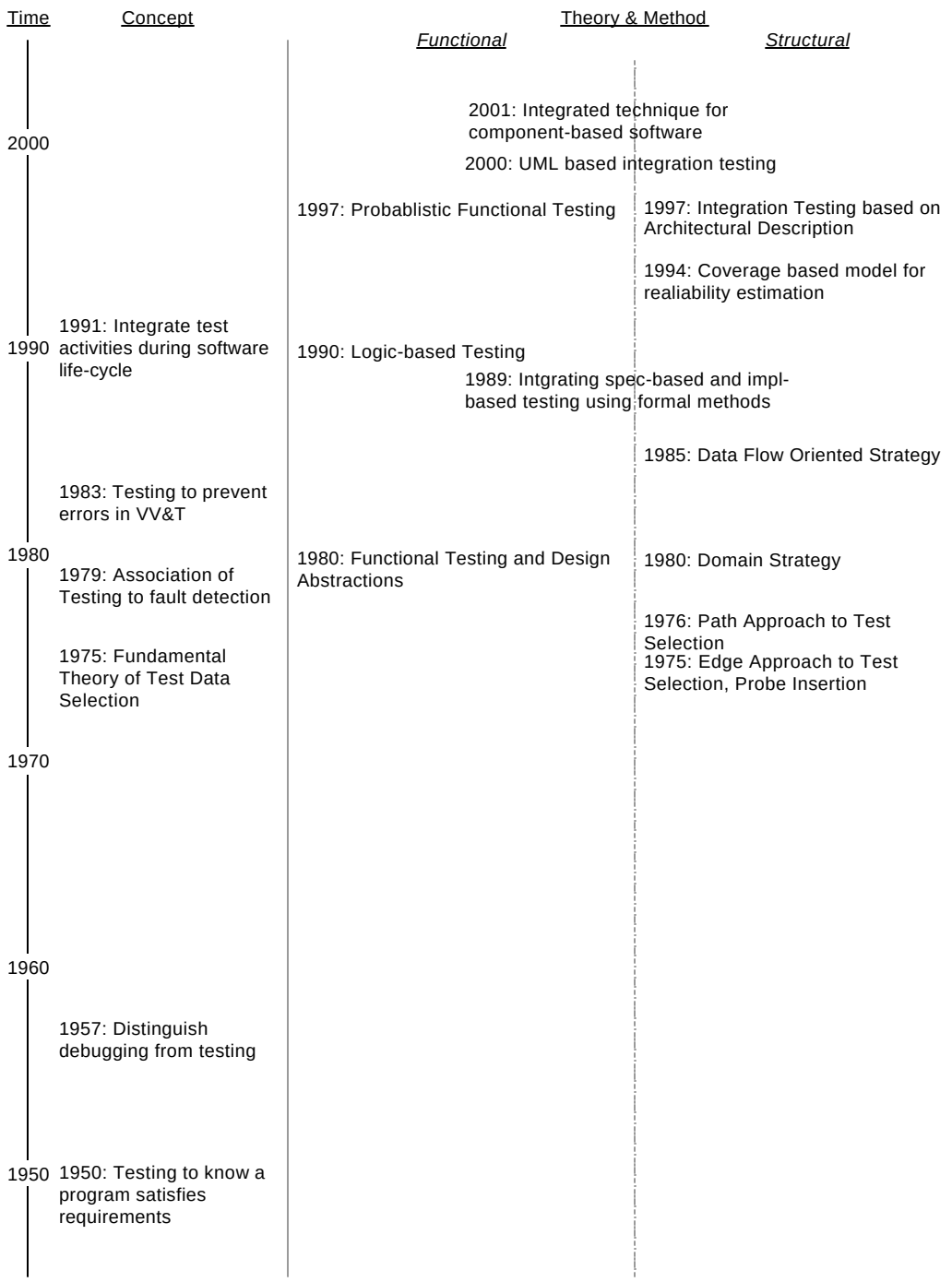

Figure 2. Major research results in the area of software testing techniques.

This diagram shows the most influential ideas, theory, methods and practices in the area of software testing, emphasizing on testing techniques. There are two columns in the diagram marked "Concept" and "Theory &Methods". The items appear in "Concept" column contains the leading technical viewpoints about the goals of testing software since the year 1950. The two sub-columns of "Theory & Method" classify theoretical and methodological research results in functional-based and structural-based.

Before the year 1975, although software testing was widely performed as an important part of software development, it remained an intuitive, somehow ad hoc collection of methods. People used principle functional and structural techniques in their testing practices, but there was little systematic research on methods or theories of these techniques.

In 1975, Goodenough and Gerhart gave the fundamental theorem of testing in their paper *Toward a Theory of Test Data Selection* [5]. This is the first published research attempting to provide a theoretical foundation for testing, which characterizes that a test data selection strategy is completely effective if it is guaranteed to discover any error in a program. As is mentioned in section 2, this gave testing a direction to uncover errors instead of not finding them. The limitation of the ideas in this paper is also analyzed in previous section. Their research led to a series of successive research on the theory of testing techniques.

In the same year, Huang pointed out that the common test data selection criterion – having each and every statement in the program executed at least once during the test – leaves some important classes of errors undetected [10]. As a refinement of statement testing criterion, **edge** strategy, was given. The main idea for this strategy is to exercise every edge in the program diagraph at least once. **Probe insertion**, a very useful testing technique in later testing practices was also given in this research.

Another significant test selection strategy, the ***path testing approach***, appeared in 1976. In his research, Howden gave the strategy that test data is selected so that each path through a program is traversed at least once [8]. Since the set of program paths is always infinite, in practice, only a subset of the possibly infinite set of program paths can be tested. Studies of the reliability of path testing are interesting since they provide an upper bound on the reliability of strategies that call for the testing of a subset of a program's paths.

The year 1980 saw two important theoretical studies for testing techniques, one on functional testing and one on structural.

Although functional testing had been widely used and found useful in academic and industry practices, there was little theoretical research on functional testing. The first theoretical approach towards how systematic design methods can be used to construct functional tests was given in 1980 [9]. Howden discussed the idea of **design functions**, which often correspond to sections of code documented by comments, which describe the effect of the function. The paper indicates how systematic design methods, such as structured design methodology, can be used to construct functional tests.

The other 1980 research was on structural testing. If a subset of a program's input domain causes the program to follow an incorrect path, the error is called a **domain error**. Domain errors can be caused by incorrect predicates in branching statements or by incorrect computations that affect variables in branch statement predicates. White and Cohen proposed a set of constraints under which to select test data to find domain errors [18]. The paper also provides useful insight into why testing succeeds or fails and indicates directions for continued research.

The dawn of **data flow analysis** for structural testing was in 1985. Rapps and Weyuker gave a family of test data selection criteria based on data flow analysis [16]. They contend the defect of path selection criteria is that some program errors can go undetected. A family of path selection criteria is introduced followed by a discussion of the interrelationships between these criteria. This paper also addresses the problem of appropriate test data selection. This paper founded the theoretical basis for data-flow based program testing techniques.

In 1989, Richardson and colleagues proposed one of the earliest approaches focusing on utilizing specifications in selecting test cases [14]. In traditional specification-based functional testing, test cases are selected by hand based on a requirement specification, thus makes functional testing consist merely heuristic criteria. Structural testing, on the other hand, has the advantage of that the applications can be automated and the satisfaction determined. The authors extended implementation-based techniques to be applicable with **formal specification** languages and to provide a testing methodology that combines

specification-based and implementation-based techniques. This research appeared to be the first to combine the idea of structural testing, functional testing, and formal specifications.

Another research towards specification-based testing appeared in 1990. Boolean algebra is the most basic of all logic systems and many logical analysis tools incorporate methods to simplify, transform, and check specifications. Consistency and completeness can be analyzed by using Boolean algebra, which can also be used as a basis for test design. The functional requirements of many programs can be specified by **decision tables,** which provide a useful basis for program and test design. Boolean algebra is trivialized by using **Karnaugh-Veitch charts**. Beizer described the method of using decision tables and Karnaugh-Veitch charts to specify functional requirements in his book [2]. This was among the earliest approaches to select test based on existing, well-known Boolean algebra logic in test data selection.

In the early 1990s, there was an increasing interest in estimating and **predicting the reliability** of software systems. Before Jalote and colleagues' study in 1994 [11], many existing reliability models employed functional testing techniques and predicted the reliability based on the failure data observed during testing. The application of these models requires a fair amount of data collection, computation, and expertise and computation for interpreting the results. The authors propose a new approach based on the coverage history of the program, by which a software system is modeled as a graph, and the reliability of a node is assumed to be a function of the number of times it gets executed during testing-the larger the number of times a node gets executed, the higher its reliability. The reliability of the software system is then computed through simulation by using the reliabilities of the individual nodes. With such a model, coverage analysis tools can easily be extended to compute the reliability also, thereby fully automating reliability estimation.

The year 1997 saw good results in both functional and structural testing techniques. In this year a framework for probabilistic functional testing was proposed. Bernot and colleagues introduce the formulation of the testing activity, which guarantees a certain level of confidence into the correctness of the system under test, and gives estimate of the reliability [1]. They also explain how one can generate appropriate distributions for data domains including most common domains such as intervals of integers, unions, Cartesian products, and inductively defined sets.

Another interesting research in 1997 used formal architectural description for rigorous, automatable method for integration test of complex systems [4]. In this paper the authors propose to use the formal specification language CHAM to model the behavior of interest of the systems. Graph of all the possible behaviors of the system in terms of the interactions between its components is derived and further reduced. A suitable set of reduced graphs highlights specific architectural properties of the system, and can be used for the generation of integration tests according to a coverage strategy, analogous to the control and data flow graphs in structural testing. This research is among the trend of using formal methods in testing techniques since later 1980s.

From late 1990s, Commercial Off The Shelf (COTS) software and Unified Modeling Language (UML) UML have been used by increasing number of software developers. This trend in development therefore calls the corresponding testing techniques for the UML components. In their 2000 paper [7], Hartmann and colleagues at Siemens addressed the issue of testing components by integrating test generation and test execution technology with commercial UML modeling tools such as Rational Rose. The authors present their approach to modeling components and interactions, describe how test cases are derived from these component models and then executed to verify their conformant behavior. The *T*nT environment of Siemens is used to evaluate the approach by examples. Test cases are derived from annotated Statecharts.

Another most recent research is also an approach to test component-based software. In 2001, Beydeda and colleagues proposed a graphical representation of component-based software flow graph [3]. Testing is made complicated with features, such as the absence of component source code, that are specific to component-based software. The paper proposes a technique combining both black-box and white-box strategies. A graphical representation of component software, called component-based software flow graph (CBSFG), which visualizes information gathered from both specification and implementation, is described. It can then be used for test case identification based on well-known structural techniques. Traditional

structural testing techniques can be applied to this graphical representation to identify test cases, using data flow criterion. The main components are still tested with functional techniques.

5 Technology maturation

The maturation process of how the testing techniques for software have evolved from an ad hoc, intuitive process, to an organized, systematic technology discipline is shown in Figure 3. Three models are used to help analyzing and understanding the maturation process of testing techniques. A brief introduction on these models followed.

5.1 Redwine/Riddle software technology maturation model

The Redwine/Riddle is the backbone of this maturation study. In 1985, Redwine and Riddle viewed the growth and propagation of a variety of software technologies in an attempt to discover natural characteristics of the process as well as principles and techniques useful in transitioning modern software technology into widespread use. They looked at the technology maturation process by which a piece of technology is first conceived, then shaped into something usable, and finally "marketed" to the point that it is found in the repertoire of a majority of professionals. Six phases are described in Redwine/Riddles model:

Phase 1: Basic research.

Investigate basic ideas and concepts, put initial structure on the problem, frame critical research questions.

For testing techniques, the basic research phase ended in the mid to late 1950s, when the targets of testing were set to make sure the software satisfies its specification. Initial researches began to try to formulate the basic principles for testing.

Phase 2: Concept formulation.

Circulate ideas informally, develop a research community, converge on a compatible set of ideas, publish solutions to specific subproblems.

The concept formulation process of testing focused between late 1950s and mid 1970s, when the arguments of the real target of testing came to an end and theoretical researches began. The milestone of the end of this phase is the publication of Goodenough and Gerhart's paper on testing data selection.

Phase 3: Development and extension.

Make preliminary use of the technology, clarify underlying ideas, and generalize the approach.

The years between mid 1970s and late 1980s are the development and extension phase of testing techniques. Various testing strategies are proposed and evaluated for both functional and structural testing during this period. Principles of testing techniques were built up gradually during this period.

Phase 4: Internal enhancement and exploration.

Extend approach to another domain, use technology for real problems, stabilize technology, develop training materials, show value in results.

Starting from late 1980s, researches on software testing techniques entered the internal exploration stage. The principles defined during the last period were accepted widely and used in larger scales of real problems, technologies began to stabilize and expand. People can now hire testing specialists or

companies to do the testing job for them. More and more courses and training programs are given in universities and companies.

Phase 5: External enhancement and exploration.

Similar to internal, but involving a broader community of people who weren't developers, show substantial evidence of value and applicability.

Testing techniques started their prosperity in mid 1990s. Along with the advance of programming languages, development methods, abstraction granularity and specification power, testing became better facilitated and challenged with more complex situations and requirements of different kinds of systems. Effective specification and development methods are borrowed employed during testing.

Phase 6: Popularization.

Develop production-quality, supported versions of the technology, commercialize and market technology, expand user community.

Although the testing activities have been carried out everywhere and all the time, we don't think research on testing techniques have entered its popularization period. Testing techniques vary with different systems and development methods. Even if specialist companies can be hired to do testing job, there lacks a common basis for the industry to share and utilize academic research result, as well as to use good practices from other enterprises. We are going to discuss this in future sections.

5.2 Brief History of Software Engineering

The evolution of software testing cannot be separated with activities in other areas of software engineering, such as programming language, formal specification, and software architecture. In her paper, Shaw summarizes the significant shift in research attention of software engineering, which can be viewed as a brief history of this broad area [12]. Starting from mid 1950s, the change of software engineering research paradigms can be described as shown in Table 1.

Table 1. Significant shifts in research attention of software engineering

1960 ± 5 **Programming-any-which-way**	Mnemonics, precise use of prose Emphasis on small programs Representing structure, symbolic information Elementary understanding of control flow
1970 ± 5 **Programming-in-the-small**	Simple input-output specifications Emphasis on algorithms Data structures and types Programs execute once and terminate
1980 ± 5 **Programming-in-the-large**	Systems with complex specifications Emphasis on system integration, management Long-lived databases Program assemblies execute continually
1990 ± 5 **Programming-in-the-world**	Software integrated with hardware Emphasis on management process improvement, system structure Abstractions for system design (client-server, object…) Heavily interactive systems, multimedia

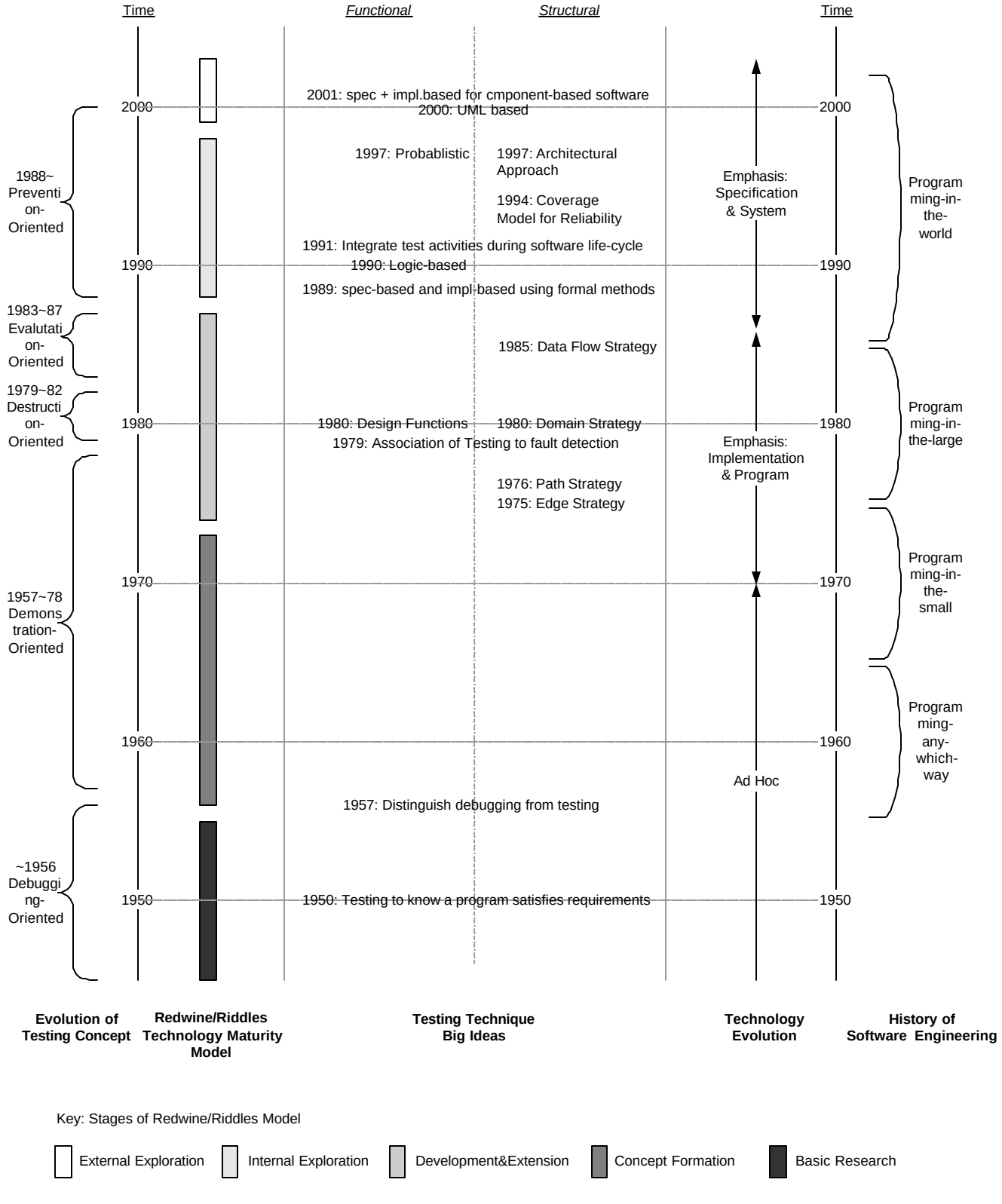

Figure 3. Technology maturation analysis of software testing techniques

5.3 Testing Process Models

The third reference model is the major testing goals model given by Gelperin and Hetzel in [6]. We have discussed this in detail in our previous section of testing concept evolution. Understanding how the goals of testing has changed helps us a lot to understand the intellectual history of testing technique research, how new ideas are based on old ideas from the same area or on the studies from other areas.

The Redwine/Riddle model provides benchmarks for judging where a research stands in the maturation process, and how a process can migrate to the next one. Shaw's model sets up the global background where studies of technology maturation can be put into and compared with the rest of the world of software engineering. Gelperin and Hetzel's model is specific in testing so that the objectives of a testing technique research are made clear. Each model assists us to understand the problem from a different point of view, but none of them alone fully explains the unique phenomena in the technology history of test techniques. We put our study in the framework built with these models, and give our own viewpoints.

5.4 The Major Stages of Research & Development Trends

Generally, we see three major stages of the research and development of testing techniques, each with a different trend. By trend we mean the how mainstream of research and development activities find the problems to solve, and how they solve the problems. As is shown in the column of "Technology Evolution" in Figure 3, testing technique technologies, thus the ways of selecting test data have developed from ad hoc, experienced implementation-based phase, and is focusing on specification-based now.

1950 – 1970: Ad Hoc

From Figure 3 we can see that between the years 1950 and 1970, there were few research results on testing techniques except for the conceptual ideas of testing goals. It's possible that research results before 1970 are too old to be in the reach of current bibliography collections. To avoid being influenced by this factor, we looked at many testing survey papers in the 1980s, which should have had the "ancient" studies in hand by the time they performed their study. We suppose their surveys at least addressed the most important technical contributes before their time, and we can build our research for the decades before 1970 on theirs.

Based on above assumption, we define the period between 1950 and 1970 as being ad hoc. During this period, major research interest focuses on the goal of testing, and there are quite a few discussions on how to evaluate if a test is good. Meanwhile, testing had become gradually independent from part of debugging activities, to a necessary way to demonstrate that a program satisfies its requirements, as is seen in the GH88 model. At the same time, if we look from Shaw's view, we can see that the whole world of software engineering was in its programming-in-any-which-way stage. It's very natural that testing stayed in its ad hoc stage, where test data is selected randomly and in an unorganized, undirected way.

1971 – 1985: Emphasize on Implementation and Single Program

Beginning from the mid 1960s to the mid 1980s, the whole software engineering research community shifted it paradigms to the program-in-the-small stage, and then started the program-in-the-large stage. The main changes this migration brought to software development were that the characteristic problems changed from small programs, to larger programs and algorithms, and were on the way to developing more complex problems. In response to this significant change, researches on testing techniques began their prosperity.

On the structural side, in 1975, 1976, 1980, and 1985, four significant papers were published, each proposing a very important structural testing strategy, and all of which were adopted as the classic criteria for later researches followed them. From Figure 3 we can see that almost all the important

theoretical structural testing researches appeared in this period. Although the whole software engineering community was facing the challenge of switching the gear of developing from comparably simple programs to complex large systems, it took time for testing community to react to the change, specifically, in approximately 5 years.

From the figure we also find that only one significant result for functional testing appeared in this period. The reason is obvious. Functional testing is based on requirements and has consisted merely of heuristic criteria. It is difficult to determine when and if such criteria are satisfied without being able to express the requirements in an efficient, rigorous, unambiguous way. This was in part the motivation for developing implementation-based testing techniques; they have the advantage that their application can be automated and their satisfaction determined. Fortunately the research appeared during this period set up a very good tone of successive researches, since it moved emphasis from the simple input/output specifications that testers often used in this period to a higher level – the design of the system.

In this period, how to test a "program", instead of a "system", still drew the attention of researchers and practitioners. However the whole software engineering had begun to get ready for moving from the stage of programming-in-the-large to a higher level.

1986 – current: Emphasize on Specification and System

As software become more and more pervasive, the engineering for this area experienced the shift from programming-in-the-large to programming-in-the-world, starting from the mid 1980s. The characteristic problems changed from algorithms, to system structures, and component interfaces. Systems have been specified in more complex ways. Studies in software architecture and formal methods have brought a lot of facilities as well as inspiration to the way people specifying their systems. Based on these studies, software system now can be specified in more rigorous, understandable, automatable ways, which has brought great chances to improve functional testing techniques. Meanwhile, software development is no longer limited to standalone systems, in reality, there have been more and more needs to develop distributed, object-oriented, and component based systems. The researchers in testing community have responded this trend and move their emphasis accordingly.

Starting from the late 1980s, many researches have made use of the achievements of formal methods and logical analysis. There is still limitation in the specification capabilities so that researchers have been calling for better specification methods to improve their results. Both functional and structural testing techniques have benefited from the enhancement of software specification technologies.

The widespread developing and using of object-oriented technologies, COTS software and component-based systems has brought a great density of testing researches on these kinds of systems. The earliest OO testing studies appeared in the early 1990s. Most of them use traditional functional and/or structural techniques on the components, i.e. classes and so on. Researchers have proposed new problems and solutions on testing the connections and inheritances among components. Both structural and functional techniques are hired in their approaches, and it has proven to be an effective method to integrate the two techniques for testing complex systems.

5.5 The Test Gap

In present, testing techniques have gradually involved from the practice of single programmers or small development teams into a systematic, managed engineering discipline. Not only have there been numerous researches on testing techniques, but also more and more considerable industry practices. There are testing classes taught in universities. There have been special testing teams, test managers, and tester job positions open to professional testers; there have been training programs and complete procedures for testing in large enterprises; and there are increasing number of companies and vendors doing testing work for other companies.

However, despite the numerous research results (quite a lot of them are really sound) testing remains an awkward, time-consuming, cost-ineffective chunk of work that is always not very satisfying in most industry practices. Only a small number of the research results have been utilized successfully in industry practices so that the test process can be greatly improved or automated. The most common testing exercises in industry are static analysis including code inspections, peer reviews, walkthroughs. Not enough testing tools can be applied directly on industry projects and products without being largely modified and even re-developed. Test plans are still written by hand, while test environment remained simple and crude. It is always the case that testing ends up being a must-end activity because the project runs out of budget and is beyond deadline. This inconsistency of testing research and practice has been called the "testing gap."

Provided many other issues involved in the testing gap, such as process and management, further researches in testing technique are among the most significant solutions that will work for the problem. Fundamental researches in techniques need to:

- Demonstrate effectiveness of existing techniques
- Address the need in new areas
- Create new adaptive techniques
- Facilitate transferring technology to industry

5.6 Research Strategies for Testing Techniques

We have studied the research strategies of twelve influential papers since the year 1975 and tried to find out the common form and successful examples of research settings that offer concrete guidance for future research work. The results are listed in Table 2. It's not hard to find out that most of researches on testing techniques are motivated by questioning if there is a better method of doing something. The question is answered by inventing, implementing, combining, refining, evaluating, alternating or proposing new ways to do this task, and the result gets analyzed through analysis most of the times. Combined with the test gap mentioned in last section, we contend that fundamental researches should address the challenges testing techniques are facing in the real world, generalize them, and pursue practical solutions for them. Research should be carried out with industry partners on real world problems, instead of simple toy systems. Researchers in academic community and in industry should talk often to address the need for each other.

Table 2. Paradigms of testing technique researches

Paper Ref.	Year	Age	Idea	Research Paradigm		
				Question	Result	Validation
GG75	1975	26	Fundamental theorem	Evaluation	Analytic Model	Analysis
Huang75	1975	26	Edge approach, probe insertion	Method/Means	Technique	Analysis
Howden76	1976	25	Path approach and its reliability	Characterization	Technique	Analysis
WC80	1980	21	Domain testing strategy	Method/Means	Technique	Analysis
Howden80	1980	21	Functional design abstraction	Method/Means	Technique	Persuasion
RW85	1985	16	Data flow strategy	Method/Means	Technique	Analysis
ROT89	1989	12	Integrate spec. and impl. testing	Method/Means	Technique	Analysis
JM94	1994	7	Coverage reliability estimation	Method/Means	Technique	Analysis
BBL97	1997	4	Probabilistic functional testing	Method/Means	Technique	Analysis
BIMR97	1997	4	Testing based on architectural	Method/Means	Technique	Persuasion
HIM00	2000	1	UML based testing	Method/Means	Technique	Experience
BG01	2001	0	Component based testing	Method/Means	Technique	Analysis

6 Conclusion

Testing has been widely used as a way to help engineers develop high-quality systems, and the techniques for testing have evolved from an ad hoc activities means of small group of programmers to an organized discipline in software engineering. However, the maturation of testing techniques has been fruitful, but not adequate. Pressure to produce higher-quality software at lower cost is increasing and existing techniques used in practice are not sufficient for this purpose. Fundamental research that addresses the challenging

problems, development of methods and tools, and empirical studies should be carried out so that we can expect significant improvement in the way we test software. Researchers should demonstrate the effectiveness of many existing techniques for large industrial software, thus facilitating transfer of these techniques to practice. The successful use of these techniques in industrial software development will validate the results of the research and drive future research. The pervasive use of software and the increased cost of validating it will motivate the creation of partnerships between industry and researchers to develop new techniques and facilitate their transfer to practice. Development of efficient testing techniques and tools that will assist in the creation of high-quality software will become one of the most important research areas in the near future.

7 Annotated bibliography

[1] G. Bernet, L. Bouaziz, and P. LeGall, "A Theory of Probabilistic Functional Testing," *Proceedings of the 1997 International Conference on Software Engineering*, 1997, pp. 216 –226

[BBL97] A framework for probabilistic functional testing is proposed in this paper. The authors introduce the formulation of the testing activity, which guarantees a certain level of confidence into the correctness of the system under test. They also explain how one can generate appropriate distributions for data domains including most common domains such as intervals of integers, unions, Cartesian products, and inductively defined sets. A tool assisting test case generation according to this theory is proposed. The method is illustrated on a small formal specification.

Question: Method/Means
Result: Technique
Validation: Analysis

[2] B. Beizer, "Software Testing Techniques," Second Edition, *Van Nostrand Reinhold Company Limited*, 1990, ISBN 0-442-20672-0

[Beizer90] This book gives a fairly comprehensive overview of software testing that emphasizes formal models for testing. The author gives a general overview of the testing process and the reasons and goals for testing. In the second chapter of this book, the author classifies the different types of bugs that could arise in program development. The notion of path testing, transaction flowgraphs, data-flow testing, domain testing, and logic-based testing are introduced in detail in the chapters followed. The author also introduces several attempts to quantify program complexity, and more abstract discussion involving paths, regular expression, and syntax testing. How to implement software testing based on the strategies is also discussed in the book.

[3] S. Beydeda and V. Gruhn, "An integrated testing technique for component-based software," *ACS/IEEE International Conference on Computer Systems and Applications*, June 2001, pp 328 – 334

[BG01] Testing is made complicated with features, such as the absence of component source code, that are specific to component-based software. The paper proposes a technique combining both black-box and white-box strategies. A graphical representation of component software, called component-based software flow graph (CBSFG), which visualizes information gathered from both specification and implementation, is described. It can then be used for test case identification based on well-known structural techniques.

Question: Method/Means
Result: Technique
Validation: Analysis

[4] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti, "An approach to integration testing based on architectural descriptions," *Proceedings of the IEEE ICECCS*- 97, pp. 77-84

[BIMR97] In this paper the authors propose to use formal architectural descriptions (CHAM) to model the behavior of interest of the systems. Graph of all the possible behaviors of the system in terms of the interactions between its components is derived and further reduced. A suitable set of reduced graphs highlights specific architectural properties of the system, and can be used for the generation of integration tests according to a coverage strategy, analogous to the control and data flow graphs in structural testing.

Question: Method/Means
Result: Technique
Validation: Persuasion

[5] J.B. Good Enough and S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, June 1975, pp. 156-173

[GG75] This paper is the first published paper, which attempted to provide a theoretical foundation for testing. The "fundamental theorem of testing" brought up by the authors characterizes the properties of a completely effective test selection strategy. The authors think a test selection strategy is completely effective if it is guaranteed to discover any error in a program. As an example, the effectiveness of branch and path testing in discovering errors is compared. The use of decision table (a mixture of requirements and design-based functional testing) as an alternative method is also proposed.

Question: Evaluation
Result: Analytic Model
Validation: Analysis

[6] D. Gelperin and B. Hetzel, "The Growth of Software Testing", *Communications of the ACM*, Volume 31 Issue 6, June 1988, pp. 687-695

[GH88] In this article, the evolution of software test engineering is traced by examining changes in the testing process model and the level of professionalism over the years. Two phase models, the demonstration and destruction models, and two life cycle models, the evolution and prevention models are given to characterize the growth of software testing with time. Based on the models a prevention oriented testing technology is introduced and analyzed in detail.

Question: Characterization
Result: Descriptive Model
Validation: Persuasion

[7] J. Hartmann, C. Imoberdorf, and M.Meisinger, "UML-Based Integration Testing," *Proceedings of the International Symposium on Software Testing and Analysis*, ACM SIGSOFT Software Engineering Notes**,** August 2000

[HIM00] Unified Modeling Language (UML) is widely used for the design and implementation of distributed, component-based applications. In this paper, the issue of testing components by integrating test generation and test execution technology with commercial UML modeling tools such as Rational Rose is addressed. The authors present their approach to modeling components and interactions, describe how test cases are derived from these component models and then executed to verify their conformant behavior. The T*n*T environment of Siemens is used to evaluate the approach by examples

Question: Method/Means
Result: Technique
Validation: Experience

[8] W. E. Howden, "Reliability of the Path Analysis Testing Strategy", *IEEE Transactions on Software Testing*, September 1976, pp. 208-215

[Howden76] The reliability of path testing provides an upper bound for the testing of a subset of a program's paths, which is always the case in reality. This paper begins by showing the impossibility of constructing a test strategy that is guaranteed to discover all errors in a program. Three commonly occurring classes of errors, computations, domain, and subcase, are characterized. The reliability properties associated with these errors affect how path testing is defined.

Question: Characterization
Result: Technique
Validation: Analysis

[9] W. E. Howden, "Functional Testing and Design Abstractions," *The Journal of System and Software*, Volum 1, 1980, pp. 307-313

[Howden80] The usual practice of functional testing is to identify functions that are implemented by a system or program from requirements specifications. In this paper, the necessity of testing design as well as requirement functions is discussed. The paper indicates how systematic design methods, such as Structured design and the Jackson design can be used to construct functional tests. Structured design can be used to identify the design functions that must be tested in the code, while the Jackson method can be used to identify the types of data which should be used to construct tests for those functions.

Question: Method/Means
Result: Technique
Validation: Persuasion

[10] J. C. Huang, "An Approach to Program Testing," *ACM Computing Surveys*, September 1975, pp.113-128

[Huang75] This paper introduces the basic notions of dynamic testing based on detailed path analysis in which full knowledge of the contents of the source program being tested is used during the testing process. Instead of the common test criteria by which to have every statement in the program executed at least once, the author suggested and demonstrated by an example, that a better criterion is to require that every edge in the program diagraph be exercised at least once. The process of manipulating a program by inserting probes along each segment in the program is suggested in this paper.

Question: Method/Means
Result: Technique
Validation: Analysis

[11] P. Jalote and Y. R. Muralidhara, "A coverage based model for software reliability estimation," *Proceedings of First International Conference on Software Testing, Reliability and Quality Assurance*, 1994, pp. 6 – 10 (IEEE)

[JM94] There exist many models for estimating and predicting the reliability of software systems, most of which consider a software system as a black box and predict the reliability based on the failure data observed during testing. In this paper a reliability model based on the software structure is proposed. The model uses the number of times a particular module is executed as the main input. A software system is modeled as a graph, and the reliability of a node is assumed to be a function of the number of times it gets executed during testing – the larger the number of times a node gets executed, the higher its reliability. The reliability of the software system is then computed through simulation by using the reliabilities of the individual nodes.

Question: Method/Means
Result: Technique
Validation: Analysis

[12] J. J. Marciniak, "Encyclopedia of software engineering", Volume 2, New York, NY: Wiley, 1994, pp. 1327-1358

[Marciniak94] A book intended for software engineers, this books gives introductions, overviews, and technical outlines of the major areas in software engineering. A review in to test generators is given where the major types of test case generators are given and their intended purpose and principles are discussed. A review on the testing process is given where the entire process of testing is discussed from planning to execution to achieving to maintenance retesting. All the common terms and ideas are discussed. A review of testing tools is given where the testing tools for each purpose is discussed and a couple for state of the art systems are given.

[13] E. F. Miller, "Introduction to Software Testing Technology," *Tutorial: Software Testing & Validation Techniques*, Second Edition, IEEE Catalog No. EHO 180-0, pp. 4-16

[Miller81] This article serves as the one of the introductory sections of the book *Tutorial: Software Testing & Validation Techniques*. A cross section of program testing technology before and around the year 1980 is provided in this book, including the theoretical foundations of testing, tools and techniques for static analysis and dynamic analysis, effectiveness assessment, management and planning, and research and development of soft ware testing and validation. The article briefly summarizes each of the major sections. The article also gives good view of the motivation forces, the philosophy and principles of testing, and the relation of testing to software engineering.

[14] D. Richardson, O. O'Malley and C. Tittle, "Approaches to specification-based testing", *ACM SIGSOFT Software Engineering Notes*, Volume 14 , Issue 9, 1989, pp. 86 – 96

[ROT89] This paper proposes one of the earliest approaches focusing on utilizing specifications in selecting test cases. In traditional specification-based functional testing, test cases are selected by hand based on a requirement specification, thus makes functional testing consist merely heuristic criteria. Structural testing has the advantage of that the applications can be automated and the satisfaction determined. The authors propose approaches to specification-based testing by extending a wide variety of implementation-based testing techniques to be applicable to formal specification languages, and demonstrate these approaches for the Anna and Larch specification languages.

Question: Method/Means
Result: Technique
Validation: Analysis

[15] S. Redwine & W. Riddle, "Software technology maturation," *Proceedings of the Eighth International Conference on Software Engineering*, May 1985, pp. 189-200

[RR85] In this paper, a variety of software technologies are reviewed. The technology maturation process by which a piece of technology first gets the idea formulated and preliminarily used, then is developed and extended into a broader solution, and finally is enhanced to product-quality applications and marketed to the public. The time required for a piece of technology to mature is studied, and the actions that can accelerate the maturation process are addressed. This paper serves as a very good framework for technology maturation study.

Question: Characterization
Result: Empirical Model
Validation: Analysis

[16] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering* , April 1985, pp. 367-375

[RW85] A family of test data selection criteria based on data flow analysis is defined in this paper. The authors contend that data flow criteria are superior to currently path selection criteria being used in that using the latter strategy program errors can go undetected. Definition/use graph is introduced and compared with a program graph based on the same program. The interrelationships between these data flow criteria are also discussed.

Question: Method/Means
Result: Technique
Validation: Analysis

[17] M. Shaw, "Prospects for an engineering discipline of software," *IEEE Software*, November 1990, pp. 15-24

[Shaw90] Software engineering is still on its way of being a true engineering discipline. This article studies the model for the evolution of an engineering discipline and applies it to software technology. Five basic steps are suggested to the software profession to take towards a true engineering discipline: to understand the nature of expertise, to recognize different ways to get information, to encourage routine practice, to expect professional specializations, and to improve the coupling between science and commercial practice. The significant shifts in research attention of software engineering since the 1960s are also given in this article.

Question: Characterization
Result: Descriptive Model
Validation: Persuasion

[18] L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing," *IEEE Transactions on Software Engineering*, May 1980, pp. 247-257

[WC80] Domain errors are in the subset of the program input domain, and can be caused by incorrect predicates in branching statements or incorrect computations that affect variables in branching statements. In this paper a set of constraints under which it's possible to reliably detect domain errors is introduced. The paper develops the idea of linearly bounded domains. The practical limitations of the approach are also discussed, of which the most severe is that of generating and then developing test points for all boundary segments of all domains of all program paths.

Question: Method/Means
Result: Technique
Validation: Analysis

[19] J. A. Whittaker, "What is Software Testing? And Why Is It So Hard?" *IEEE Software*, January 2000, pp. 70-79

[Whit00] Being a practical tutorial article, the paper answers questions from developers how bugs escape from testing. Undetected bugs come from executing untested code, difference of the order of executing, combination of untested input values, and untested operating environment. A four-phase approach is described in answering to the questions. By carefully modeling the software's environment, selecting test scenarios, running and evaluating test scenarios, and measuring testing progress, the author offers testers a structure of the problems they want to solve during each phase.

Question: Characterization
Result: Qualitative & Descriptive Model
Validation: Persuasion