DISTRIBUTED TESTING IN COLLABORATIVE SOFTWARE DEVELOPMENT

by

CAGLAN KUYUMCU

Bachelor of Science in Civil and Environmental Engineering

Middle East Technical University, 1998.

Submitted to the Department of Civil and Environmental Engineering in Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING IN CIVIL AND ENVIRONMENTAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Signature of Author.....
Department of Civil and Environmental Engineering
May 14, 1999

Certified by.....
Dr. Feniosky Peña-Mora
Associate Professor, Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by.....
Andrew J. Whittle
Chairman, Departmental Committee on Graduate Studies

DISTRIBUTED TESTING IN COLLABORATIVE SOFTWARE DEVELOPMENT

by

CAGLAN KUYUMCU

Submitted to the

Department of Civil and Environmental Engineering

on May 14, 1999

In Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING IN CIVIL AND ENVIRONMENTAL ENGINEERING

ABSTRACT

Distributed Testing in Collaborative Software Development aims to provide an understanding of the fundamentals of software testing, and testing in geographically distributed teams. It provides a thorough coverage of the definition of testing, its evolution and current state, techniques and strategies in testing, and special aspects of testing in the development of Distributed Software Applications. A case study of CAIRO, the Civil and Environmental Engineering Information Technology Master of Engineering yearly project performed by the DiSEL-98 team, is also included as an example. Various concerns and problems regarding both technical limitations and collaboration difficulties in testing in distributed teams are stated throughout the analyses. Finally, various comments and solutions to these problems are offered, and an overview of the future of testing is given.

Thesis Supervisor: Feniosky Peña-Mora

Title: Associate Professor, Department of Civil and Environmental Engineering

CONTENTS

List of Tables

List of Figures

CHAPTER 1

Insight to Testing

1.1 Introduction

Software testing has long been regarded as an important component of software engineering and development. Traditionally, testing was a brief, informal exercise performed by the programmer. Its purpose was to check that the code would execute and produce expected results from a few test inputs. It was still capable of detecting errors, but not of proving their absence, and thus provided very little confidence in the reliability of the software. Few sources were allotted to this activity, and relatively little importance was attached to it. It was expected that some errors could be found by the users, and would be corrected with amendments, patches, and updated releases sent long after the first issue of the software product. Software was regarded acceptable as, and even expected to be flawed.

For the last two decades, this attitude has been reversed. Software programs today control the smooth running of many of the functions of daily life, as well as some life-and-death applications. As a result of this importance, software unreliability is not tolerated. The increased standards for establishing high confidence in software programs have created a further-stretched-demand than what the capabilities of traditional testing methods for software quality assurance could offer. In order to answer these new demands in today's software development, more powerful software validation and testing functions are used compared to that of the past. These present techniques

are much improved and diversified versions of these past validation and testing methods. They promise greater accuracy and require fewer resources.

A great deal of research has been done towards improving the efficiency and effectiveness of software testing through the past three decades. The general improvements achieved in the process have focused on two areas. The first area has been formalizing and extending of existing testing techniques through analysis of different program structures and reliability theories. Work in this area has created the well-structured standard-based testing that we know of today. The second area has been the formation of new testing techniques. In addition to the previously mentioned improvements in the methodologies of the past, some completely new approaches to testing have been developed and perfected throughout the past years. Much of this contribution has been on the following areas:

- Verification techniques – Existing deficiencies of conventional testing have led to investigations of new possibilities, for logically proving the correctness of software. In 1975, the use of 'proving' was found out to be as error-prone as testing, and possible only with more effort [Infotech, 1979]. Following more research in the area, verification was decided to be more conclusive than any testing method for showing correctness, albeit with much higher effort requirements. The high effort requirement made verification worthwhile for only small, essential code modules, but also a powerful tool due to its higher reliability. More recent symbolic execution methods have provided higher confidence in a program's correctness, with less effort requirement compared to conventional verification techniques.

- Static testing – Static testing techniques concentrate on finding errors without executing the program code. This provides the tester with a tool to begin testing during early stages of the software development cycle. It is shown by empirical studies that most errors are made early in the software development cycle in the form of simple design, and coding faults. Since these errors will remain undetected in the code for a long time, it decreases costs significantly to find and correct these errors as early as possible. A majority of faults, if not all, can be detected by simple static testing techniques, such as design reviews and code inspections.

- Test data selection – The input data selection for the program is a crucial determinant of the amount of errors dynamic testing will detect. A large amount of research has gone into

generating optimal test sets, or test cases with enough coverage so as to prove the reliability of the software with a desired degree of confidence.

- Automation of testing techniques – The repetitive nature of testing causes boredom, which in effect leads to mistakes when applied by humans. To take maximum advantage of present techniques, automation, formed by a set of tools tailored to individual needs, is a necessity.

- Testing for parallel and distributed computing systems – New software engineering methods have been developed and introduced throughout the evolution of the computer industry, in order to keep software development in pace with the increasing power of computers. Demands of today's highly competitive global economy brought about a new trend, one requiring the use of parallel and distributed computer systems by utilizing the high capacity of present hardware. Similarly, demands on the software industry have changed to higher standards, which require special methodologies for software quality assurance for distributed computing systems. Finally, higher reliability standards led to distributed working environments that utilize these computer systems.

All these complementary techniques and approaches form a useable set for complete software testing. Different methods apply to different stages of software development, and it is intended that a variety of techniques be applied in turn for any piece of software being tested. A methodology of systematic testing integrates this approach into the phases of the software development cycle, ensuring adequate testing, as well as maximum flexibility.

Testing is no longer regarded as a secondary, informal function of software development. Instead, it has become a major function requiring dedicated test engineers and having its own standards and techniques. In today's software development sector, extensive testing is performed throughout the world and in the development of all software programs. The result is programs with high levels of globally accepted reliability, and thus a more confident and widespread use of computer software for all types of tasks.

1.2 Definitions and Theory of Testing

1.2.1 Characteristics of Errors in a Software Program

There are various notions that are commonly attributed to various aspects of programs that perform something unintended, unexpected, or otherwise wrong. The first one is a human or environment initiated event that produces an unintended result, considered to be a **fault**. Most common examples of faults are typos that escape compiler detection. Any fault gives rise to errors.

An **error** is the result of a fault during the creation of a software item; it is incorporated into the item and is eventually revealed by a failure of that item. A **failure** is the inability of a system (or one of its components) to perform a given function within given limits. A failure may often be produced in occurrence of an error.

A **bug** is a commonly used word to refer to an error of an originally unknown location and reason. It can also be attributed to a poorly prepared and executed test, as well as a bad program description that may lead users to mistake good behavior for bugs.

Software testing aims to reveal errors, ideally by exercising the run-time code in an attempt to observe various behavior and output anomalies in a series of experiments. These experiments constitute *dynamic analysis*. On the other hand, program behavior can often be simulated before compilation, based on a realistic interpretation of the source code. Experiments of such nature constitute *static analysis* [Krawczyk et al, 1998].

While detecting the presence of errors in a program is the task of testing, locating and removing errors that cause particular bugs involves another process, known as **debugging**. The difficulty in debugging is that knowing that a program has a bug is not sufficient to determine the error behind it. Different bugs may manifest themselves in the same way, and another bug may have small symptoms. Debugging requires repeated executions of a code for the conductance of small detailed tests in order to observe patterns in the code behavior. Debugging and testing use essentially the same mechanisms for monitoring program variables, detecting states and

evaluating conditions during program execution. Testing, however, puts more emphasis on semantics of test data in order to assess test coverage with regard to specific classes of errors. Debugging is more concerned with systematically collecting various manifestations and symptoms related to a particular bug in order to devise and verify a hypothesis for the reason behind it.

Many experts throughout the past three decades have tried the development of a categorization structure, for possible errors in computer software programs. There is, however, no single consistent result, but a very high number of different classification schemes. These categories tend to be influenced by two important factors in general: the environment in which the study is done, and whether the study focused on the location of the problem or the cause of the problem.

1.2.2 The Role of Testing in Software Engineering

Software Engineering has been described as "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them" [Boehm, 1976]. It has become a significant part of the computer science field during the last three decades due to the rapid advances in hardware technology, coupled with less noticeable advances in software technology. As a result, more and more of total computing cost has become software costs, as opposed the major portion held by hardware costs some twenty to thirty years ago.

Computer software follows a life cycle, which begins with a **specification** and **analysis** of the **requirements** of the software product, followed by a **design** phase in which the conceptual solution is proposed, refined and evaluated. Once the design of the product has been completed and accepted, the solution is worked into a machine processable form during the **implementation** (coding) phase. It is ultimately **validated** for acceptance as a working solution. Finally, the software enters a maintenance phase, where problems with the delivered product are corrected and the enhancements to the product (usually resulting from new specifications) are made. A figurative representation of the life cycle is given in Figure 1.1.

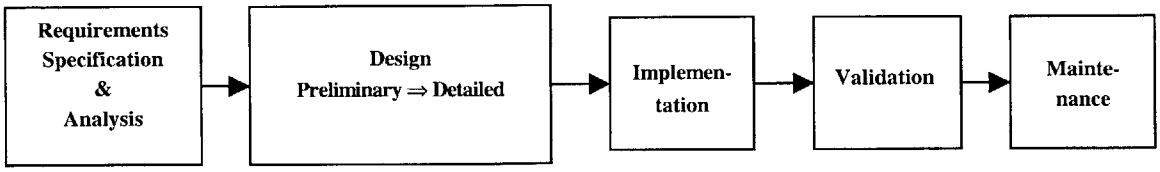

Figure 1.1 - Simplified View of Software Life Cycle
Adopted from [Chandrasekaran et al, 1981]

The above figure represents a simplified view of the actual software life cycle. Realistically, however, there are no clearly separated cycles, where a product will move into a next phase and never return. For example, when trying to write code to implement a particular design feature, it is possible that some problems with the design will be noticed (for the first time). This will require various changes in the software design, thus causing another iteration in the design phase within the same life cycle. Similarly, but worse, this noticed problem might have its roots in the requirements, thus causing an iteration of the requirements specification and analysis. Theoretically, there can be problems first noted in any phase, whose effect can be felt on earlier phases. While iterations for the corrections of these problems may be performed, remaining sections of the initial life cycle will be carried out simultaneously. As a result, the application of the simplified life cycle given in Figure 1.1 leads to a more realistic picture given in Figure 1.2.

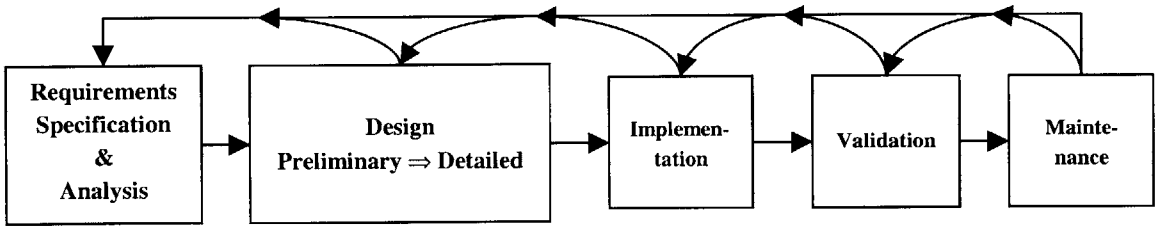

Figure 1.2 - Modified View of Software Life Cycle
Adopted from [Chandrasekaran et al, 1981]

Of the stages in the software development life cycle, validation is of primary interest to the tester. It should be noted that the goal of validation is to establish confidence that the software performs as intended. Although validation is presented in the figures as succeeding implementation, in practice, some sort of validation procedure, such as a review, is present at each phase of the software life cycle.

There are several ways for establishing confidence on a software product through the process of validation. Firstly, it should be possible to **verify** that software is **correct**, preferably by using a formal proof of correctness. Although this approach to software validation has been tried at times, it is made difficult to both programmers and researchers due to the high load of mathematical computations required, and the lack of ideally formalized specification methods against which the proofs can be made.

The second approach in validation is **reliability analysis**. In this approach, the product is monitored to determine its ability to run in a fashion that has been called "without cause for complaint" [Kopetz, 1979]. The main idea is to develop a model that can predict the "mean time to failure" for the system, using past information on recorded errors. On the other hand, the assumptions that must be made to develop the model have been under serious dispute.

The third and the most common approach to software program validation is **testing**. Testing is still under rapid development, with many different testing strategies, each having various weak and strong points.

It should be noted that software testing also aims to achieve similar results with the previously mentioned methods of software validation, namely a certification of software quality. It does not, on the other hand, require the use of formal proving methods, or debugging.

1.2.3 Software Validation and Testing

It was stated previously that, although testing is accepted as a separate task of software development today, it is also a major part of 'software quality assurance' or validation. In this sense, the simple reason to the typical programmer, for testing a software program, is "To see if it works." In practice, the notion of such a "working" program is a complex one which takes into account not only the technical requirements of the programming task but also economics, maintainability, ease of interface to other systems and many other less easily quantifiable program characteristics. A number of different definitions for software quality are offered in the technical literature on program testing. Although the term 'quality' applied to software is still not very well defined, it is a common approach to accept quality as the possession of the characteristics of **correctness** and **reliability**. Furthermore, it is also a safe assumption to say that

these characteristics are checked through methods of software testing today, as opposed to the traditional proving methods of the past.

It is important to note the distinction between correctness and reliability. Correctness is a programmer's notion that the program performs according to the specification. Reliability is more commonly a user's notion that the program does what is desired when desired. Ideally, these notions are equivalent. In contradiction to this, however, correct programs may be unreliable if they are unforgiving of user errors, or if they are used on unreliable systems. If the requirements for the program are too high for the users, this will cause problems. On the other hand, despite the small differences, most papers on the subject of software quality are concerned only with the correctness of the software. Hence, the primary objective in software quality improvement approaches is to reduce the number of errors in the final product. A simplified model of the software development cycle as viewed by this approach is given in Figure 1.3.

A further description of the methodology for software correctness testing, which relies on the comparison of the test results with the expected results derived from specifications, is provided in Appendix A.1.

Similar to reliability and correctness, a third element, the notion of **integrity**, was more recently introduced to quality assurance, and is used for a very similar concept. Integrity is the notion that the software program does what it is intended to do, and nothing more. For software to have integrity, it must possess three characteristics:

1. Evidence that the program is **correct**.
2. Evidence that the program is **robust**.
3. Evidence that the program is **trustworthy**.

For a program to be correct it must provide evidence that it satisfies its mission, requirements, and specifications.

A robust program must include the mechanisms to maintain appropriate levels of performance even under pressures which occur due to operator errors, procedural program flaws, and user initiated errors (e.g. keystrokes or mouse clicks).

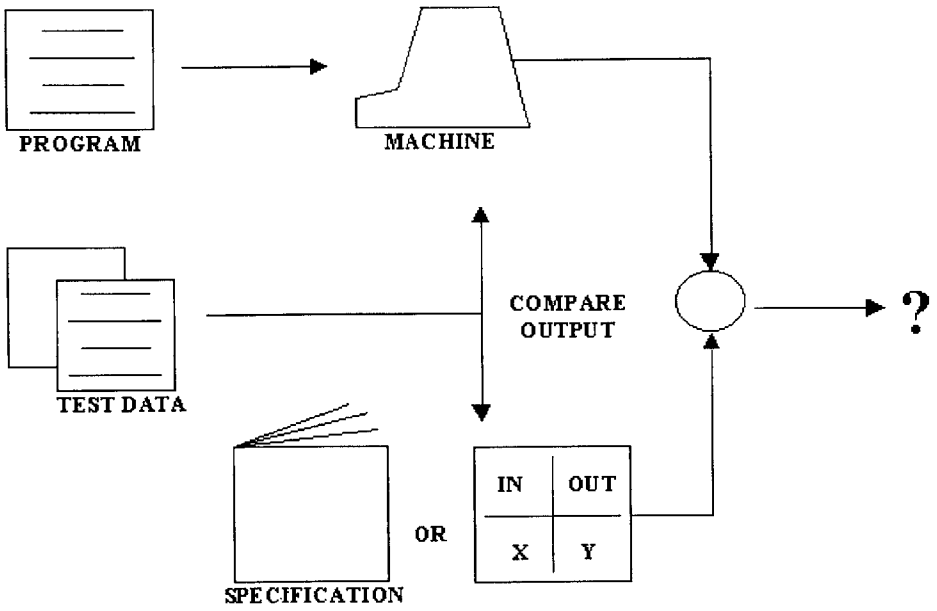

Figure 1.3 - Testing for Correctness
Adopted from [DeMillo et al, 1987]

A trustworthy program must be well documented, functionally simple (as compared to complex), modular in nature, relatively short in length, integrated into as rigorously structured architecture, and produced as a result of good programming practices and sensible programming standards.

A final and more general definition is made for "quality" simply as those properties, which allow the software to meet its minimum level of acceptability.

1.2.4 Testing Techniques & Strategies

In software development, software testing is usually performed after the code has been produced. It has been observed, however, that the later an error is detected, the more expensive it is to correct. This observation encourages testing early in the development of the software.

There are two main different considerations for the early testing of software programs during their development cycle. The first approach considers software development as a statement of

objectives, design, coding, testing, and shipment. In this approach, the inspection of objectives, i.e. design plans, and code, is performed before the code is actually tested, thus without providing inputs into the program to determine the presence of errors. Design, code, and test stages can be repeated until it is believed that the software meets its requirements. The first phase of testing is the **manual analysis** in which the requirement specifications, design and implementation plans, the program itself, and all other available information are analyzed. The second stage is **static analysis** in which the requirements and design documents, and code are analyzed, either manually or automatically. The latter is commonly performed by a typical **compiler**. **Dynamic analysis** is the third stage in which the software is actually executed with a set of specifically selected test data, called **test cases**. Proving the correctness of the program is considered as an optional fourth stage.

It should be noted that more recent literature on software testing consider an additional phase after static analysis, called **symbolic testing**. This technique for testing is based on providing symbolic inputs to the software and "executing" the code by symbolically evaluating the program variables. It does not require compilation of the source code, but requires the selection and use of the appropriate test data.

A second approach considers the life cycle model with respect to two factors: problem comprehension and ease of implementation. Problem comprehension refers to the completeness of problem understanding before implementation begins. If the project has a high technical risk, then considerable analysis is required before the design is complete. Ease of implementation refers to the availability of tools, which will help to support the late changes in requirements.

In this sense, and by considering a broader view, a **strategy for testing** can be defined as the approach the tester uses to select the test data, and to coordinate the testing of multiple components. A good testing strategy should be capable of determining the existence of an error every time one is present. Such a strategy is termed reliable.

Even though all testing strategies are reliable for a correct program, there is no one universally reliable testing strategy capable of determining the existence of errors for any incorrect program or for all incorrect programs. The simple reason is that every testing strategy depends on the selection of a finite set of test data, called the test cases. The effective testing of a software program requires its execution (in addition to other test procedures) either by using a set

of test data tailored to the specifics of that program, or by using all possible test data. The latter option is not applicable to most programs. Consequently, since almost every testing strategy implies the use of a different set of test cases for the testing of a program, and since all programs have different structures and common error types, there is no right strategy for reliable software testing. For the same reason, there is a large variety of testing strategies with corresponding pros and cons. For instance, a software testing strategy designed to find software defects will produce far different results than a strategy designed to prove that the software works correctly. In order to take advantage of all the capabilities offered by different strategies, a complete software testing program uses a combination of strategies to accomplish its objectives.

1.2.5 Test Cases

A test of a piece of software can be thought of as a set of **test cases**. A test case is an instance of presenting the software to some testing device, such as a code reader, compiler, or processor. The resulting output is then evaluated for appropriateness. For almost all software programs, it is impossible to test all possible inputs. The reason for this impossibility, or impracticality at other times is twofold. The primary reason is the huge number of permutations of test case elements, which form the set of all possible inputs. This results in an extremely high number of test cases, which requires too much effort to input fully into the system for testing. The second reason is the high time consumption in the execution of some test cases. Some tasks performed by software programs occasionally take very long, and therefore multiple test cases may reach beyond available time resources. A simple example can be set by considering a program containing three inputs, each of 16 bits, which requires a constant time of 100μs to produce the desired output for a given input. The complete testing of the set of permutations for three 16-bit data groups would require an approximate 900 years of CPU time. As a result, effective ways of selecting a relatively small set of test cases capable of revealing errors and providing a maximum degree of confidence must be found.

1.2.6 Test Plans

Testing of large software programs requires an effective selection of testing strategies, techniques, and a careful preparation of test data. Given all different types of common error

expected in a large program, and the high number of different execution paths, it is obvious that an acceptable validation of such programs is a complex job. Therefore, the management of all the different parameters in the testing task of such a program requires a well-documented planning system. For this reason, **test plans** should be created during the software development phases prior to testing.

These plans aim to identify the test schedules, environments, resources (i.e. personnel, tools), methodologies, cases (inputs, procedures, outputs, and expected results), documentation and reporting criteria. They are prepared in connection with each of the specification phases, such as requirements, design, and implementation; and are correlated to the other project plans including the integration plan. Individual test cases are definitively associated with particular specification elements and each test case includes a predetermined, explicit, and measurable expected result, derived from the specification documents, in order to identify objective success or failure criteria.

Test plans identify the necessary levels and extent of testing as well as clear pre-determined acceptance criteria. The magnitude of testing is presented as linked to criticality, reliability, and/or safety issues. Test plans also include criteria for determining when testing is complete. These test completion criteria include both functional and structural coverage requirements. According to this, each externally visible function and each internal function is tested at least once. Furthermore, each program statement is executed at least once and each program decision is exercised with both true and false outcomes at least once. Finally, the test completion criteria also include measurements or estimates of the quality and/or reliability of the released software.

1.2.7 Other Documentation for Testing

The effective management of all the aforementioned parameters in testing also requires the documentation of all reported test results in addition to the test plan. These test results (inputs, processing comments, and outputs) are documented in a manner permitting objective pass or fail decisions to be reached, and in a manner suitable for review and decision making subsequent to running of the tests. Similarly, test results are documented in an appropriate form, which can be used for subsequent regression testing. Errors detected during testing are logged, classified, reviewed and resolved prior to release of the software. It should also be noted that all test reports must comply with the requirements of the corresponding test plans.

1.2.8 Test Personnel

For satisfactory testing of any software program, sufficient personnel should be available to provide necessary independence from the programming staff, and to provide adequate knowledge of both the software application's subject matter and software or programming concerns related to testing. This condition is of utmost importance, since it has been proven through past applications that in the traditional testing structure, the programmers that perform all testing duties are able to recover only a small percentage of errors compared to today's testing methods [Infotech, 1979]. There are certain alternatives to this approach, such as the use of various techniques (i.e., detailed written procedures and checklists) by small firms in order to facilitate consistent application of intended testing activities and to simulate independence [Center for Devices and Radiological Health web page, 1997].

1.3 History & Evolution

The history of software testing stems from the evolution of computers, and computing. It starts with the introduction of the computing concept, where for the first time, a universal computer can be programmed to execute a given algorithm. The next stage is the development of these algorithm descriptions, which is classified as "software" in the following years. The third stage is the creation of the software-engineering concept, bringing a structured form into the development of software programs. The final stage is the creation of standards for software quality, and later its diversification into software proving and testing.

1.3.1 Brief History of Computing

Calculation was always a need from the early days, in order to maintain inventories (of flocks of sheep) or reconcile finances. Early man counted by means of matching one set of objects with another set (stones and sheep). The operations of addition and subtraction were simply the operations of adding or subtracting groups of objects to the sack of counting stones or pebbles.

Early counting tables, named **abaci**, not only formalized this counting method but also introduced the concept of positional notation that we use today [Computing History web page, 1997].

The next logical step was to produce the first "personal calculator" called **the abacus**, which used the same concepts of one set of objects standing in for objects in another set, but also the concept of a single object standing for a collection of objects, i.e. positional notation [Computing History web page, 1997].

The first use of decimal points, invented logarithms, and of machines for simple multiplication operations started in the early 1600s. In 1642, Blaise Pascal created an **adding machine** with an automatic carrying system from one position to the next. The **first calculator**, which was capable of multiplication was built by Gottfried Leibniz in 1673 [Computing History web page, 1997].

The use of **punch cards**, as the primary means of changing patterns on a machine, began in 1801 in France, in the control patterns of fabrics. These punch cards were also the first representation of any form of **machine code**. The use of this simple automation caused the first riots against a risk of replacement of human workers with machines [Computing History web page, 1997].

In 1833, Charles Babbage designed the **Analytical Engine**. This design had the basic components of a modern computer, and led to Babbage being described as the "Father of the Computer". In 1842, Ada Augusta King, Countess of Lovelace, translated Menabrea's pamphlet on this Analytical Engine, adding her own notes, and thus becoming the world's first programmer [Computing History web page, 1997].

The American Institute for Electrical Engineering (AIEE) was founded in 1884, representing the first form of any **standardization** in the areas of computer and software engineering. AIEE was the first of a group of organizations, which eventually merged to form the IEEE in 1963. The Institute of Radio Engineers, founded in 1912, and the AIEE Committee on Large-Scale Computing Devices, founded in 1946 were the others [Computing History web page, 1997].

Due to the increasing population in the US, and the demands of Congress to ask more questions in the 1890 census, **data processing** was actually required for the first time. Herman

Hollerith won the competition for the delivery of data processing equipment, and went on to assist in the census processing for many countries around the world. The company he founded, Hollerith Tabulating Company, eventually became one of the three that composed the Calculating-Tabulating-Recording (C-T-R) company in 1914, and eventually was renamed IBM in 1924 [Computing History web page, 1997].

In 1930, Vannevar Bush, MIT, built a large-scale **differential analyzer** with the additional capabilities of integration and differentiation, which was founded by the Rockefeller Foundation. This was the largest computational device in the world in 1930 [Computing History web page, 1997].

The first large scale, automatic, general purpose, **electromechanical calculator** was the Harvard Mark I (AKA IBM Automatic Sequence Control Calculator - ASCC) built in 1944 [Computing History web page, 1997].

1.3.2 Programming, Verification, and Testing

In 1937, Alan Turing developed the idea of a "**Universal Machine**" capable of executing any describable algorithm, and forming the basis for the concept of "**computability**". This was the first realization of changeable code, following the hard-wired machine code introduced in 1801. More importantly, Turing's ideas differed from those of others who were solving arithmetic problems by introducing the concept of "**symbol processing**", which was an even bigger leap in the formation of computer software [Computing History web page, 1997].

The first bug was discovered in 1945. Grace Murray Hopper, working in a temporary World War I building at Harvard University on the Mark II computer, found the **first computer bug** beaten to death in the jaws of a relay. She glued it into the logbook of the computer and thereafter when the machine stopped (frequently), it was reported that they were "**debugging**" the computer. Edison had used the word bug and the concept of debugging previously, but this was the first verification that the concept applied to computers [Computing History web page, 1997].

In 1947, William Shockley, John Bardeen, and Walter Brattain invented the "transfer resistance" device, later to be known as the **transistor,** which revolutionized the computer and

gave it the reliability that could not have been achieved with vacuum tubes. This was another breakthrough for computers, resulting in the initialization of an era, in which computing units of high capacity were developed rapidly and for lower costs [Computing History web page, 1997].

In 1948, the Selective Sequence Control Computer (SSEC) was built for IBM. Though not a stored program computer, the SSEC was the first step of IBM from total dedication to punched card tabulators to the world of computers. Following this step, in 1949, the first large-scale, fully functional, **stored-program electronic digital computer** was developed by Maurice Wilkes and the staff of the Mathematical Laboratory at Cambridge University. It was named EDSAC (Electronic Delay Storage Automatic Computer); the primary storage system was a set of mercury baths (tubes filled with mercury) through which generated and regenerated acoustic pulses represented the bits of data [Computing History web page, 1997].

In 1954, John Backus proposed the development of a **programming language** that would allow users to express their problems in commonly understood mathematical formulae, later to be named **FORTRAN**. In 1957, Backus and his colleagues delivered the first FORTRAN **program compiler** for the IBM 704, and almost immediately **the first error message** was encountered; a missing comma in a computed GO TO statement. In the same year, Herbert Bright at Westinghouse in Pittsburgh received an unmarked (a first for a user) 2000-card deck. By using the long expected FORTRAN compiler, he created the **first user program**, complete with an error. This step was an important step upward for the world of programming languages, from a domain in which only specially trained programmers could complete a project, to a domain in which those with problems could express their own solutions [Computing History web page, 1997].

In 1955, during the development of the programming language FORTRAN, Harlan Herrick introduced the **high-level language** equivalent of a "jump" instruction in the form of a "GO TO" statement.

In 1973, Don Knuth delivered a dozen volumes on the "Art of Programming"; the first three of which formed the basis of software development for many years. These volumes contained many of the basic algorithms of the field that became known as "**data structures**" and many of the techniques of programming that became the basis of "**software engineering**" [Computing History web page, 1997].

In 1974, Intel introduced the 8080 for the purposes of controlling traffic lights, which found fame later as the processor for the Altair [Processor Hall of Fame web page, 1999]. This was the initial launch of the **Personal Computer** era, which led to higher and cheaper computing power, and drastically improved software programs reaching down to all individuals.

1.3.3 Evolution of Software Testing

Through the past thirty years, software testing has developed from a simple and rather inconclusive exercise carried out by the programmer as the final stage of the coding process, into a formal, systematic methodology which is applied to all stages of software development, and by dedicated test engineers.

Starting from the early days of software testing, there have always been arguments about the use of practical testing as opposed to that of theoretical proving. It was decided early at some point that testing is the empirical form of software quality assurance, while proving is the theoretical way [Infotech, 1979].

Early efforts in the testing of software programs were based on executing the program for a large (where large was not formally defined) number of test cases intended to exercise different control paths through the program. For proving program correctness, the technique of inductive assertions was used. These techniques were successful for locating a good portion of errors that were present in these early programs, despite the low efficiency compared to the techniques used today. As a result, there have been some software programs tested with these techniques that have worked perfectly, as well as some failures in other programs that were also tested with these techniques. Among these failures, was also the famous missing comma in the lunar landing software, where the structural faults had once led to the detection of the moon as an incoming inter-continental ballistic missile [Miller, 1978]. A standard feel of confidence, however, was not established during these earlier periods.

In the later stages, the importance of automation in testing increased in parallel to the need for software program development, due to a higher demand for more reliable testing. Since then, automated tools for testing have been developed, providing environments for the testing of

individual modules and for systematic symbolic-execution of the various paths of a program. The nature of testing changed to a more theoretical basis, as opposed to the more pragmatic techniques of the past. This development of a systematic view of software testing was encouraged by the general adoption of software engineering as a profession by **The Institute of Electrical and Electronics Engineers, Inc. (IEEE)**, on May 21, 1993, which aimed to formalize the whole software development process [Establishment of Software Engineering as a Profession web page, 1999].

At 1968 NATO conferences [Science Program web page, 1999], which established the concepts of software engineering, a call was made for a radical improvement in methods of ensuring that software is consistent with its specification. It was felt that traditional testing methods were inadequate, and attention should be turned to developing proving techniques.

In today's world of software quality assurance and testing, the standards are both higher in comparison to the past, as well as of a wider scope. The computer world has changed beyond recognition in the 1990's, which opened with the introduction of the revolutionary 386 as the generic processor, an era which is now ready to be closed with the introduction of the GHz processor [Pentium III Xeon web page, 1999]. (It should be noted that processors of such capacity did exist in the 1980's, but ownership required a special manufacturing technique, a cooling system of significant capacity, an investment of a few million dollars, and special permission from the US government due to national security reasons [Infotech, 1979].) Due to the dramatic increase in hardware capacity, demands on software programs have also been greatly increased. The booming of the computer sector has reflected in the software industry, thus increasing competition, and bringing about a need for fast and reliable software production. Use of new communication channels such as new communication protocols or Internet-based computing has expanded the software industry into new dimensions. The result is a fast developing group of improved software development methods, along with their components of software quality assurance and testing.

1.4 Motivation for Testing

The basic motivation in testing comes from the structural changes in the computer and the software industries. Having looked at thirty years ago, we can see that most of the money spent on computer systems of that time was spent on hardware. Since hardware costs were significantly high compared to today, only specialized firms and institutions were able to purchase computer systems. This meant two things. Firstly, the software programs used by these companies and institutions were custom made for their specific needs, due to the high prices of the systems, as well as the relatively high prices of the programs. Secondly, these organizations had technicians and other maintenance personnel to operate and maintain their hardware and software. Consequently, fixing problems in these old systems was easier, and could be accomplished both by the employed personnel, or by the software firm which custom made the program.

In today's computer industry hardware prices are extremely low, within the reach of almost anyone. As a result, the major use of computers is at the personal level, rather than organization wide complex systems. Most software companies target either the consumer, or off-the-shelf commercial markets, and software programs are no longer tailored to specific needs except special orders totaling in very large amounts. These software programs, which are sold in large quantities at low prices, are used by people having less information in the area. Furthermore, these people have a lot less direct contact with the software companies, compared to the past. This change in the structure of the market implies a change in software requirements, and a demand for more reliable software. Due to this implication, both users and designers of software products have a need for techniques applicable for the quantitative assessment of software reliability, as well as for prediction of software quality and reliability.

A primary motivating force for improving software-testing methods today is the cost factor. It is known that with today's strict specifications for software quality and reliability, a large proportion of the total software development cost, of up to 50% of the whole, is incurred by the testing process [Infotech, 1979]. Furthermore, a second sizable proportion is incurred by the inefficiencies of running software, which still contains errors after being tested. These results explain the increasing need for effective software quality assurance at affordable prices.

Similarly, the need for early testing comes about through a desire to minimize testing costs. History shows that errors discovered late within the development cycle are significantly more costly to repair than those discovered early. Requirements and design errors, for example, typically take longer to discover, and are more costly to repair. The complexities and cost of testing also increase significantly during the later phases of software development. Any testing which can be reduced or eliminated by a more reliable previous phase can significantly reduce overall testing, and thus development costs.

CHAPTER 2

Analysis of Software Programs

2.1 General Overview

The traditional methods used for assessing software quality are *software evaluation* and *software testing*. Software evaluation examines the software and the processes used during its development to see if stated requirements and goals are met. In this approach, the requirements and design documents, and the code are analyzed, either manually or automatically, without actually executing the code. Software testing assesses quality by exercising the software on representative test data under laboratory conditions to see if it meets the stated requirements.

The older software testing and analysis methodology used to classify testing techniques into two groups, as **static analysis** and **dynamic testing**. Later classifications added a third group called **symbolic testing**, which formed a transition between the initial two. Dynamic testing techniques constitute the software testing part of validation, where static analysis techniques form the software evaluation part. It is still under argument as to which side symbolic testing performs, along with other inductive assertions techniques, but because it involves the execution of the code, even if manually, most approaches take it into consideration as a dynamic testing method. It should further be noted that some dynamic testing techniques also make use of symbolic testing, and even of static testing at times.

Of the given three types, **static analysis** involves the detailed inspection of program specifications and code to check that an adequate standard has been reached in the

implementation. Its function is to detect structural faults, syntax errors, and to check that programming standards have been followed. Since some errors cannot be found without executing the code, not all errors can be detected by static analysis.

Symbolic testing, involves the execution of a program, using symbols, as variables representing different value possibilities, rather than the actual data. In this way, the programmer does not have to select values to produce all of the required inputs.

Dynamic testing, is the process of running programs under controlled and observable circumstances, such as execution over selected test data, and comparing the resulting output with the expected.

A different classification scheme exists for general testing techniques, with respect to the level of software testing. The first phase, which is called **alpha testing** is the in-house testing of the software throughout the software development cycle. The second phase is the **beta testing**, which exposes a new product, which has just emerged from in-house (alpha) testing, to a large number of real people, real hardware, and real usage.

Beta testing is not a method of getting free software long-term, because the software expires shortly after the testing period.

2.2 Static Analysis

The main goal of static analysis is to evaluate a given program on the basis of its structure. In this technique, the requirements and design documents and the code are analyzed, either manually or automatically, without actually executing the program code. It is believed to aid testers in checking (predicting) program behavior in situations that cannot be easily achieved during real execution, such as due to the asynchronous actions of physically separated processes.

One of the underlying facts that give static analysis its importance in software validation is that the cost of reworking errors in programs increases significantly with later discovery of these errors. For this reason, static analysis is a very important technique for early and cost effective correction of errors in the system. Experimental evaluation of **code inspections** and **code**

walkthroughs have found static analysis to be very effective in finding logic design and coding errors, with a statistical success rate of 30% to 70% [DeMillo et al, 1987]. It was further found out that walkthroughs and code inspections were able to detect an average of 38% of the total errors in these studied programs. Given that the software testing costs for typical large-scale software systems have statistically been shown to range from 40% to 50% of overall development costs [Infotech, 1979], static analysis saves an average of 17% of the overall costs.

In addition to the obvious cost savings, when performed prior to dynamic testing, static analysis can provide information useful for test planning, reduce the number of test cases required, and increase confidence in test results. Because static analysis is concerned with analyzing the structure of code, it is particularly useful for discovering logical errors and questionable coding practices. For example, an uninitialized variable is an error of logic, while a variable that is set but never used may not denote an error, but is a questionable coding practice.

The fact that the program code is not exercised during static analysis is beneficial in certain points, while adding limitations at others. A major limitation of static analysis concerns references to arrays, pointer variables, and other dynamic constructs. Static analysis cannot evaluate references or pointers, because these are mechanisms for selecting a data item at run-time, based on previous computations performed by the program. As a result, static analysis is unable to distinguish between elements of an array, or members of a list under such circumstances. Although it might be possible to analyze references and pointers using techniques of symbolic execution, it is generally simpler and more efficient to use dynamic testing. In this sense, dynamic testing can often be used to obtain information that is difficult or impossible to obtain by static analysis.

2.2.1 Requirement Analysis

The first stage of static analysis of a software program is the **requirement analysis**. These requirements are user defined, and usually use a checklist of correctness conditions, including such properties of the requirements as consistency, necessity to achieve goals of the system, and feasibility of implementation. The requirement documents are usually written in a requirement specification language, and then checked by the analyzer (test engineer).

2.2.2 Design Analysis

The second stage of static analysis is the **design analysis**. This analysis is usually carried out by using a similar checklist to the one used in the requirement analysis. The controlled features are the design elements of the software system, such as the algorithms, data flow diagrams, and the module interfaces. The common practice is the use of a different method for the analysis of each different element. For instance, the consistency of module interfaces can be determined by comparing the common parts of different design elements. For the formal analysis of some design elements, an inductive assertions method, such as symbolic testing can be used.

2.2.3 Code Inspections and Walkthroughs

The final stage of static analysis consists of **code inspections** and **walkthroughs**. They involve the visual inspection of the program by a group of people, first individually, then as a group, in order to detect deviations from specifications. This final stage can be performed with the assistance of a static analyzer, who can control the data flow of the program, and record certain problems in a database. (Some problems such as uninitialized variables, statements that are not executed, and inconsistent interfaces among modules will require the use of a database or a similar recording tool, since they are likely to be in high numbers yet still individually critical.)

A **code inspection** is a set of procedures to detect errors during group code reading. Two things take place during a code inspection session. Firstly, the programmer narrates the logic of the program statement by statement. After the narration, the program is analyzed using a checklist for common programming errors such as computation errors, comparison errors, and unexplored branches. Because the code is analyzed for each item on the checklist, repetitions of the analysis on certain program sections and skipping of certain other sections may occur.

A **walkthrough** operates similarly to a code inspection, but employs different error detection techniques. During the group meeting for a program code walkthrough, a small set of test cases is walked through the logic of the program by the participants. The walkthrough does not involve explanations about the code, or the checking of common programming errors from a checklist. Instead, a dynamic testing procedure is modeled, in order to run through the code with the

provided test cases and to locate the errors as well as their causes. This function also eliminates the need for additional debugging after this analysis.

2.3 Symbolic Analysis

Symbolic analysis is a derivation of the highly structural static analysis approach, towards a more practical dynamic analysis approach. It appears as a compromise between verification and testing. It is more demanding than conventional testing in that it requires symbolic interpretation (either manual or computer-based interpreter), and someone to certify that symbolic results are correct. On the other hand, it provides much higher confidence in a program's correctness, in that each symbolic test may represent an arbitrary number of conventional tests.

Symbolic analysis, hence its name, uses symbolic variables and data during the testing of a software program. These symbolic variables may be elementary symbolic values or expressions. The usual approach in the symbolic analysis of a software program is to execute the program on a set of input values, similar to dynamic testing in that sense, and to examine the output of the program. The examination of the output is performed most likely by the programmer, which in turn determines the correctness of the program behavior. In contrast to dynamic testing, the input values may consist of symbolic constants, and the output values may consist of symbolic formulae or symbolic predicates. The symbols used as input data may represent a group of variables instead of only one (such as X being used for any integer between 0 to 100), symbolic testing of one such test case will be effectively equal to repetitive dynamic testing of the program with different input data. This property makes the proving of a program easier using symbolic testing. Secondly, since symbolic testing runs with a manual execution, it can still discover many of the errors that can be overlooked by dynamic testing, but can be found by static analysis.

An unfavorable consequence of symbolic execution is that the symbolic representations can sometimes be too long or complex to convey any meaning. Even when a symbolic representation is short, the tester may not be able to detect the error in a representation, as opposed to comparisons of numerical outputs in dynamic testing. Another major drawback of symbolic testing is that it must be applied to completely specified program paths including the number of

iterations for each loop. In general, a program may have an infinite number of program paths and only a subset can be chosen for practical analysis.

2.4 Dynamic Testing

2.4.1 General Overview

Static and symbolic analysis techniques have been introduced as efficient methods of reducing total testing effort requirements and overall software development costs. On the other hand, it should be noted that these two are methods created only to clean the excess part of errors in a software program code, prior to the initialization of dynamic testing. The reason for such a mechanism is simply that manual execution processes in relatively large software programs often become significantly complex, way beyond the reach of human capabilities. This is also the same reason computers and software programs are used in the first place. As a result, there can be no better technique to test how a program runs, than to run the program repetitively with a good selection of test data.

In view of the above, dynamic testing can be defined as the process of running programs under controlled and observable circumstances, such as execution over selected test data, and comparing the resulting output with the expected. This small set of test data is selected from the much bigger (often-infinite) set of all possible test data applicable for the software program. Furthermore, the selected set of test cases to be exercised during experiments adequately represents all realistic situations of the program behavior, and is usually defined with regard to some specific *test coverage* criteria determining when to stop testing. Each test case is exercised with specially selected test data, which must be provided in place and on time, as explicitly determined by the related *test scenario*. A test case usually contains the following information [Krawczyk, 1998]:

- A unique identifier to distinguish test cases from one another.
- A list of statements exercised by the platform function calls, application processes and computing environment services when executing the given test case.

- Names and settings for all required inputs (variables, conditions, and states).
- Predicted outputs.

Following the execution of the prepared test cases according to these specifications, information about the observed program behavior is recorded to a special *log file*. Such a log file typically collects the following information:

- Time of making the record, relative to some characteristic point, e.g. starting or completing execution of the program under test.
- Execution context of the program when making the record, e.g. the content of relevant program counters, specific input and output conditions, etc.
- Values of the related data objects, i.e., the content of local and global variables pertinent to the recorded event.

Finally, the actual program outcome recorded in the log file is compared to the expected one. In any case of disagreement, error identification is initiated, otherwise a next test case is exercised until satisfying the test stopping criteria.

2.4.1.1 Testing & Debugging

Two different main parts of dynamic analysis can be defined. The first one is the testing to obtain results of program execution suitable for code evaluation from an acceptance point of view. The second part is the debugging to identify the reasons for wrong program behavior in order to eliminate defects from the program code. These two parts support one another, forming a specific cycle shown in Figure 2.1 [Krawczyk et al, 1998].

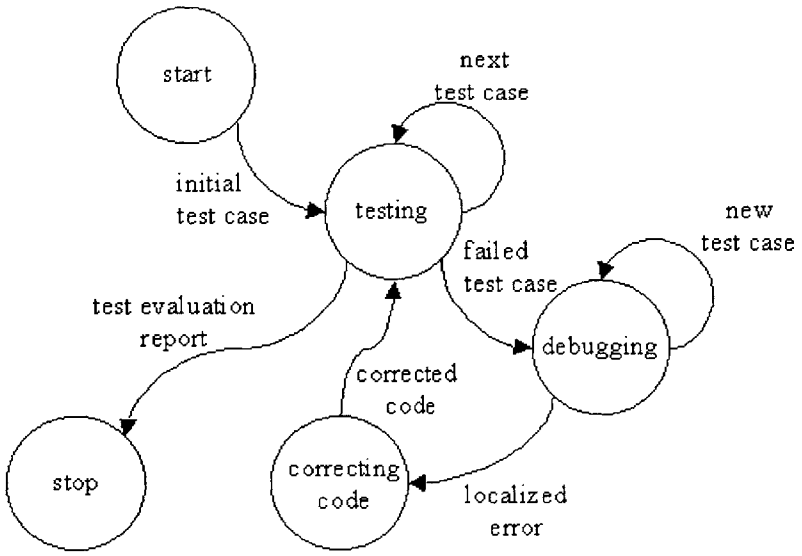

Figure 2.1- The testing-debugging cycle

Figure adopted from [Krawczyk et al, 1998]

As mentioned earlier, the term bug is commonly used to refer to an error in a software program, but can also be attributed to a poorly prepared and executed test, or a bad program understanding. There are three generic debugging strategies, each one approaching the problem of hypothesis on error location in a different way. Upon spotting a bug, the relevant test case is used as a starting point for generating new test cases to be run one by one until a hypothesis on error location is:

- *Gradually built* in a series of experiments demonstrating good and bad program behavior.
- *Identified* from the bunch of ad-hoc hypotheses formulated immediately upon the first bug manifestation.
- *Backtracked* by replaying the related program behavior in a series of experiments, each time exercising the original test case one step before the last point of the bug manifestation.

If the final hypothesis on error location is verifiable, and the error can be localized, the program source code is corrected and the new executable code is generated. Otherwise, more experimentation (and probably more comprehension of the program being debugged) is required.

When the error is finally eliminated, regression analysis is required to determine what test cases should be repeated to show the elimination of detected errors.

2.4.1.2 Unit Testing

Given the changes in the computer industry throughout the last decade towards significantly greater computing power and a matching fast production, and hence raised standards of software quality, the design and development of a typical program has become a very complex task today. Since the beginning of the formation of this complex software development structure, a need to develop software in stages was present. With the introduction of program modularity with object oriented programming languages having reusable classes, this need became better defined as a need to employ top-down software development. Today, top-down and bottom-up testing strategies have become popular, making a well-balanced distribution of simultaneous testing tasks among test engineers inside a team possible.

Composite testing is the result of such a demand, which consists of breaking up the system into several components and testing these separately. It can be further classified into **unit (formerly module) testing** and **integration (formerly subsystem) testing**, where unit testing has the goal of discovering coding errors in the individual units (modules) of the system, while integration testing tests subsystems that consist of the tested modules. **System testing** is the complementary strategy, which tests for subtleties in the interfaces, decision logic, control flow, recovery procedures, throughput, capacity, and timing of the entire software system.

Unit testing is applied to coded and compiled programming units that have to be verified against their functional specifications or intended algorithmic design structure. This is where the most detailed investigation of the programmer's work can be done by the testers. A unit (or module) can be defined as a set of one or more continuous program statements having a name by which other parts of the system can invoke it. A unit will preferably have its own distinct set of variable names.

Unit testing involves the process of testing the logical units of a program (such as procedures or subprograms) individually, and later integrating the individual unit tests to test the overall system in the later stages of dynamic testing. In order to perform unit testing two things need to

be considered: the design of test cases and the coordination of testing multiple modules. Test cases may either be constructed from the program specifications or by analyzing the module code (**black-box** and **white-box** testing respectively). There are two approaches to combining tested units, namely incremental and non-incremental approaches, which have been addressed in the Integration Testing section.

Unit testing forms the basis of dynamic testing, and therefore requires a planned and systematic approach for assurance of unit reliability. Only satisfying the relevant test completion criterion applied to the unit under test means passing the unit test, and implies its readiness for integration with other units that have passed their respective unit tests. If during integration, errors are detected, the unit is debugged and tested again.

2.4.1.3 Integration Testing

Integration testing involves two or more tested units when they are merged into a larger structure. These larger structures are generally called subsystems, which consist of several modules that communicate with each other through well-defined interfaces, and integration testing actually tests the operation of these interfaces.

The focus is on two aspects: the interface between units, and a composite function jointly represented by the larger structure of units. Essentially test cases and test completion criteria for integration testing are the same as in unit testing, as the larger structure under test can be considered simply as a larger unit. However, localization and elimination of bugs require not only a comprehensive understanding of the composite object's structure but knowledge about the way the object has been integrated using smaller units.

Rules for merging smaller units into larger bigger objects constitute a specific integration strategy. In general, these strategies are of two types:

1. *Incremental Strategies* – which merge just one unit at a time to the set of previously tested units.
2. *Non-incremental Strategies* – which group some (or all) units to test them simultaneously.

Intuitively, incremental integration strategies are more systematic and helpful in localizing errors, but this does not mean that they are better in detecting errors when compared to non-incremental strategies.

Depending on the order followed throughout the program hierarchy when determining the level of the next unit to be merged into the subsystem, variations of incremental integration testing exist. **Top-down** and **bottom-up** testing are two different strategies for incremental testing, which use the opposite directions for level increments. **Thread testing** is a third incremental strategy, which is modified from the black-box approach to unit testing, based on a system verification diagram derived from the requirements specification.

2.4.1.4 System Testing

System testing provides the necessary evidence to support a decision whether all system components are connected and working together as specified in the relevant requirement and design specifications. Unlike integration testing, it does not consider the system structure and views it as one entity. It also attempts to assess non-functional properties and characteristics of the system. It does not matter at this level of testing how well the system has been tested so far. If it turns out that the system (program) does not solve the application problem for which it has been designed, then system testing fails. Like any other kind of testing, system testing looks for errors. Test cases are selected based on system definition or requirements specification, with the necessary involvement from the user's side.

2.4.2 Testing Strategies

The general structure and relations of dynamic testing strategies can be represented as shown in Figure 2-2. Some of these strategies have been discussed previously.

Dynamic testing is classified as component and system testing. Component testing is comprised of unit and integration tests. Unit testing is the process of testing logical units of a program individually, and integration testing is the coordination of the individual module tests to

evaluate subsystems. System testing does the same for the overall system. Considering the design of test cases, they may be constructed from specifications (black-box), or by analyzing the module code (white-box). Considering the coordination of the tested units, there are two main approaches, namely non-incremental and incremental. Top-down and bottom-up are two incremental testing strategies for testing a software system composed of hierarchic modules, where thread testing is another such strategy based on system requirements specification.

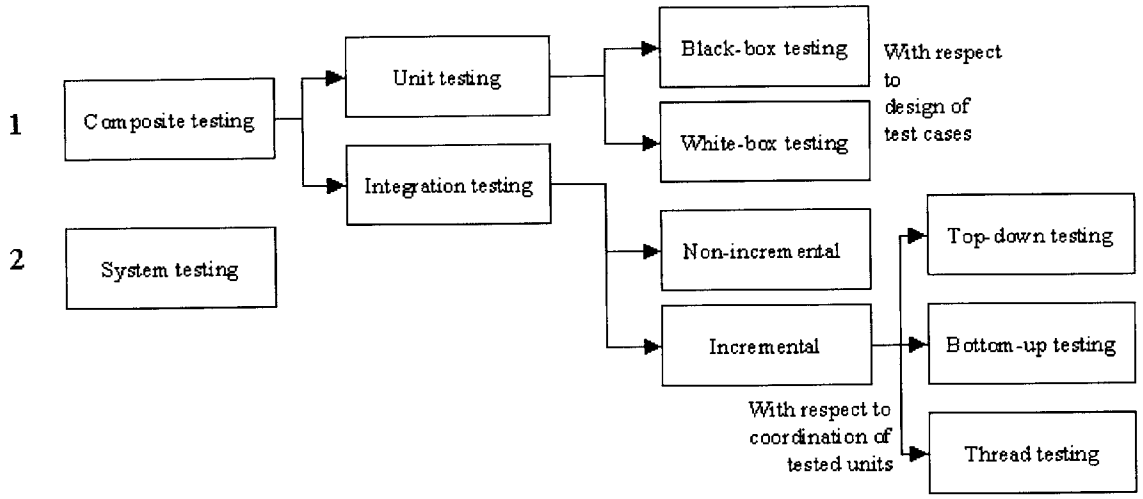

Figure 2.2 - Testing Strategies

2.4.2.1 Black Box Testing

Test data selection techniques can be partitioned based on whether or not they involve knowledge of the specific program structure. Those, which do not make use of this structure, can be described as **black box** testing strategies. They view the software as a "black box", which accepts certain input and in effect produces certain output. Selection of test data using this approach is usually based on the requirements specifications for the software.

In **black-box** (or functional) testing, the internal structure and behavior of the program is not considered. The objective is to find out solely when the input-output behavior of the program

does not agree with its specifications. As mentioned above, the test data for the program are constructed from its specifications, and it is required that the tester be very familiar with these specifications. It is quite often the case that numerous test cases are included within the program specifications.

In order to minimize the number of test cases, the input space (set of possible inputs) of a program are classified into various classes with respect to the program's input specifications. These classes are called equivalence classes, and each equivalence class is a test case covered by one input specification. To further simplify the partitioning operation, given the specifications of a program are described by a formal specification language, these specifications can also be partitioned into equivalence classes (subspecifications). It should be noted that partitioning for black-box testing can also be performed in the output space of the program, instead of the input space.

As a second method of selecting test data, test data that lie on and near the boundaries of input equivalence classes may be selected. This method, in fact, works in a complementary way with the first, increasing reliability in the input space partitioning approach by testing its transition points into other partitions. The drawback with these approaches, however, is that the total number of input data combinations is generally very large. Due to this inefficiency in the partitioning approach, several other ways of black-box test data selection are suggested [DeMillo et al, 1987].

Error guessing is a test data generation technique in which possible errors are listed and test cases based on this list are constructed. Since this is a largely intuitive process, a procedure for test data generation cannot be given.

Random testing is another black-box testing strategy in which a program is tested by randomly selecting some subset of all possible input values. This strategy is considered to be the poorest in test case design methodology, due to its lack of a defined system resulting in a very low efficiency. Since the test cases are selected entirely at random, the tester virtually has no control over the testing process other than to record outputs and compare these with the expected. On the other hand, the use of random test data generation is recommended for the final testing of a program by selecting test data from an "expected run-time distribution" of its inputs. In

agreement with this, the methodology of generating random test data can also be used for real-time testing.

Finally, test data selection can be performed automatically. There are various test data generation tools used for this purpose.

Black-box approach in test case selection is preferred primarily due to its program independence. This gives the tester a different perspective to the program. Since the tester cannot know what the program code was intended to do, he/she can select test cases that are closer to the expectations of the user, which are represented in the program specifications. On the other hand, black-box testing also has various limitations. The main drawback is the inability of black-box to determine whether the set of selected test cases is complete. There is no way of seeing whether or not the selected test cases will exercise all the critical statements in a program code, without actually exposing the tester to the code. In order to be able to establish 100% confidence in module correctness through black-box testing, every possible input has to be exercised as a test case, which is often impractical if not impossible.

The second drawback related to black-box testing is its dependence on the correctness of the program specifications. This further decreases the reliability of this testing approach due to the mentioned dependence on another part of the software, which is designed within the software development cycle and is therefore prone to errors.

2.4.2.2 White Box Testing

The alternative to the black box approach is to base the selection of test data on specific elements of the program structure. This technique, described as **white box** testing, most often involves the selection of inputs, which will cause certain specified statements, branches, or paths in the software to be executed. In most testing application, the two strategies are used in combination, applied to different parts of the program. This approach is further termed **gray-box** testing.

In white-box (or structural) testing, the tester has access to the source code and all other documentation. Tests are planned to demonstrate the functions of the software from analysis of

the code. The structure of the program is examined and test data are derived from the program logic. There are certain criteria to determine this logic. One such criterion is to require that every statement in a program be executed at least once. This criterion is necessary, but is in no way sufficient since some errors may still go undetected.

On the other hand, a testing strategy, which is not capable of executing every statement of the software program at least once on one of its test cases, would be incapable of finding errors in the unused statements. For example, in testing of the program code segment in Figure 2.3, if the condition evaluates "true" for every selected test case, the last line of the given code, and thus the appropriateness of the statements in block 2 will never be tested for errors. As a result of this redundancy problem, many testing strategies seek 100% statement coverage [DeMillo, 1987].

```
.
.
// If / else statement given below
    if (condition)
            { block 1}
    else
            {block 2}
```

Figure 2.3

There is also a more elaborate type of coverage of the program structure called **branch** or **decision coverage**. In branch coverage, each branch in the program must be traversed at least once, where a branch corresponds to an edge in the program's flowgraph. It requires every possible outcome of all decisions to be exercised at least once. This means, for example, that the true branch and the false branch of every predicate must be exercised at some point in the testing procedure. It includes statement coverage since every statement is executed if every branch in a program is exercised at least once. The problems associated with this criterion are as follows:

- A program may contain no decision statements.
- A program may contain multiple entry points.
- Some statements may only be executed if the program is entered at a particular entry point.
- If a program contains exception handling routines, these routines may not be executed at all.

While this approach solves the problem that was present in Figure 2.3, it creates a different redundancy problem, shown in the program segment in Figure 2.4.

```
.
.
// If / else statement given below
    if (x < 0)
            { x = x – 1;}
    else
            {x = x + 1;}
    if (y < 0)
            { z = Math.sqrt(-x);}
    else
            {z = x + y;}
```

Figure 2.4

Assume that the sqrt function expects a nonnegative input. According to the branch coverage approach, we can achieve coverage by selecting an input that follows the "if" branch of each predicate, and another input that follows the "else" branch of each predicate. By doing this, however, we miss the problem with the square root operation, which will give an error for if $x \geq 0$; that is, if we had followed the "else" branch of the first predicate. Therefore, branch coverage, too, is in general an unreliable strategy.

In order to prevent a problem similar to that in the previous example, it would be necessary to require that each **path** be traversed at some point in the testing procedure. This criterion requires partitioning the input space of a program into path domains and constructing test cases by picking some test data from each of these path domains to exercise every path in a program at least once. It is a similar procedure to the partitioning practice performed on the program requirements in black-box testing. The obvious difficulty with this approach, however, is the same as that for exhaustive testing (testing of all possible test cases) of a software program. Namely, it is that there would be far too many paths to be tested for full **path coverage** than a feasible testing procedure would allow. Programs containing loops may, for example, have an infinite number of

possible paths. Another complication arises due to the fact that a program may contain many paths to which no input data will flow. Such paths are called infeasible. They may result from a sequence of mutually exclusive predicates, such as the double if statements given in Figure 2.5.

```
.
.
// Notice that the condition for the second if statement is already decided due
// to the operations in the first if statement.
if (y < 0)
        { x = y - 1;}
else
        {x = y + 1;}
if (x > 0)
        {x = x + 3;}
```

Figure 2.5

The presence of infeasible paths reduces the total number of feasible paths in the program, and only the feasible paths need to be tested. On the other hand, there is no systematic way of determining feasible paths in an arbitrary program.

Missing paths represent another fundamental limitation to a testing strategy based on the structure of the program. A missing path typically occurs as a result of some condition, which the programmer forgot to include. This limitation, which is in effect a logic error, may affect only a single input value (such as a case in which special action is required for x = 0) there is no systematic way to test for the absence of such a predicate.

Another criterion is **condition coverage**, which requires each condition in a decision statement to take on all possible outcomes at least once. The problems of branch (decision) coverage also apply to this criterion. In the case of IF statements, condition coverage is sometimes better than branch coverage since it may cause every individual condition in a decision to be executed with both outcomes. Condition coverage criterion does not include branch coverage since test data exercising every condition value may not cover all decision outcomes.

Condition and branch coverage criteria may sometimes be applied together, but even this approach has a weakness in that the errors in logical expressions may go undetected. This is because some conditions may mask other conditions. A criterion that handles this problem is called a multiple condition criterion. In addition to the requirement of a decision/condition criterion, a multiple condition criterion requires construction of test cases to exercise all combinations of condition outcomes in every decision statement.

Finally, the phenomenon of **coincidental correctness** has to be considered as another factor decreasing the reliability of path coverage. As a simple example, we may consider the logically faulty expression y = x + 2, instead of the desired operation y = x * 2. A testing performed with x = 2 would not indicate that any error is present. Mathematically, it is possible to find an infinite number of functions, which will give the same results for any given finite set of input values. Therefore, unless we are willing to restrict the class of possible functions that will be computed by the program we are testing, coincidental correctness is a fundamental limitation to any testing strategy.

Despite the given limitations, white-box testing is a more powerful testing approach compared to black-box testing. There are two main reasons for such a preference. Firstly, white-box testing allows the tester to choose a relatively small and "sufficient" subset of test cases from the set of all possible test cases in a systematical way. As a result, this smaller set of test cases provide at least the same amount of confidence as the black-box-selected test cases, while requiring less resources and less effort for the execution and evaluation of program outputs. Secondly, the systematic selection of test cases allows the tester to focus on the more critical errors, or on certain highly critical sections of the program code, thus yielding a much higher efficiency compared to black-box testing.

2.4.2.3 Incremental and Non-Incremental Integration

In the previous chapters, a second classification for composite testing strategies, with respect to the coordination of the tested units in a program, was introduced. It was also mentioned that there were two such strategies, namely incremental and non-incremental integration testing strategies. The latter of these two strategies was further classified according to the direction of the level (up or down) increments in the merging of the subsystem with the next level unit. These

classifications were given as top-down testing and bottom-up testing. A third method called thread testing, which was both incremental in perspective of unit merging, as well as black-box in perspective of test case selection, was also mentioned.

2.4.2.4 Top-Down Testing

A **top-down** testing strategy starts with the top module in the program (the main program) and then proceeds to test modules at lower levels progressively. In order to stimulate the function of the modules subordinate to the one being tested, some dummy modules, called **stub modules**, are required.

Although there is no formal criterion for choosing an order among subordinate modules for testing, there are some guidelines to obtaining a good module sequence for testing. If there are critical modules (such as a module that is suspected to be error prone), these modules are added to the sequence as early as possible. Similarly, input-output modules are added to the sequence as early as possible.

Major advantages of top-down testing can be listed as:

- It eliminates separate system testing and integration.
- It allows one to see a preliminary version of the system.
- It serves as evidence that the overall design of the program is correct.
- It results in a primary and frequent testing in top-level interfaces.
- It allows the localization of errors to the new modules and interfaces being tested.
- It is usually a driving element in improvement of programmer morale.

Major disadvantages of a top-down testing strategy, on the other hand, are that stub modules are required and the representation of test data in stubs may be difficult until input-output modules have been tested. Test data for some modules may be difficult to create if data flow among modules is not organized into a directed acyclic graph, since stub modules cannot simulate the data flow, and observation and interpretation of test output may be difficult.

2.4.2.5 Bottom-up Testing

Bottom-up strategies start with modules at the lowest level (that do not call any other modules) in a program. They consist of module testing, followed by subsystem testing, and then by system integration testing. Driver modules are needed in order to simulate the function of a module superordinate to the one being tested. Modules at higher levels are tested after all of their subordinate modules at lower levels have been tested.

An advantage of bottom-up testing is that there is no difficulty in creating test data, since driver modules simulate all the functions of the superordinate modules, and thus the calling parameters, even if the data flow is not organized into a directed acyclic graph. If the critical modules are at the bottom of the calling sequence graph, a bottom-up strategy is advantageous. Secondly, a bottom-up testing can be used for time saving in large software design teams, where the testing of the low-level modules can be performed by separate teams.

Major disadvantages of bottom-up testing are that a preliminary version of the system does not exist until the last module is tested, and design and testing of system cannot overlap since one cannot start testing before the lowest level modules are designed. Furthermore, the need for driver modules to simulate input into individual modules is a factor, which increases complexity.

2.4.2.6 Thread Testing

The third method in the incremental integration testing classification is thread testing. It is an approach based on the "system verification diagram" (SVD) derived directly from the program requirements specification. In this sense, it is similar to the black-box approach of unit testing. Secondly, thread testing joins software test and construction. Therefore they do not occur separately or sequentially.

The order in which software is coded, tested, and synthesized is determined by the SVD that defines the test procedure. This SVD segments the system into demonstratable functions called threads. Following, the development of these threads is scheduled. Next, the modules associated with each thread are coded and tested in an order given by the schedule. The threads are synthesized into higher order sections called **builds**. Each build incrementally demonstrates a

significant capability of the system, which in turn culminates in a demonstration of the whole system.

2.4.2.7 Practical Comparison of Strategies

Appendix C.1 [Infotech, 1979] contains a program that looks up an inventory number in a merchandise table. Black-box testing would require data sets where inventory number is present in the table, is not present in the table, and is illegal. White-box testing would also detect these cases, and in addition, require testing the table representation. In the example, a linear search algorithm is used. Two additional test cases are needed with the inventory number set to the first element in the table. Another search algorithm would probably require data sets.

Black-box testing has the advantage of testing special cases that may have been overlooked or incorrectly implemented. White-box testing has the advantage of uncovering special cases that were included in the code but not in the specifications. It also has the advantage of concentrating on implementation problem areas. For best results in detecting errors, both methods should be combined.

Top-down, bottom-up, and non-incremental component testing are testing methodologies that are often proposed. The choice between these methods is often determined by the software development method that is imposed. Black-box and white-box testing can easily be used in conjunction with all three.

Black-box testing can be facilitated by the use of formal specifications. Structural testing can be assisted by a system that analyses the program structures, monitors the test runs, and then aids in the selection of test data to execute the unexercised sections of the program. This can be particularly useful in large systems containing many special cases to handle erroneous data or erroneous computations. The actual program paths to these selections of code may be short but it still may be tedious to analyze each one manually. A more efficient method is to allow the human tester to concentrate on the critical areas of the code and use an automated testing tool to assist in data selection for the more mundane portions of the code.

Finally, it should be noted that the majority of testing is structural (white-box) [Infotech, 1979].

2.4.3 Techniques for Dynamic Testing

Given the general structure of dynamic testing, there are various techniques used in the realization of actual executions of test data. Some of these techniques are used to create benchmarks to compare the incorrect outputs with (e.g., program mutation testing, and algebraic testing), while some other techniques are used to work with very specific sets of test cases (i.e., input space partitioning), or with specific types of software programs (i.e., compiler testing). Some of these different approaches in dynamic testing can be listed as:

- Program instrumentation
- Program mutation testing
- Input space partitioning (path analysis & domain testing)
- Functional program testing
- Random testing
- Algebraic program testing
- Grammar-based testing
- Data-flow guided testing
- Compiler testing
- Real-time testing

2.4.3.1 Program instrumentation

The basic idea of program instrumentation is to execute the program with different test cases, and to record the number of occurrence of certain error types. These recording processes are sometimes called probes, monitors, or software instruments. The nature of the recording process is dependent on the type of event performed as well as on the characteristics of the desired measurement data. Programs can be instrumented by the implementation of some recording process statements, which are designed not to interfere with the functioning of the program.

Various output statements, assertion statements, monitors, and history-collecting subroutines are among the used processes. The recorded events may range from the execution of certain statements to the violation of some condition statements.

It should also be noted that there are certain instrumenting compilers, which can automatically instrument a program during its compilation. The most commonly used automated instrumentation process is the history-collecting subroutine. A history of the behavior of a program instrumented as such will be automatically produced during its execution with different test cases, as well as other data such as ranges of variables, or CPU times for each routine. This historical information can then be used to prepare a new set of data better suited to find likely errors, and the tester may continue the iteration until a certain level of confidence is reached. In order to limit the execution history size of a program, the tester should select and concentrate on only selected regions of the program.

A well-known method of program instrumentation is the manual insertion of additional output statements into a program. Once a program is completely debugged, it is necessary to remove these statements. Since this process becomes tedious and time consuming in the case of large programs, the idea of using special recording processes in the program, called **assertions**, has been developed. The general format of the assertion statement found in some programming languages is as follows:

ASSERT boolean expression statement.

Semantically, if the boolean expression evaluates to FALSE, then associated statement is executed and the program is terminated. Some programming languages have built-in assertion statements, e.g. PLAIN and EUCLID. In other languages a pre-processor, called the **dynamic assertion processor** implements the assertions inserted by the user in the form of a comment. By recompiling the output of the pre-processor, these comments can be removed from the program.

2.4.3.2 Program mutation testing

Mutation testing is an error-based testing technique. In error-based testing, the goal is to form test cases that reveal the presence or absence of specific errors. Therefore, mutation testing is

simply a technique for the measurement of test data adequacy, and cannot be used to detect or locate any errors by itself. The benefit of this testing technique is realized by executing the created adequate set of test data in other programs (with other testing techniques). For the purpose of constructing such an adequate data set, mutation test employs mutants of the program under test. These mutants are basically the same program under test, with the addition of single specific error types that are chosen from a list of likely errors. The mutation (adding of errors) and testing operations are carried out in such a fashion that the resulting data set is able to distinguish a correct program from its mutated versions that contain these most likely errors. (This statement also implies that a program, which passes the mutation test, still has a chance of containing more unlikely errors, depending on the scope of the mutation test).

As a result of the executions of the selected test cases on the program and its mutants, a mutation score is formed. The mutation score is a simple representation of the fraction of mutant (nonequivalent) programs that were distinguished by the execution of the test case in question. A high score (fraction) will indicate high adequacy of the test data set.

Weak mutation testing and trace mutation testing are two variations of mutation testing. In weak mutation testing, mutant operators (automated programs which construct mutant programs from the parent) are applied to simple components of the program. The mutation score is then calculated based on a comparison of the outputs of the corresponding components of the mutant and the parent program. (A high mutation score does not guarantee that the passed errors in the test are absent in the whole program.)

In trace mutation testing, the creation of mutants is handled in the same way as in mutation testing. During the comparison of the execution results, however, trace mutation testing uses program traces rather than simple output comparisons.

Mutation testing is not intended for use by the novice programmer. On the contrary, program mutation testing has the underlying assumption that **experienced programmers** write programs which are either correct or are "almost" correct. The "almost" correct program is a mutant – i.e., it differs from the correct program by simple, well-understood errors. In order that it be feasible to perform mutation testing, the size of the set of mutant programs and the size of the mutation test data must be small.

Besides being a tool for determining adequate test data, program mutation also provides the type of information that managers need to monitor software development and personnel performance. By using an automated program mutation system with report generation capabilities, managers may extract the following information during software development:

- Mutant failure percentages for each module indicating how close the software is to being acceptable.
- Who is responsible for classifying which mutants are equivalent.
- Which mutants have yet to fail.

2.4.3.3 Input space partitioning (path analysis & domain testing)

Input space partitioning starts with the path analysis and testing of the program. A path is defined as some possible flow of control inside a program. In path analysis, the input space of a program is partitioned into path domains. This forms subsets of the total domain of possible program inputs, each of which causes the execution of a different path. Next, the paths are tested separately in effect, by testing the program with selected input data from corresponding subsets of the input domain, with a total number and combination of test cases so that each path in the program is executed at least once. In practice, however, a selection usually has to be made from the set of subsets, since a program may contain an infinite number of paths. The errors that can be found by path analysis and testing are computation, path selection, and missing path errors.

Domain testing detects path selection errors by selecting test data near or on the boundary of a path domain. Path selection errors occur when the branching predicates are incorrect. Several assumptions used with this technique are that an oracle exists, which determines the correctness of the result produces by the execution of the program with some test case, that coincidental correctness cannot occur, and that the input space (domain) is continuous.

In partition analysis, which forms the static part of input space partitioning, the specification of a program written in formal specification language is partitioned into subspecifications, along with the input domain of the program. Next, these two partitions (of the specifications and of the domain) are joined in order to get a set of input data, for which a subspecification is available and a specific path is known. These elements are called "procedure subdomains" [Infotech, 1979].

Partition analysis then uses symbolic testing to check the equivalence of a subspecification and a subdomain, and that of their corresponding computations. The occurrence of an inequality in the comparison will imply the existence of an error in the path in question.

Input space partitioning allows the tester to perform a variation of unit testing, in which the units are reduced form functional components to paths inside the software code. In this way, the effort requirements for the localization of determined errors are reduced greatly, since all paths are analyzed separately. On the other hand, with the advanced programming languages and the high complexity of program structures today, a program may contain an infinite number of paths, and determining the separate domains may not be possible. Consequently input space partitioning may not be able to detect all of the path selection, computation, or missing path errors. This technique is generally found most feasible for the testing of medium size applications. Large software applications cause various difficulties in the partitioning of the domain, as mentioned, and easier and less time consuming testing techniques exist for smaller software programs. Testing techniques of the latter type usually employ black-box testing technologies, and therefore do not expose test engineers to the program code. Two different techniques of such nature are explained in this section, namely functional program testing, and random testing.

2.4.3.4 Functional program testing

Functional program testing is a design-based approach to program testing, in which the design of a program is viewed as an abstract description of its specifications. This description, in turn, is used to identify the abstract functions of the tested program, and to construct the functional test data.

In the first step of functional testing, the functional abstraction methods used to design the program are used as a guideline in decomposing the program into functional units. Function abstraction is a program design strategy, which views the program as a hierarchy of abstract functions. A function at one level of this hierarchy is defined from the functions at a lower level. In the second step, the necessary test data is generated to test these functional units independently.

Functional testing is similar to the black-box strategy of testing, but is performed as a complete testing technique rather than an approach in test case selection. It is particularly suitable for novice programmers working in the testing of relatively simple programs, and only limited reliability is offered, similar to black-box testing. As a result, functional testing is generally limited to simple programs with relatively inexperienced testing teams, in order to provide a rapid reliability assurance function with low resource requirements.

2.4.3.5 Random testing

Random testing is a black-box testing strategy. It involves the random selection of a set of test cases from a set of all possible values. This technique has been proven to be a cost-effective testing method for many programs including real-time software. This makes it especially suitable for the testing of software with a high number of possible test cases, such as for the testing of real-time software. It is, on the other hand, one of the least reliable testing techniques. This is because of the fact that the selection of test cases does not employ any method to concentrate on type of error, or distribute the probability of error occurrence evenly between likely errors. In this sense, it will require a lot more effort and time to provide a similar level of confidence, as other testing techniques will.

2.4.3.6 Algebraic program testing

Algebraic program testing is based on the concept of program equivalence. This approach requires the tested program and the benchmark program to lie in some restricted programming language class (such as having only integer type variables, or no branch statements) for which testing on a small set of data is sufficient to prove program equivalence. This requirement is because such equivalence is impossible to determine with more powerful programming languages. The purpose of this technique is to show the correctness of a program under test, by showing its equivalence to a hypothetical correct version of that program. The assumptions in this approach should be viewed as somewhat close to that in mutation testing, since they are both concerned with benchmark correct programs that are equivalent to others if they are also correct.

The testing is carried out in numerous steps. First, a class of equivalent correct programs, and a set of test cases capable of proving equivalence are selected for the testing of a given software program. Next, all selected programs, along with the tested program are executed with the selected set of test data. Finally, outputs of the programs are compared for computational equivalence, which is then used to prove the equivalence of the programs.

One of the problems encountered in this approach is the fact that algebraic testing is limited only to programs with a certain low level of capabilities. Secondly, the existence of a correct benchmark program is an assumption, which decreases the reliability of the technique. Lastly, the generation of a set of test cases, which will exercise the program for all possible errors, is another problem of software testing that has to be solved on its own. As a result, this situation requires another assumption that the selected set of test cases will actually exercise the program for all errors. This assumption, of course, is very unlikely to become a reality, which is also the reason for the existence of many testing methods that are dedicated only to the construction of such a set of test cases for a software program.

Algebraic testing is an extremely valuable approach for designing improved versions, or versions for different platforms, of a given software program. In such cases, the testing function can be performed as if the program under development was the next cycle of the software development of the similar product. This will allow the development team to have a model throughout the software lifecycle, and the testing team to have a benchmark to test against. In cases where such a model can be found, algebraic testing is almost always desirable due to its rapid testing capabilities as well as low resource requirements.

2.4.3.7 Grammar-based testing

Grammar-based testing is used for the testing some software systems, such as airline reservation systems, which have formal specifications that can be modeled by *a finite-state automaton* (FSA). For the testing of such a system, a regular grammar for the language accepted by the FSA is constructed, and the testing is based on this grammar to generate inputs and outputs for the system.

The components of a grammar-based testing system are the Requirements Language Processor (RLP), the Test Plan Generator (TPG), and the Automatic Test Executer (ATE). Input to RLP is a formal specification of the system under test. The output of the RLP is a state-transition matrix (STM) representation of the FSA. Since the RLP may assure that there are no inconsistencies in the requirements, it is assumed that the STM represents a deterministic FSA. The reachability of each state is assured by computing the transitive closure of the STM [DeMillo et al, 1987]. Using a result from automa theory, a regular grammar for the language accepted by a FSA can be constructed [DeMillo et al, 1987]. The regular grammar is manually augmented to take into account the relevant system information for each state transition and to indicate the observable outputs of the FSA, e.g., "the observable outputs from the finite-state machine must be terminal symbols" in the grammar [DeMillo et al, 1987].

Input to the TPG is an augmented FSA. The TPG then outputs a set of test scripts. Each script is a sequence of executable inputs and corresponding outputs expected from the system under consideration. The ATE executes each test script and reports whether the system responds in a desired manner or not [DeMillo et al, 1987].

An important consideration associated with these strategies is that it is necessary to employ a criterion for choosing the productions in the grammar to prevent loops. One such criterion is to limit the number of times a production is used.

Grammar-based testing strategies have been applied for testing nested conditional statements in Ada, testing a sort program, and a testing reformatter [DeMillo et al, 1987].

Another application of grammar-based test data generation strategies involves generating test programs for input to a compile under test [DeMillo et al, 1987].

2.4.3.8 Data-flow guided testing

In this technique, the basic assumption is that a software program establishes meaningful relationships among program variables. Given these relationships, data-flow analysis is a technique for optimizing the compiler to analyze certain structural properties of the program and to extract structural information about the program. Next, a testing strategy is defined for

exercising different data transformation paths for all or some variables in the program. Finally, obtained control flow information is used to define a set of corresponding paths to be executed.

2.4.3.9 Compiler testing

Compiler testing is a testing technique that is used to verify that a compiler runs correctly, and accepts programs only if they are correct. For this purpose, grammar-based testing strategies are employed. In these strategies, the compiler is exercised by a set of compilable programs, generated automatically by a test generator, which is driven by a description of the source language. The output, which is the set of programs to be compiled by the compiler, is a group of programs covering all syntactical units of the source language. In addition, users may also manually generate various incorrect programs, the definition for which can be provided by the test generator.

2.4.3.10 Real-time testing

The drastic level of changes in the computer industry through the last twenty years has brought most software programs to the real-time level. Made possible by advanced and powerful programming languages, and dramatically faster CPU's, and high graphics capabilities, many software programs that are used by today's consumers are interactive, and therefore function with external devices, objects, or real-time inputs.

Real-time software is defined as software which interacts with external devices or objects, and whose actions control activities that occur in an ongoing process. Since such capabilities require a high number of lines of code, as well as a high number of modules, the testing of modern real-time systems is more difficult compared to traditional programs. A second problem complicating the testing of real-time software is the infinite number of input variables (test data), since time itself is a new parameter in test data, which means that same data input at different times may create totally different actions in a correct real-time program. This problem also makes random testing, which was previously stated as being a less reliable technique for software testing due to its lack of a systematic data selection procedure, the preferred solution for the testing of real-time software, due to its cost effectiveness.

There are two types of computers involved in the testing of real-time software, namely the host and the target computer. The host computer is used to construct programs for the target computer, and is usually commercially available, whereas the target computer is the real-time computer, which controls the activities of an ongoing process. As a result of this involvement of two different computer types with different functions, real-time software testing is performed in two different phases. The first phase is the testing of the host computer, and the second is the testing of the target computer.

Most of the techniques used for the first testing type, the host computer testing, are the same as those used for non real-time applications. Its goal is to reveal errors in the modules of the software. The complete testing of an integrated system is rarely done on a host, and requires running an environment simulator and controlling on-going processes appropriately.

In target computer testing, module testing is conducted first. This is followed by system integration testing, which also sometimes uses an environment simulator to run the target computer. Finally, the testing is completed by running full system testing, often with real data instead of the simulator.

Testing Technique Developed for NASA Space Shuttle Program [DeMillo et al, 1987] - The following technique was used by IBM during the verification of the NASA Space Shuttle Project. The object of the Flight software Verification Project was "an independent test, analysis and evaluation of the Space Shuttle Flight Program to ensure conformity to specifications and satisfaction of user needs".

During the development cycle the modules were tested individually. After the completion of the development cycle, the software was released to a verification organization which was formed independently from the software development organization. Members of the verification organization participated in the design and code reviews conducted by the software development organization. During this phase, testing checklists were developed to design test cases.

The development and verification of the flight software were performed in an IBM-developed simulation testbed, which provided "a sophisticated array of user capabilities" e.g., real-time flight software patching. The verification project involved two distinct and complementary

phases: detailed and performance testing (phases in testing of real-time software, performance testing is explained in detail in Testing for distributed software Applications).

For the detailed testing, the requirements specifications were divided into functional units. An individual analyst was responsible for developing a test plan to exercise each defined requirements path and a "sufficient" number of path combinations for each such unit. Additionally, "a level of requirements validation and user need testing" which was based on "the analyst's observations of the flight software response and knowledge of the intent of the requirements" was planned.

Detailed testing analysis, which followed the acceptance of the test plan, involved explicit testing of all requirements defined in the requirements specifications, explicit testing of I/O compatibility of the modules, and the execution of each detailed test case with the entire operating software. Each test case, which revealed the existence of problems was executed again on the corrected software.

Performance testing involved testing flight software under "normal operation, 'reasonable' failures, and severely degraded conditions". Test cases were constructed in such a way that the components of software were stressed selectively. The criteria for performance success were "to operate successfully across the range of the customer's flight performance envelope" and "to satisfy a set of software design constraints" e.g., CPU utilization.

2.4.4 Test Data Generation

Given a program, and a segment within that program, it is desired to generate a test data that will cause the program to execute the selected segment. In practice, however, there are various constraints, which make the realization of the above statement more difficult.

In this sense, the important factors affecting test case selection can be listed as follows:

- The primary concern is the **level of ease at which test cases can be constructed**. It is known that some test cases may be extremely complicated in nature, especially if they are constructed to form a consistent set of data for exercising a complex software program, such

as one for the management of a system with many parameters. The difficulty and cost of preparing such test cases can usually be significant. As a result, test cases that are easier to construct should be given priority, provided all alternatives are able to exercise the same segments of the software program.

- A second important concern is the **ease of checking program output for correctness**. In cases similar to software programs requiring complex test data, some programs may output results that are difficult to obtain through other means. For instance, the correct answer for some numerical algorithms may not be easily determined without using a computer. Such consequences will require a lot of effort to calculate the expected results, alone. As a result, in the selection of test cases for such programs, priority should be given to test data for which the results are either known, or are easily computed.
- A major concern in all software development tasks is that of cost. Likewise, **the cost of executing selected test cases** is an important issue in software testing. Required costs as well as other resources for execution of test cases must be kept within given limits. This can be achieved either by skipping some less important test cases, which may occur in practice, or by preferring simpler test cases for the execution of the same program segment.
- Since testing resources are always limited, importance should be given to the **types of expected errors** as a result of the execution of selected test cases. In order to achieve the highest efficiency with these limited resources, activities must be focused on checking certain critical kinds of errors.

According to the selected test data, program testing is roughly divided into the following parts:

- **Exhaustive testing** – This type of test data selection aims to test every feasible input value for the software program. The result should have all possible values and situations tested, and the outputs examined for correctness. Due to this approach, exhaustive testing is almost always impracticable, and prohibited by cost and schedule. If, for instance, we consider a loop containing nine paths, which is executed ten times, we get 9^{10} possible execution sequences, which is clearly impossible to complete.
- **Selective testing** – This type of testing runs the program with test data prepared either by the programmer or by the test engineer. It is an improved, and therefore a more practical form of exhaustive testing. The aim is to try to find all errors in the program with the selected test data, rather than check every possible output. Selective testing is subdivided into two groups:

- **Statistical testing** – In this subgroup, test data generation is realized according to estimated distributions of the input data. The significance of this data may not be great due to the assumed distribution, instead of the actual.
- **Structural testing** (manual/automatic) – In this type, expected results are extracted directly from program specifications and requirements. If the dynamic analysis finds a path that has not yet been executed, a path predicate for the path is calculated by performing symbolic execution. Structural testing based on path analysis, is the most common test data-generation technique.

Given the major considerations in test case selection, and the general classification of program testing with respect to the nature of selected test cases, some of the important problems related to making the selection of test data are the following:

- Test path selection
- Symbolic execution of dimensioned variables
- Solving general inequalities of real and integer variables
- Measuring program testedness

The common techniques used for test data generation are:

- **Pathwise test data generation** – The principle of this data generation technique is to choose progress paths such that the desired level of test coverage will be accomplished. Basically, a program path is selected, and a correct set of results for that path is formed, commonly through symbolic execution. Path selection can be performed as *manual dynamic path selection* (manual selection by the user of the next statement whenever a decision point is encountered), *manual static path selection* (path is specified completely before commencement of analysis), *automatic static selection* (automatic selection prior to symbolic execution), *and automatic total selection*. The two major problem groups encountered with this type are those associated with the complexity of the path selection process, and those associated with pathwise test data generation (ambiguity of symbolic expressions derived through symbolic execution) [Infotech, 1979].

As mentioned earlier, this behavior is preferably modeled by the Pathwise Data Generation technique for distributed software applications due to their requirements of interactive, multiple-path test scenarios.

- **Random test data generation** – This technique is used as an alternative to pathwise data selection, where data is simply selected from the group of all valid test cases. Such selection does not require knowledge of the source code or the architecture of the program, and is therefore used in black box testing strategies. (See Random Testing – Chapter 2)
- **Test data generation by program mutation** – Program mutation, as previously discussed, is an effective way of test data generation for experienced programmers. It is based on an assumption that programmers will have an idea about the errors that are likely to occur in the software program under test. (See Program Mutation Testing – Chapter 2)

2.5 Standards

Standards are documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines, or definitions of characteristics, in order to ensure that materials, products, processes and services are fit for their purpose.

For example, the format of the credit cards, phone cards, and "smart" cards that have become commonplace is derived from an ISO International Standard [ISA homepage, 1999]. Adhering to the standard, which defines such features as an optimal thickness (0,76 mm), means that the cards can be used worldwide. International Standards thus contribute to making life simpler, and to increasing the reliability and effectiveness of the goods and services we use.

Similarly, the use of standards throughout all stages of a software development task, including testing of the software, provides a level of quality to the software, that can be recognized globally or at least in all related organizations. A standard for testing ideally defines an integrated approach to **systematic** and **documented** testing. The systematic condition is important due to the fact that only a systematic testing approach can show a consistent level of reliability, as opposed to random approaches. Secondly, the documented condition is important in

order to provide a universal acceptance to the work performed on the software. Other members of the software development team, as well as future members and third parties must have access to an easily interpretable form of the testing work so that the quality of the software is certified. For this certification to be satisfactory at all times, most testing standards describe testing processes composed of a hierarchy of phases, activities, and tasks, and define a minimum set of tasks for each activity. Furthermore, the performance of each activity is often a requirement.

Today, there are various globally accepted standard-issuing organizations, which function to create standards for quality assurance in their area of service. These standards include the related standards for testing in the case of almost any organization dealing with critical software programs. In addition to these organizations, there are also certain smaller and highly specialized standard-issuing organizations, which function as divisions of the larger ones or are accredited by one or more global organizations. One such example is the American Society for Quality Control (ASQC), which functions as a secretariat for the American National Standards Institute (ANSI) [ANSI homepage, 1999]. See Appendix B.1 for a more thorough list of other related standard-issuing organizations.

As in many other fields of the industry, the need for globally accepted standards and standard-issuing organizations for effective software development is extremely high. In 1994, the Standish Group examined 8380 Industry Software Project Developments to identify the scope of software project failures. They found that 53% were "challenged" (cost overruns, late delivery and/or missing capabilities); 31% were canceled during development; and 16% were successfully completed (on time, within budget, full capability) [IV&V index page, 1998]. The biggest causes of non-success were identified as incomplete/changing requirements and lack of user involvement. The purpose of software development standards in that sense is the mitigation of risk in the development of quality software that meets the needs of a project.

Major organizations and related standards concerning the testing task of software development are as follows:

The Institute of Electrical and Electronics Engineers, Inc. (IEEE) Standards [IEEE homepage, 1999] - The IEEE is an organization that was founded to promote the engineering process of creating, developing, integrating, sharing, and applying knowledge about electro- and information technologies and sciences for the benefit of humanity and the profession. The IEEE

is a not-for-profit association and has more than 330,000 individual members in 150 countries. Through its technical publishing, conferences and consensus-based standards, the IEEE

- Produces 30% of the world's published literature in electrical engineering, computers and control technology
- Holds more than 300 major annual conferences
- Has more than 800 active standards with 700 more under development.

The IEEE Standards Association (IEEE-SA) is the 21st century organization under which all IEEE Standards Activities and programs are carried out. It was formed to provide a major entity that would offer increased responsiveness to the standards interests of IEEE societies and their representative industries.

The NASA Software Independent Verification & Validation Facility Standards [IV&V index page, 1998] – Services of this division of NASA assist customers with developing software within initial cost estimates and on schedule, while achieving and maintaining high quality. The IV&V Facility primarily develops and maintains a focused effort to improve safety and quality in NASA programs and missions through effective applications of Software IV&V disciplines, tools, and techniques. Additionally, the facility provides balanced, tailored, and appropriate IV&V technical analyses for NASA programs, industry, and other Government agencies, by applying a variety of techniques, often supported with automated software tools, to evaluate the correctness and quality of critical and complex software throughout the software development life cycle.

Software IV&V and the related standards have been demonstrated to be effective on large, complex software systems to increase the probability of software being delivered that meets requirements and is safe, within cost, and within schedule. When performed in parallel with the development life cycle, software IV&V technique has been a proven aid in the early detection and identification of risk elements.

Department of Defense (DoD) Test and Evaluation (T&E) Standards for Electronic Warfare Systems [DefenseLINK homepage, 1999] – The Department of Defense Single Stock Point was created to centralize the control, distribution, and access to the extensive collection of military specifications, standards, and related standardization documents either prepared by or

adopted by the DoD for the testing and evaluation of electronic warfare systems. The DODSSP mission and responsibility was assumed by the Defense Automated Printing Service (DAPS) Philadelphia Office, in October 1990.

The responsibilities of the DODSSP include electronic document storage, indexing, cataloging, maintenance, publish-on-demand, distribution, and sale of military specifications, standards, and related standardization documents and publications comprising the DODSSP Collection. The DODSSP also maintains the Acquisition Streamlining and Standardization Information System (ASSIST) management/research database. The document categories in the DODSSP collection are:

- Military / Performance / Detail Specifications
- Military Standards
- DoD-adopted Non-Government / Industry Specifications and Standards
- Federal Specifications and Standards
- Military Handbooks
- Qualified Products / Manufacturer's Lists (QPLs/QMLs)
- USAF / USN Aeronautical Standards / Design Standards
- USAF Specifications Bulletins

Although the DODSSP Collection contains over 50,000 line items, not all documents specified in Government procurements are included (e.g.: engineering drawings, some Departmental documents, and the majority of all Non-Government / Industry Standards). A list of military departments utilizing DoD T&E Standards is given in Appendix B.2.

Other major sources of international or US-only standard sources for the area of software development may be listed as follows:

2.5.1 International Standardization Organizations [Online Standards Resources, 1997]

- ISO (International Organization for Standardization)
- ITU (International Telecommunication Union)
- IEC (International Electrotechnical Commission)

- IFIP (International Federation for Information Processing)

2.5.2 U.S. Headquartered Standards Organizations [Online Standards Resources, 1997]

- ANSI (American National standards Institute)
- NIST (National Institute of Standards and Technology)
- NCITS (National Committee for Information Technology Standards)

See Appendix B.1 for the details regarding the above organizations, as well as a more thorough list of other related standard-issuing organizations.

CHAPTER 3

Testing for Distributed Software Applications

3.1 Introduction to DSA

The rapid spread of network-based development platforms has made it possible for application programmers and software developers to fully utilize the processing power of multi-processor installations from any place in the world and use this to provide fault tolerance and reliability whenever safety-critical application problems are to be solved. There is an emerging community of distributed software developers and an even greater community of clients, including banks, telecommunication companies, transport organizations, and central and local administration. Maintaining high quality of such products for these clients is a greater challenge than before, since development and testing of distributed software applications (DSA) requires entirely new methodologies and tools, in additional to the conventional.

An application may be defined as an advanced piece of software offering custom services strictly determined by a particular class of user requirements. Any general-purpose software, such as an operating system, a database system, a language compiler, or a debugging tool are merely elements of the skeleton for the considered software.

The latest challenge in the development of DSA is the need for engineers and domain experts from substantially different disciplines to collaborate to complete one application. This requires a new kind of tool, which supports cooperative work, and leads to entirely new classes of applications, called *cooperative* or *groupware* applications. A groupware application involves design, strategic planning, decision making, problem solving or software inspection. It can be classified as:

1. A system-oriented application, such as database systems, editors, data communication systems, etc.
2. A user-oriented application, addressing a strictly determined group of users or a single user.

Three main veiwpoints can be considered in order to distinguish between distributed software applications (DSA) and conventional applications.

3.1.1 Computing environments

Computing environments that form the means for distributed computing have a different architecture compared to conventional systems. Four fundamental types of computer systems, with respect to the type of processing, can be distinguished, namely ***centralized, parallel, networked, and distributed*** systems. Each type has its own specific organization standards and management strategies.

A ***centralized system*** involves a main computer playing the role of a central point, where it is usually accessed with terminals. These systems are rarely used in today's computing jobs. In a directly opposite approach, ***parallel systems*** require concurrent action of two or more processors and memory units, which are coupled together. A system may consist of such processing elements connected over an area network in order to form a ***networked system***.

A ***distributed system***, on the other hand, is a collection of autonomous computers linked by a computer network and equipped with distributed system software. This distributed system software enables computers to coordinate their activities and to share the system's software, hardware, and data. A distributed system contains the following:

- Multiple processing elements that can run independently. (Each processing element must contain at least one CPU, and local memory)
- Interconnection hardware that allows processors to run in parallel and to communicate and synchronize with each other.
- Fail independent processing elements, where recovery of a single processing element does not require the recovery of the others, and similarly a failure of one does not imply a failure of the others.
- Shared state implying that each processing element possesses some partial information about the global system state.

3.1.2 User domain applications

User domain applications are a subgroup of DSA, which are targeted for specific computing environments and are supported by various programming platforms. In this perspective, distributed computing is one of the most efficient means for speeding up computations, due to the combined processing power of many computers with different computational capabilities.

A distributed program can either be sequential or parallel. In the sequential case, modules of a program are distributed among different computers and are executed one after another by transferring control and data to subsequent modules. In a parallel-distributed program, the modules can execute simultaneously and several computers can be active at the same time. This is the desired domain arrangement for utilizing the full potential of multiple processing elements.

3.1.3 Programming Platforms

A local computer network consisting of independent workstations and servers connected through a fast transmission medium is usually sufficient for the special needs of distributed software application development. In order to support the development cycle of DSA programs, many new platforms with added capabilities have been developed in addition to the existing set of platforms, such as PVM, HeNCE, Linda, HP Task Broker, and Amoeba.

n view of the above considerations, typical DSA programs may be of the following types:

- Scientific computing applications – Utilize parallelization capabilities for added speed.
- Network applications – Create data sharing server, and application running clients for allowing lower congestion due to multiple users.
- Real-time applications – Use sequential or simultaneous operations for real-time simulations.
- Distributed database applications – Utilize combined memory for higher capacity, flexibility (modularity), reliability, and performance (simultaneous query execution).
- Multi-media applications – Extend the geographical scope of user interaction.
- Critical applications – Applications where the loss of service may cause serious consequences, such as loss of life, economic losses, or ecological catastrophes. Utilize distribution of both hardware and software to increase dependability.

3.2 Development of Test Scenarios

3.2.1 General Overview

During the analysis of test case construction for conventional software programs up to this point, testing strategies have been examined as specific combinations of static and dynamic analysis of the software code. The purpose for employing these strategies was given as that of checking various hypotheses that had been formed on the program code. As mentioned earlier, this behavior is preferably modeled by the Pathwise Data Generation technique for distributed software applications due to their requirements of *interactive*, *multiple-path* test scenarios. This type of analysis, however, allows the tester to realize and evaluate only sequences of concrete events such as data and control flow during communication events, message buffer changes, object status changes, etc. It does not expose him/her to the functional aspects of the program under test. As a result, the proposed testing methods up to this point can only offer testing limited to a certain level of *structural abstraction*, which emphasize on the syntactic aspects of program behavior as they rely mostly on the program text. In order to effectively test DSA programs, however, the tester should also be able to emphasize the semantic aspects of test strategies. Therefore, on top of structural abstraction, a more general form of test scenarios is required,

which completely defines certain functional properties of DSA programs. Throughout this section, four such strategies are described, namely:

- *Performance Testing* – concerning efficiency of DSA programs running in a concrete computing environment.
- *Fault-Injection Testing* – attempting dependability evaluation of DSA programs.
- *Assignment Testing* – aimed at time dependent errors.
- *Self-Checking Testing* – involving applications with various built-in self-checking mechanisms.

It should be noted that these strategies are described with the general consideration of DSA, since these applications generally have much higher functional requirements compared to conventional software programs. On the other hand, functional considerations are also of high importance for all remaining programs, given the role of computing in the daily life at present. As a result, the testing strategies described throughout this section are almost always applicable and often required for all types of software programs.

3.2.2 Testing Strategies for DSA Software

3.2.2.1 Performance Testing

In general, the objective of performance testing is to find causes of poor performance in a distributed computing environment running a given DSA program. There may be many reasons for poor performance: insufficient size of communication buffers, wrong values of time-outs, conflicts of interconnection networks, race conditions, deadlocks, etc. Some of these reasons may imply typical software errors. Therefore, *performance testing* should be clearly distinguished from *software testing by performance analysis*, which concentrates on detection of software anomalities that cause poor system performance.

There are three broad classes of performance evaluation approaches, based on *measurement*, *simulation*, and *analytic models*. Of these classes, measurement, which depends on direct

information gathering through program execution, is the most flexible and accurate and gives the user a great deal of knowledge about the program and its computing environment. There are two main drawbacks to this approach. The first is the required instrumentation of the program, which is performed by implementing various instruments inside the code that output the desired data during run-time. It is known that instrumentation may cause problems in achieving the required accuracy in certain program components (i.e., program clocks) [Krawczyk et al, 1998]. The second drawback is the intrusiveness in instrumentation, which may result in a change of measured characteristics.

Simulation is the second performance testing approach, which aims to speed up interactive path interpretation for DSA software, by eliminating the need for individual branch selection when determining the next possible target state in a testing operation. Instead, the program model can be run on just one machine and timing of processes and data can then be simulated by random delays of all other branches during their progress through the test scenario. However, since simulation always refers to the model, it must be carefully scrutinized to ensure an acceptable level of correctness.

The third approach, analytic modeling, is based on building up a mathematical description of performance. It is a powerful and flexible testing approach suitable for the design stage of the software development cycle, since it shows how performance may vary with regard to individual parameters, allowing designers to identify regions of high and low performance. As a result, tested programs can be optimized for certain parameters during their design stages. Even if a precise prediction of system behavior may not possible through this approach in certain cases, they can help in determining performance trends or comparing the performance of different applications.

A broad spectrum of data collecting mechanisms can be used to obtain information about program performance, such as:

- *Profiles* – showing the amount of time spent in different program components, as represented by histograms, for example.
- *Counters* – constituting storage locations that may be incremented each time a specified event occurs, such as an interval timer determining the length of time spent executing a particular piece of the program code.

- *Traces* – generated in the form of log files containing time-stamped recorded events, including communication actions.

3.2.2.2 Fault-Injection Testing

In critical applications, which are commonly recognized as systems where the loss of service may cause serious consequences, such as flight control systems, railway signaling systems, nuclear power plant or health carrying systems, dependability requirements have primary importance. Fault-injection techniques have been developed as a result of such strict requirements on dependability, in order to validate critical applications and their underlying computing environments. They depend on the insertion of faults into both the hardware and the software components of a target system, or just its simulation model (providing it accurately reflects the components' behavior). The system under test is evaluated with a representative workload during each experiment, in order to obtain realistic responses, and the effects of each inserted fault are adequately monitored and recorded.

One of the most common uses of fault-injection testing is the functional testing of prototypes during the software development cycle. When a product (system) is completed and ready for deployment, fault-injection can be used to observe the behavior of its code in the presence of a given set of faults. In this manner, information about the quality of system design and the level of dependability can be achieved. In the case of DSA programs, the primary reason for consideration of fault-injection testing is the complexity and diversity of these programs, and the existence of dormant bugs resulting from this complexity, which do not cause any observable failures until suddenly triggered on occurrence of extreme conditions.

There are two basic areas considered in fault-injection testing, both of which are applied depending on the relevant dependability standards. According to the areas of fault-injection, the two types of fault-injection testing are:

- *Code injection*, dealing with the source code or the object code directly, causing code or data structure corruption.
- *State injection*, achieved through altering the state behavior of the running software code. Modifications are made in the operating system, processors, storages, dynamic states of

running processes, messages, etc., as well as in specific service, administration, and maintenance commands.

3.2.2.3 Self-Checking Testing

The most spectacular anomalies in software programs can be discovered quite early during the development cycle. More malicious bugs are usually detected later, after a refined series of experiments. However, a small number of *residual errors* may survive even the most sophisticated tests and remain dormant throughout the rest of the product lifetime. Consequently, there exists a need for fault-tolerance, especially for critical applications mentioned in conjunction with fault-injection testing. Self-checking testing aims to determine the most tolerant way possible for the delivery of desired services from the software program under test. For this purpose, different versions of the software called *variants* are created, and the most fault-tolerant is selected. These application variants differ from replications of the same application because they deliver an identical service via independent design and implementation by different programmers, but using different methods and tools, and even programming languages than the original application.

Self-checking testing employs one of three different detection mechanisms implemented on top of different variations of the application or the computing system under test. These detection mechanisms are as follows:

- In *N-version programming* (NVP), different versions of the application program are executed on different hosts in parallel, and results this obtained are voted on.
- In *Recovery blocks* (RB), all versions are executed serially and special tests are used to check if the obtained results are acceptable. An alternative test version is executed only if the output of the currently run version fails the test.
- In *N-self-checking-programming* (NSCP), each component variant is equipped with tests to check the delivered outcomes, or pairs of component variants are specifically associated with a comparison algorithm to check on their behavior. Upon failure of the acting variant, service delivery is switched to another self-checking component, called a *spare*.

3.3 Problems & Concerns for Testing in Distributed Teams

The concept of software development in a distributed environment has come a long way, from its initially infeasible form in the past, due to low technology and high prices of hardware, software, and connections available to support quality communication between geographically isolated teams throughout the software development process. It was commonly accepted that success in distributed teams was more difficult compared to conventional face-to-face teams. On the other hand, the use of distributed teams today, in software development and other design related sectors, has become a common practice due to the savings in cost and time made possible by integrating geographically distributed members into a team without actually flying them over to meet each other. This is made possible through today's improved technology, bringing advanced connection capabilities, purpose-built computer software for distributed design work, and improved means of communication (i.e., e-mail, Internet chat programs).

Despite a long list of advantages offered by using distributed teams, there are various problems and other concerns regarding this issue. Despite the fact that distributed design and development is accepted as a common an effective approach at present, some other perspectives still claim that the technology is not fully developed to allow a feasible use of such approaches. Many problems related to lack of technology, to distributed team management still being a relatively new concept under development, or to ethnic compatibility issues still exist. Furthermore, these problems also reflect to individual functions and tasks of software development, at times forming unique problems peculiar to every function. Software testing is one such function, having various characteristic problems in the case of a distributed effort. This section examines a list of these problems encountered in distributed testing performances, and the Comments for Testing in Distributed Environments section offers solutions to these problems where possible.

The list of problems in distributed software testing and their possible causes are as follows:

1. In the testing of a major portion of today's large-scale software applications, testing starts in the form of static analysis at the commencement of program coding. It commonly progresses with dynamic testing of units, which are then joint and integration tested. As a result of this

structure, the importance as well as the difficulty of software testing depends on the performances of three other software development functions. The most obvious of these functions is the **programming** function, involving the writing of the software code. The other two functions, **requirements analysis** and **software design**, are less obvious but more important in terms of the base they form for performance of testing. The difficulty of testing and programming both depend on the quality and success of these two previous functions. If there are undiscovered problems, delays, or missing parts in these tasks, they are transferred to programming and in effect to testing functions. Furthermore, the difficulty of testing also depends indirectly on the success of the **quality assurance** function, which is responsible for ensuring satisfactory performances of the aforementioned functions. Some specific examples regarding the presented state, are as follows:

- In spite of being a rare application, in the case of software programs that are dynamic-tested with only a black-box approach and that rely on the programming function for static analysis of the software code, a **poorly written code** will make testing extremely long and costly. In such projects where static testing and white-box testing are not emphasized, it has to be accepted that the code is free of simple errors such as syntax errors, and typos, as opposed to rational errors such as flow logic errors. If the programming function fails to accomplish this requirement, these simple errors combine to form an extremely difficult task of testing. Furthermore, since debugging usually has to be carried out for simple errors because they do not make their location obvious in the error, more effort and time is required in these cases. In addition, these extra requirements may produce delays in the overall schedule, which will seem to be resulting from the testing task.
- In cases of black-box testing, the creation of test scenarios depends primarily on use cases presented in the requirements document. As a result, any **lack of well-kept documentation**, or **deficiency in the requirements document** will cause difficulty in the generation of test cases. In order to prevent the generation of redundant test cases, and in order to provide a complete coverage of software functionality to assure satisfactory reliability, a clear understanding of the software development process, as well as accurate definitions for the expectations and requirements from the program must be provided.
- In a similar fashion to the case presented above, **lack of** a **successful design** of the software program will cause the program to accomplish the required task through a more error-prone path, or with different capabilities than the required. As a result, the testers

often have to report on the compliance of the software with the requirements document when preparing testing reports. Furthermore, in the case of more error-prone paths, added testing effort is required in the form of extra use-cases, or performances of **Self-Checking Testing** in the case of critical software applications.

- As previously stated, it is the duty of the quality assurance function to ensure that all functions of the software development process are performed to their best. For a similar reason, and to provide a wider acceptance for the developed program, all software development processes that are carried out today also employ software standards. The quality assurance function also works to check the compliance of all software development functions with their related standards. In that sense, if there is a flaw in the quality assurance task of a software development process, the result is reflected in the form of a **lack of compliance with the standards** in all development tasks. Because testing function is dependent upon three other functions for maintaining a certain ease of conductance, such a case will result in a significantly painstaking and long testing task. Lack of standards will cause difficult interpretation of all relevant documents, usually resulting in a lack of needed information for forming effective test scenarios, and thus a redundant testing act.
- As with any other role in the software development process, **delays in preceding tasks** reflect to their successors, which in effect alter their schedules for completion. Because testing starts slower in the beginning of the software development cycle through analysis work, and speeds up towards the end with unit and integration dynamic testing and iterations of software development cycles, the effect of such accumulated delays is relatively larger than other tasks in the case of software testing.

2. All of the factors highlighted as having importance in the construction of test cases apply to this section. Firstly, the **test cases** should be chosen as **easy to construct** as possible, since a complex test case designed to exercise a certain function of a software program will be more costly than a simple test case with the same purpose. Furthermore, the **cost of executing test cases** will also reflect in this calculation, giving the simpler one greater cost advantage. Thirdly, the results of some test cases may prove to be **difficult to calculate or obtain**. This may occur either because of the complexity of the required operations for obtaining the result, or because of a lack of knowledge of the test engineer in the specific area of operation. Finally, test cases may not be directed at the portion of more commonly expected errors in the

software program, thus increasing the total number of test cases required for complete assurance of confidence in the system.

3. Some further problems are caused when virtual teams are used in software development, due to the geographical isolation between the team members. These problems are as follows:

- Effective keeping of documents is more difficult in distributed teams due to the possible **lack of communication** at times. This difficulty is reflected in all functions of software testing, and even more so in the case of testing and quality assurance, since these two tasks depend primarily on requirements and specification documents on the components of the software program. Related problems in software testing occur during the generation of test scenarios, where some parts of the mentioned requirements or specifications on the program, or bits of information on certain special program attributes may occasionally be among the missing information in the poorly kept documents.
- Similarly, in virtual teams in which different components are produced by geographically isolated teams, lack of proper user's manuals is a common problem. This similarly decreases the efficiency and coverage of the test data, especially in the case of black-box testing, since user's manuals a form of after-design representation of what is expected from the program.
- In some cases of distributed software development of relatively small programs, extra strict standards for the formats of documents kept in the team may require the test engineers to spend large amounts of time on the preparation of these documents, in exchange for actual testing work. It was mentioned previously that proper documentation is indeed very important for successful software development, and especially in the case of distributed teams in geographically isolated areas. On the other hand, it should be noted distributed teams might suffer significant losses of efficiency if this importance is not interpreted correctly. The determination of the optimum amount of effort that should be put in the formatting and preparation of required documents is also of utmost importance to any software development project and is within the initiative of the project manager and the quality control officer.
- In certain cases of distributed software development where the testing team is also distributed across areas in addition to the rest of the team members, some test cases may be repeated due to lack of efficient communication among the testing team. Such redundancies will theoretically result in a more reliable testing task, but with almost no contribution in practice, since the resources used in such repetitions may easily be

salvaged by using different test cases instead, and providing a wider coverage of possible errors and a greater efficiency of testing.

CHAPTER 4

Case Study: CAIRO

All of the studies on testing and on testing in distributed teams were applied to the development of the next generation of a chosen software program, within the scope of a Course 1 graduate class, by the new DiSEL-99 team. The US wing of the distributed-class, concerning testing applications, was offered under the name of 1.120 - Distributed Software Engineering Laboratory - Collaborative Information Technology Systems Development in a Distributed Environment, at the Massachusetts Institute of Technology (MIT), Department of Civil & Environmental Engineering. The development of the next generation of the software program called CAIRO was performed within the scope of the 1.120 - Information Technology Master of Engineering Project for Fall 1998, IAP 1999, and Spring 1999.

Students in the DiSEL lab worked in geographically-distributed teams located in Mexico, and the US, throughout the project. These teams were, in the most general definition, assigned the task of improving Internet-based communication tools that would enable synchronous and asynchronous collaboration.

4.1 Project Overview

4.1.1 Objectives

The objective of the DiSEL-99 Project was to deliver a functional version of an existing software program called CAIRO (Collaborative Agent Interaction and synchROnization). The program would aim to create a more responsive environment for social interaction and casual contact for users. The new version would be delivered on time, would include all specified requirements in the created Requirements Document (CAIRO Project: Dec. 1998, Requirements Document), and would be in compliance with the quality standards specified in the related Quality Assurance Plan (CAIRO Project: Dec. 1998, Quality Assurance Plan).

Work activities were assigned priorities based on their contribution towards on-time delivery of the software. These activities included all development-related work items, such as the design, implementation, and testing of the software. Other activities included the management and monitoring of the design, implementation, and preparation of related documents.

All project preparatory work, which involved the completion of the Requirements Document, and the Quality Assurance and Configuration Management Plans, were completed in January 1999. The remaining time was allocated to the design, implementation and the testing of the product in four iterative cycles.

The creation of the next generation CAIRO was performed as a collaborative effort between two geographically distributed environments. The software development team consisted of 10 members at MIT and 4 members at CICESE (Centro de Investigación Científica Y de Educación Superior de Ensañada). Meetings between all members of the CAIRO project team were conducted using Microsoft NetMeeting™ for meetings that involved presentation sharing, and using CAIRO v1.0 for regular meetings held between department members.

The major resources used throughout the project were a Programmer Development Environment (PDE) chosen by the programmers, three PC's running on a Windows-based OS

with Internet access, audio and video equipment to be used as peripherals during distributed meetings, and copies of previous documentation related to earlier versions of CAIRO.

4.1.2 Project Deliverables

The deliverable of the DiSEL-99 Project was a next generation CAIRO software package that included the following:

- A CD that containing the code for the software and a "pdf" format of the user's manual
- A hardcopy of the User's Manual
- A testing report of the last version of CAIRO
- A design report of the last version of CAIRO
- A CD containing all documentation related to the DiSEL-99 Project
- Hardcopies of the same documentation

4.1.3 Initial State of CAIRO

In the beginning of the DiSEL-99 Project, CAIRO was obtained as the result of previous studies performed within the scope of the DiSEL-98 project. It was designed as a base program, which was then calibrated to run with a separately designed add-on module called CLIQ [Yang, 1998]. The distributed software application program was designed as a parallel multi-media network system, which utilized the World Wide web as the media of communication. The delivered package was an operational software program, which had numerous bugs interfering with the smoothness of its running. Furthermore, owing to past problems in communication, there was a significant lack of documentation on the initial software development efforts performed on CAIRO.

The software development effort for the DiSEL-99 Project for the modification of CAIRO was planned in four stages, in view of the Requirements Analysis Document Version 3.0 (See References for details).

4.2 Testing of CAIRO

4.2.1 Software Development Cycles

The cycles and corresponding testing work-items in the four stages of the testing of CAIRO were as listed below:

CYCLE 1 – Testing of initial CAIRO for stability

1. Downloading
2. Installation
3. Program launch (Startup)
4. Creating / Logging into a forum
5. GUI (avatars, buttons, menus)
6. Chat component (talk, sidetalk)
7. Whiteboard component
8. Saving, loading, quitting

CYCLE 2 – Unit testing of the individual VRML interface

1. Navigation controls (instant response)
2. Navigation controls (long term response)
3. Visual response to navigation (same environment)
4. Visual response to navigation (switching environments)
5. General user friendliness according to requirements
6. Logical correctness of environments/participants
7. Other functions

CYCLE 3 – Integration testing of VRML environment and CAIRO canvas

1. All of the tests in CYCLE 2 repeated
2. Participant images / mouse listener functions
3. Whiteboard / mouse listener functions
4. Status displays (talk, side-talk, busy)
5. Interaction displays (request chat, accept)

6. Installing, quitting, restarting
7. Saving, loading

CYCLE 4 – System testing of CAIRO with VRML environment and awareness tools

1. All tests in CYCLES 2 and 3 repeated
2. Correctness of the data output by awareness tool
3. Creating, logging into a forum
4. Whiteboard
5. Chat – talk, side-talk
6. Protocols, different meeting rooms

Phase I of the software development of CAIRO involves the improvement of the initial product. It involves Cycle 1 of the software life cycle. The second phase aims the implementation of new features into CAIRO, and is comprised of cycles 2, 3, and 4.

The ANSI/IEEE Std 1008-1987 Standard for Software Testing was employed as the standard throughout the testing effort. Although the testing team was localized entirely at MIT, the software development of CAIRO was performed in a distributed environment, where members of the development team were located at two geographically distinct locations. Despite the localized testing team, the total testing effort was still carried out a distributed team, where programmers also contributed to the static analysis and the white-box testing of the product. In addition, during the DiSEL-99 project, the testing team also worked in collaboration with the analysts, and the designers, and in supervision of the quality control officers, as well as the project manager.

4.2.2 Classification of the Testing of CAIRO

The testing of CAIRO can be classified in the following types:

- The testing of CAIRO was carried out entirely in the **alpha testing** stage. The development, including the testing of the product involved only DiSEL-99 team members. No testing outside MIT and CICESE was carried out, and hence the product was not exposed to any potential users.

- Partially complete **static testing** was performed for all cycles in the development of CAIRO. During the static testing stages, the program specifications and code were inspected. This testing effort was lead by the programmers for the primary purpose of becoming familiar with the code, and a secondary purpose of checking the code for faults. Due to the informal nature of this procedure, no reports were produced. Due to the time constraints on the project combining with the limitation of human resources, the programmers were assigned to coding and simultaneous dynamic testing in collaboration with the testing team, after a short period of static analysis. The new formation proved to be more efficient in the later stages of the testing effort, where additional help was needed from the programmers in the **white-box testing** and debugging of CAIRO.
- **Dynamic testing** was used extensively in the testing of CAIRO. The purpose of this testing effort was to reveal errors in the program by executing a selected set of test data (test cases) in the compiled program code. The results were reported, and compared to the expected results given in the Requirements Document Version 3.0. Individual results that were not reported in the relevant documents were calculated by methods of **symbolic testing** where appropriate. Refer to Appendix D.1 – Software Testing Plan Version 1.0 for a detailed list of test cases used by the testing team.
- A **bottom-up testing** strategy was preferred due to reasons of required simplicity and fast development. The initial cycle in the software development plan was the testing of the existing CAIRO product, as delivered by the DiSEL-98 team. The system testing was carried out beginning from the lowest level units, such as the whiteboard or the chat modules. The following cycles were carried out in the appropriate order for the bottom-up development of next generation CAIRO, starting with the **unit testing** of the individual VRML interface, continuing through the **integration testing** of the VRML environment and the CAIRO canvas, and finalizing with the **system testing** of CAIRO with the VRML environment and awareness tools.
- **Black-box testing** was used for the purpose of checking the software against its functional requirements. This was the only extensively used testing strategy, given the programming capabilities of the testing team. Due to limited human resources, the testing team worked in collaboration with the programmers throughout the project for testing procedures requiring the analysis of the source code. The testing team was not exposed to the operation of the program, or the source code of CAIRO.
- Testing of CAIRO, in terms of the selection of the test cases that were executed, was carried out in the form of **selective, structural testing**. Statistical testing was not applicable due to

the functional result type of CAIRO, as opposed to numerical, and exhaustive testing was impossible to perform given the high number of decision branches. The selected test cases were prepared so as to reflect as many possible situations as possible, and were derived in compliance with the Requirements Document Version 3.0. As many test cases as was feasible, in perspective of resource limitations and time constraints were executed.

4.2.3 Test Cases

Prior to the selection of test cases for the testing of CAIRO, input and output characteristics were determined, in accordance with the IEEE Standards [ANSI/IEEE, 1987]. The findings were as follows:

The means of input of data into CAIRO by the user were determined as:

1. Keyboard
2. Mouse
3. Disk drive (floppy / zip)
4. CD-ROM
5. Hard Disk
6. Internet downloading
7. (Internet) Document forwarding by other users

Where (1), and (2) in fact triggered (3), (4), (5), (6), and (7).

Output from CAIRO would be in the forms given below:

1. Output to screen (screen display).
2. Output to printer (print out)
3. Output to files in disk drives (floppy, zip), CD-R drives, and hard disk drives.
4. Output to a connected participant's computer (to be viewed by the participant in either of the 3 above methods).

Following the recording of the above data, specific test cases were constructed for the testing of CAIRO for each cycle of the software development process. These test cases were constructed using the use cases specified in the Requirements Analysis Document Version 3.0 (see References for details), the nature of which was as given below:

1. A casual contact capability must be provided.
2. Sensing devices must provide a measure of attitude (or affective state) of the users.
3. A viewing environment must provide feedback a user about the state of the other users.
4. A message board capability must be provided.
5. A private messaging capability must be provided.
6. A security infrastructure must be provided.

As previously mentioned, test cases were selected according to constructed test scenarios, using a random black-box methodology. Since the operation of CAIRO was performed through a user-friendly GUI, the corresponding test cases consisted of data input in the form of mouse clicks on relevant buttons of the GUI in order to launch the desired component or operation. They were created from interactive test scenarios with a selective structural test case generation approach. A sample of the constructed test scenario is as given below:

Test Case #1[1]

1. Chairman wants to create a forum. He/she logs in as the chairman. He/she is asked his/her password and username. He/she inputs an incorrect password. Permission is denied.
2. Chairman wants to create a forum. He/she logs in as the chairman. He/she is asked his/her password and username. He/she inputs the correct password. Permission is granted.
3. Chairman creates a forum for the **round table meeting**.
4. Chairman creates the agenda for the meeting.
5. The image of the chairman exists in the image database. It is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
6. The agent comments about the meeting environment selection.

[1] C. Kuyumcu. DiSEL-99 Software Testing Plan Version 1.0. See Appendix D.1.

7. Chairman views the images of other participants, who join the meeting on time. All images are displayed correctly. Every participant has a chair to sit in.
8. Chairman sends a greeting message to all participants, by talking to everyone simultaneously.
9. The chairman loads a previously saved chat from a previous meeting.
10. Chairman starts the meeting. Agenda is displayed and monitored.
11. Emotion (awareness) sensors connected to one of the participants displays his/her emotions. The chairman sees this display. (Both 2D and 3D-VRML cases). In the 2D case, the chairman sees an image representing the emotional state of the participant. In the 3D case, the chairman sees the change in the 3D image of the participant according to the sensed emotion.
12. The chairman side-talks with one of the participants.
13. One of the participants requests a side-talk with the chairman. He/she accepts, and they perform the side-talk operation.
14. One of the participants requests a side-talk with the chairman. He/she rejects.
15. The chairman opens the whiteboard. He/she draws a figure, and then edits it.
16. One of the participants edits the image drawn on the whiteboard. The chairman is able to see the modifications.
17. The chairman saves the image for future reference, and deletes it. He/she then loads a previously saved image. He/she edits the image, and saves it for future reference.
18. A late arriving participant requests permission to enter the meeting. The chairman accepts permission. He/she is successfully added to the meeting environment, and his/her image appears correctly. (Both 2D and 3D cases). The agent comments on the meeting environment selection again.
19. A passing by user requests permission to enter the meeting. (Casual contact).The chairman accepts permission. He/she is successfully added to the meeting environment, and his/her image appears correctly. (Both 2D and 3D cases). The agent comments on the meeting environment selection again.
20. A passing by user requests permission to enter the meeting. (Casual contact). The chairman rejects the request.
21. The chairman offers a change of the meeting environment. They change to **rectangular table meeting**.
22. The chairman offers a change of the meeting environment. They change to **conference hall meeting**.
23. The chairman offers a change of the meeting environment. All but two participants leave the meeting. The remaining change to **personal meeting**.

24. The agenda monitor warns the chairman of time running out.
25. The chairman saves the chat for future reference.
26. The chairman exits CAIRO successfully. (logout)

4.2.4 Test Results

The results of the tests performed on CAIRO in every cycle of software development were recorded and reported in the form of testing reports, to relevant team members, which always included the project manager and the programmers. A total of three testing reports and one testing plan were produced during the DiSEL-98 project work. The determined errors in CAIRO and its components were represented both in description, as well as in a tabular form in the submitted testing reports. A brief summary of the results is as given below:

CYCLE 1 – Testing of initial CAIRO for stability

Testing of CAIRO within the scope of the first cycle comprised the system testing of CAIRO in its initial state. Dynamic testing was performed by the testing team, simultaneously with the static analysis, which was performed by the programmers. All test scenarios were created using a black-box approach, and for real-time testing. A list of examples to commonly encountered problems is given in Table 1:

Table 1 - Summary of Errors in Cycle 1 Testing [Hor, 1998]

AREA	DESIRED FUNCTION	ERROR
GUI - Pop-up windows created in main CAIRO interface	All windows of described nature should be terminated when the "**X**" (window- close) button in the upper right hand corner of the window is single-clicked.	No response to clicks on the aforementioned button. Only way to terminate pop-up window is to click on "**Close**" or "**Cancel**" buttons inside the window.
CAIRO Main Menu: Meeting → Pending, Agenda Display	There should be functions attached to these options on the GUI: 1. Pending 2. Agenda Display	No response to single clicks on mentioned option icons. Minimum requirement: These options should be removed or should display a status message when invoked, stating that these functions have not been implemented.
CAIRO Main Menu: Driver → Audio, Video	There should be specific functions attached to these options on the GUI: 1. Audio 2. Video	No response to single clicks on mentioned option icons. Minimum requirement: These options should be removed or should display a status message when invoked, stating that these functions have not been implemented.
CAIRO Main Menu: Help	When single-clicked this option icon should invoke a "Help" function for the user.	No response when clicked. Option not disabled. Option should be removed or should display a status message when invoked, stating that the function has not been implemented.
CAIRO Main Menu: Quit → Save & Quit	Purpose of option is unclear. Save & Quit	No response to single-clicks on "Save" or "Quit" icons. Desired functions with these options should be determined. Appropriate functions should be attached.
CAIRO Main Menu: Agent	"Agent" button below the agent icon When single-clicked, facial expression shown in main window should change.	Single-clicks perform facial expression changes successfully. However, there is no apparent function that is associated with this change. User image expression inside meeting room does not change accordingly.
Whiteboard : Triangle	The triangle button should enable user to draw a triangle.	Unstable behavior. When drawing multiple triangles, later triangles are superimposed on previous ones in negative in a random fashion.
Whiteboard : ¾ Circle	Should enable user to draw a ¾ complete circle. Exact nature of circle is not defined.	Only allows ¾ circles with right lower corner open. No explanation in requirements about desired functionality.
CAIRO Main Window: "Side-talk" function	User should be able to close side-talk after side-talk end.	Unable to close side-talk. Tab in both the text chat box and the main window remains even if users are no longer talking.
General CAIRO usage	Numerous problems in maintaining network connection, detection of connected users, handling of user requests.	White-box testing required. Too many errors for debugging.

As seen form the table, most common errors were caused by unfinished coding in the initial product. Many of the function icons located in the GUI of CAIRO were not attached to any component inside the program. Furthermore, various typos, and various logic errors (i.e., triangle function in whiteboard) were detected. Finally, it was concluded that the initially delivered product was "incomplete" rather than having errors. As a result, all missing requirement specifications were defined, missing designs were completed, and coding of the missing parts as well as correction of the reported errors in the incomplete product were completed before the commencement of the second phase of the software development cycle of CAIRO.

CYCLE 2 – Unit testing of the individual VRML interface

The development of the VMRL interface used for creating 3D meeting environments was not completed at the initialization of Cycle 2 testing, according to the schedule given in the testing plan.[2] The interface required in the specifications would consist of a number of navigateable rooms, and various avatars for up to a certain number of different characters. At the time of testing, the avatars had not yet been integrated into the meeting rooms, and the meeting rooms were still under development. Furthermore, a completed, single-button initiated navigation sequence to pre-determined locations for avatars (i.e., to a chair) had not yet been established. The navigation inside the meeting rooms was only manual. As a result, testing was initiated as scheduled, but the testers limited their analysis to considerations of compliance of the current product with its requirements. A more informally reported initial testing was performed, similar to beta testing due to its very high level of freedom from conventional testing procedures. Although the process was completed with the use of computer programs such as CosmoPlayer™ for viewing VRML elements, it was closest to static analysis since there was no executable compiled program code involved in the testing. The main considerations were as follows:

1. Structure of the meeting rooms, in terms of variety, and capacity.
2. Aesthetics of the meeting rooms, in terms of colors and furniture, and general ambiance of the rooms.
3. Functionality of the rooms, in terms of layout of individual elements and participants.
4. Ease of modification of room capacity (i.e., addition or removal of chairs).

[2] C. Kuyumcu. DiSEL-99 Software Testing Plan Version 1.0. See Appendix D.1.

5. Ease of navigation in meeting rooms.
6. Verisimilitude of user perspective view, in terms of things in and out of view.
7. Aesthetics of hallways.
8. Functionality of hallways, in terms of access offered to an optimum number of rooms.
9. Verisimilitude of avatars, general looks.
10. Variety of available avatars that match different potential user profiles.
11. Verisimilitude of emotions displayed by avatars, and availability of all required emotions.
12. Motion capabilities of avatars, such as walking, running, sitting down, standing up, and talking.

The reported results were as follows:

Table 2 – Summary of Errors in Cycle 2 Testing [Kuyumcu, 1999]

Test Case	Tested	Passed	Corrected
Enter corridor – visibility	✔	✔	
View meeting names on doors in corridor	✔	✘	✘
Auto navigate to selected door in corridor	✔	✘	✘
Auto navigate to door at the end of the corridor	✔	✘	✘
Enter meeting room from corridor – visibility	✔	✘	✔
Enter next corridor from corridor – visibility	✔	✘	✘
Go back to corridor from meeting room – visibility	✔	✘	✔
Go back to previous corridor from current – visib.	✔	✘	✘
Auto navigate to selected chair in room	✔	✘	✘
Auto sit in selected chair in room	✔	✘	✘
Look up in room	✔	✘	✘
Look down in room	✔	✘	✘
Look left in room	✔	✘	✘
Look right in room	✔	✘	✘
View other avatars in room – standing up	✔	✘	✘
View other avatars in room – sitting down	✔	✘	✘
Add / remove chairs to room	✔	✘	✘
Go to new meeting room from current, w. avatars	✔	✘	✘
View emotions on avatars	✔	✔	
Change emotions of avatars	✔	✘	✘
View whiteboard in VRML room	✔	✘	✘
Activate whiteboard in VRML room	✘		
Activate pull-down menu about participant	✔	✘	✘

CYCLE 3 – Integration testing of VRML environment and CAIRO canvas

The integration testing of the VRML environment and the CAIRO canvas could not be performed within the scope of the DiSEL-98 project due to delays in the general project schedule, which also reflected in the design and development of the VRML environment interface.

CYCLE 4 – System testing of CAIRO with VRML environment and awareness tools

Similar to the third cycle of the software development of CAIRO, the completion of the awareness tool was delayed due to various project difficulties, eventually resulting in a change of requirements. As a result, the awareness tool, and thus Cycle 4 of the DiSEL-98 project were removed from the scope.

4.3 Comments for Testing in Distributed Environments

Various common problems encountered during software testing in distributed teams were described previously, in the Problems & Concerns for Testing in Distributed Teams section. Next, a case study of CAIRO was examined and some additional problems were determined. This section aims to address these problems, as well as offer additional points of consideration regarding the case study.

As a result of the experience gained through the testing practices performed during the software development of CAIRO, the following recommendations are appropriate:

1. In order to prevent problems faced in the case of **poorly written requirement and design documents**, which affect the generation of well-selected test scenarios, the test engineers have to prepare testing plans early in the software development cycle. In this manner, the analysts, the designers, and the quality assurance engineers can check the document, thus eliminating any possibility of a misinterpretation of the requirements, or program specifications.

2. All test engineers performing any type of testing on a software program must keep **good documentation** of their work and their results. In this manner, these documents can be passed on to the new test engineers (if different) in the next cycle of software development, so that these test engineers can have previous information about common past errors, weaknesses of the program, and common causes for encountered errors. Furthermore, good documentation will also provide information about the previously executed test scenarios, and may choose to execute the same or different test scenarios in the next set of testing procedures.
3. It is a good combination to include people with **high programming skills in the testing team** for the performance of white-box testing. The reason for this requirement is the previously addressed fact that black-box testing can detect only a small portion of all errors in a software program, and the location of these errors cannot be determined without debugging operations. Since debugging takes precious time and is therefore best when kept to a minimum, and because locating errors in other ways is not possible without programming skills and information about the source code, a black-bow only testing team is quite inefficient in testing of relatively large scale applications.
4. Testing teams should ideally have a **power** equal to that of the programming teams, if a software product is to be developed with satisfactory levels of reliability. This statement, of course, is flexible, given the different testing needs and the different reliability requirements of software programs. On the other hand, it should be noted that the discovery of their own errors by the programmers has been proven to be significantly less successful compared to error detection by dedicated testers [Infotech, 1979]. Given that early detection of errors provides great cost and time savings, it can be concluded that a good testing function will greatly increase the efficiency of a development cycle. For this reason, the testing team should be at least of comparable power in terms of total capabilities of test engineers and total resources, with the programmers. Moreover, this large testing team should include a blend of a larger variety of techniques in their efforts.
5. At today's level of technology, there is no single **ideal combination of available technologies** for software development or testing in distributed environments. As a result, members of a distributed testing team may benefit from basic computer-mediated capabilities such as e-mail, calendaring and scheduling systems, whiteboards, and document sharing. Such means of communication are expected to improve rapidly over the following years, especially with the introduction of the Hyper-Link Networks in place of the currently used Internet World Wide Web. In addition to these technologies, even the highly experienced

teams rely on other, non-computer-mediated technologies for communication such as telephone or facsimile.

6. For the case of distributed collaborative software development efforts, it has been proven with certain tests in certain circumstances that all teams benefit **from face-to-face discussion**, especially in the beginning of the development process [Lumsden at al, 1993]. Therefore, it may be beneficial for all members of a distributed software development team to have a face-to-face meeting in the beginning of the software life cycle.
7. Some tasks such as standard review meetings can be performed better with other means than face-to-face interaction, in order to provide less distractions and better focus on the material.
8. A distributed team needs a clear **strategy of matching available technologies** to the required task. This match should consider various factors, such as the needs for social presence, information richness, time constraints, and experience levels of team members.
9. **Complicated or unproven high technology** is not always a good choice for distributed teams. In its current state, for example, video conferencing is often not as effective as telephone conversation, or well-run audio conferencing.
10. **Bandwidth**, **cost**, and **compatibility** issues should be in consideration when assessing or estimating team performance.
11. Understanding of the **complexity of the tasks** is an important issue for all functions of software development, including testing. This understanding will prevent the selection of the wrong technology for the job.
12. **Business practices** and **ethics** may often be different between cultures. If the members of the distributed development team belong to different cultures, it is important to distinguish performance-based problems from problems resulting from cultural differences. In that sense, the team needs to clearly develop approaches to business practices and ethics, which every team member will understand and abide by.

CHAPTER 5

Conclusion

In the first chapter of this thesis, a general **introduction to functions of software verification and software testing** was given. The position and importance of software testing in software development efforts today was described, starting with an overview of important definitions and concepts related to software testing, through an analysis of the history of computing and software development, and finalizing with the discussion of motivating factors for testing.

In the second chapter, a more **detailed analysis of the software testing function** was provided with explanations of different testing strategies, approaches, and techniques. Moreover, the conditions for preferences among different means of testing were given, and their pros and cons were discussed. The chapter was concluded with an insight to test scenario generation methods prior to testing, and to standards used in software testing and related standard-issuing organizations.

The focus of this thesis is on distributed testing efforts, which is a common practice in the development of distributed software applications. For purposes of better coverage of the subject, the third chapter was dedicated to the analysis of peculiarities of **distributed software applications** in terms of software testing, and several different strategies were introduced. The chapter was finalized by stating numerous problems and points of concern about general software testing as well as distributed testing efforts.

The fourth chapter was created with the purpose of providing an example to the discussed theories, methodologies, and problems in software testing. Due to the focus of the thesis, the past software development and testing efforts in a distributed multi-media software application, namely **CAIRO**, were studied. Because the development of the next generation of CAIRO was also the Civil and Environmental Engineering Information Technology Master of Engineering project for the 1998-1999 academic year, a more thorough analysis of the past efforts were possible using the documentation for all of the performed tasks throughout the year. For the study of CAIRO, a brief overview of the project was given, followed by descriptions of the overall CARIO software development process, requirements, and the testing efforts performed within the scope of the project. A results-list of the conducted tests was provided, along with a list of the test scenarios used in the testing. The chapter was finalized with various comments for distributed software testing efforts, using examples from the case study, and certain proposed solutions for the previously stated problems.

As mentioned earlier, software applications are becoming more and more complex, which necessitates remarkable changes in current design and testing practices. Furthermore, with the growth of the role of computing and software programs in people's daily lives, requirements for sufficient software reliability also increase. In parallel to these changes, it is also seen that the realization of the need for successful and thorough software testing has become much stronger through the past few decades, and is still growing.

The growth of the role of software testing leads new software development teams to utilize greater numbers of experienced programmers as test engineers, in order to be able to satisfy all reliability conditions. On the other hand, this demand for better testers is becoming more and more difficult to fulfill with the continuously raised standards in software quality due to safety regulations, technological improvements, and fierce competition. This creates a second trend in software testing, in which testers are driven away from conventional testing techniques that depend on executing the program with different test cases to check for errors. Instead, automated testing, either in the form of programs that perform test data generation and software testing or in the form of direct comparison with benchmarks such as program mutation testing and algebraic testing gain popularity. In view of the increasing inclination towards this latter approach to testing, it is estimated that a significant portion of software testing in the future shall be carried out as static analysis. The dynamic part of testing that is performed by test engineers shall be transferred to the testing of "dynamic testing programs" to much higher reliability requirements,

instead of dynamic testing the program under consideration. Program testing shall be completed either with the aid of these reliable testing programs, or with direct comparisons as previously indicated.

Appendices

Appendix A.1 – Testing of Software Correctness

Correctness is a programmer's notion that the program performs according to the specification. The correctness testing is carried out based on a methodology given in Figure 1.3 and described below [Infotech, 1979].

At the start of the programming task, the programmer is supplied with a specification of the program. The specification may be as formal as a document which details the intended behavior of the program in all possible circumstances or it may be as informal as a few instances of what the program is intended to do. In practice, the programmer has several sources of information available to him, which comprise the specification. These may include a formal specification document, a working prototype, instances of program behavior such as in requirements document use cases, and a priori knowledge about similar software.

Working from this specification, the programmer develops the software. The test – or, more generally, validation – of the software lies in the comparison of the software product with the specification of intended behavior.

A specification provides a description of the input data for the program. This input data is called the **domain** of the program and is usually represented by D. The specification also provides a description of the intended behavior of the program on D. In the explanations given below, individual input data included in the domain D are represented by d, and the intended behaviors on input data d are represented by f(d). Despite this simple denotation f(d), these behaviors may be quite complex in practice. Similarly, although its actions may often be complex, the behavior of program P is represented by P^*. Thus, $P^*(d)$ forms a mathematical idealization of the behavior of program P on data item d. Furthermore, the shorthand notations f(D) and $P^*(D)$ are used to represent the intended behavior of all input data and the behavior of P on all input data, respectively.

The program P is said to be correct with respect to a specification f if $f(D)=P^*(D)$, that is, if P's behavior matches the intended behavior for all input data.

Certain problems arise in the application of this mathematical model. The above given mathematical model requires the presence of a completely formal requirements document, which can answer all questions for a target domain D in testing. This, of course is hardly the case, and documents are almost always unable to answer to what f(D) should be in checking $f(D)=P^*(D)$, for at least some portion of the d in the target domain. When the specification is not formalized, f(d) may be obtained by hand calculation, text book requirements or by the application of estimates obtained from simulations.

Usually, instead of dealing with such problems in program testing theory, the existence of on **oracle** is assumed. This oracle can judge for any specific d, whether or not $f(d)=P^*(d)$. This idealization is essential for software testing. In some other software testing strategies, the specification is assumed to be uniform. This means that the specification document must provide a method for computing f(d), and in some cases it should itself be executable. Other strategies, on the other hand, make no assumptions at all about how the oracle is to operate. They simply present the tester with $P^*(d)$. The determination of whether $P^*(d)=f(d)$ is left to the oracle.

This approach to software quality assurance from only a correctness point of view was found to be too restrictive at the NATO conferences [Science Program web page, 1999]. As a result, software quality was defined to include not only correctness but also attributes such as performance, portability, and adaptability. In this respect, software quality assurance should ensure that all the system's objectives are met.

Appendix B.1 – Standard Issuing Organizations

Contact information on various related standard-developing institutions are listed as follows:

1. AAMI (Association for the Advancement of Medical Instrumentation)
   3330 Washington Blvd, Suite 400
   Arlington VA 22201-4598 USA
   Phone: +1-703-525-4890
   Homepage: http://www.aami.org

   The AAMI standards program consists of over 100 technical committees and working groups that produce Standards, Recommended Practices and Technical Information Reports for medical devices. Standards and Recommended Practices represent a national consensus and many have been approved by the American National Standards Institute (ANSI) as American National Standards. AAMI also administers a number of international technical committees of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), as well as U.S. Technical Advisory Groups (TAGs).

2. AIIM (Association for Information and Image Management)
   1100 Wayne Avenue
   Suite 1100
   Silver Springs MD 20910 USA
   Phone: +1-301-587-8202
   Fax: +1-301-587-2711
   Homepage: http://www.aiim.org

   The Association for Information and Image Management (AIIM) International is the a global association bringing together the users and providers of information and document management technologies. Its focus is on helping users understand how these technologies can be applied to improve critical business processes. It does this by serving as the voice of the industry; providing a worldwide forum for information exchange; and providing an industry showcase so that users can anticipate, understand, and apply next-generation

technologies. An ANSI-accredited organization, AIIM is the universal home for standards development, and is the umbrella organization for industry coalitions working together to develop worldwide interoperability standards.

3. ANSI (American National Standards Institute)
   11 West 42nd Street
   13th Floor
   New York, NY 10036 USA
   Phone: +1-212-642-4900
   Fax: +1-212-302-1286
   Homepage: http://www.ansi.org

The American National Standards Institute (ANSI) has served in its capacity as administrator and coordinator of the United States private sector voluntary standardization system for 80 years. Founded in 1918, the Institute remains a private, nonprofit membership organization supported by a diverse constituency of private and public sector organizations.

Throughout its history, the ANSI Federation has maintained as its primary goal the enhancement of global competitiveness of U.S. business and the American quality of life by promoting and facilitating voluntary consensus standards and conformity assessment systems and promoting their integrity. The Institute represents the interests of its nearly 1,400 company, organization, government agency, institutional and international members through its headquarters in New York City, and its satellite office in Washington, DC

ANSI does not itself develop American National Standards (ANSs); rather it facilitates development by establishing consensus among qualified groups. The Institute ensures that its guiding principles -- consensus, due process and openness -- are followed by the more than 175 distinct entities currently accredited under one of the Federation's three methods of accreditation (organization, committee or canvass). ANSI-accredited developers are committed to supporting the development of national and, in many cases international standards, addressing the critical trends of technological innovation, marketplace globalization and regulatory reform.

4. ATIS (Alliance for Telecommunications Industry Solutions)
   1200 G Street NW
   Suite 500
   Washington DC 20005 USA
   Phone: +1-202-628-6380
   Homepage: http://www.atis.org

   The Alliance for Telecommunications Industry Solutions (ATIS) is a membership organization that provides the tools necessary for the industry to identify standards, guidelines and operating procedures that make the interoperability of existing and emerging telecommunications products and services possible. ATIS also sponsors the Internetwork Interoperability Test Coordination Committee (IITC).

5. ASQC (American Society for Quality Control)
   611 East Wisconsin Avenue
   Milwaukee WI 53202 USA
   Phone: 1-800-248-1946
   Phone: +1-414-272-8575
   Fax: +1-414-272-1734
   Homepage: http://www.asqc.org

   As the secretariat for the American National Standards Institute's (ANSI) ASC Z-1 Committee on Quality Assurance, ASQ provides direction on and builds consensus for national and international standards. ASQ volunteers play key roles in developing the ISO 9000 series standards, originally adopted nationally as the Q90 series standards and recently revised and redesignated as the Q9000 series standards. They do so through their involvement in the U.S. Technical Advisory Group for ISO Technical Committee 176, administered by ASQ on behalf of ANSI. ANSI represents the United States within ISO.

6. CSA (Canadian Standards Association)
   178 Rexdale Boulevard
   Etobicoke, Ontario, Canada
   M9W 1R3

Phone: +1-416-747-4058

Fax: +1-416-747-4149

Homepage: http://www.csa.ca

CSA International is an independent, not-for-profit organization supported by more than 8,000 members. It has a network of offices, partners, and strategic alliances in Canada, the U.S., and around the world. Established in 1919, CSA International is a leader in the field of standards development and the application of these standards through product certification, management systems registration, and information products.

7. DISA (Data Interchange Standards Association)
   1800 Diagonal Road, Suite 355
   Alexandria VA 22314 USA
   Phone: +1-703-548-7005
   Homepage: http://www.disa.org

   The Data Interchange Standards Association (DISA – an association for electronic commerce) is a non-profit organization that supports the development and use of electronic data interchange standards in electronic commerce. In 1987, in response to the rapidly growing number of industries employing the X12 standards, DISA was chartered by the American National Standards Institute (ANSI) to provide the committee with administrative support. In addition to national standardization efforts, DISA offers the opportunity for its members to participate in the international standards-setting process through its affiliation with the United Nations Rules for Electronic Data Interchange For Administration, Commerce and Transportation (UN/EDIFACT).

8. ECMA (European Computer Manufacturers Association)
   114 Rue du Rhone
   CH1204 Geneva
   Switzerland
   Phone: 011 41 22 35 3634
   Fax: 011 41 22 86 5231

9. EIA (Electronic Industries Association)
   2500 Wilson Boulevard
   Arlington VA 22201 USA
   Phone: +1-703-907-7500
   Fax: +1-703-907-7501
   Homepage: http://www.eia.org

   For nearly 75 years EIA has functioned as a trade organization representing the U.S. high technology community. EIA has created a host of activities to enhance the competitiveness of the American producer including such valuable services as, technical standards development, market analysis, government relations, trade shows, and seminar programs.

10. FDA (Food & Drug Administration)
    FDA (HFE-88)
    5600 Fishers Lane
    Rockville, MD 20857
    Phone: 1-888-INFO-FDA (1-888-463-6332)
    Homepage: http://www.fta.gov

    U.S. food and Drug Administration is a government agency functioning under the Department of Health and Human Sciences. The primary objective of FDA is to ensure that all consumed food is safe and wholesome, cosmetics are not of harmful nature, and medicines, medical devices, and radiation-emitting consumer products such as microwave ovens are safe and effective. For the latter objective, FDA issues its own set of standards for approving electronic consumer products. For the case of products utilizing software, such as medical devices, FDA requires that the development of this software including the testing be in compliance with the FDA standards for software development.

11. IEC (International Electrotechnical Commission)
    3 rue de Varembé
    CH 1211 Geneva 20
    Switzerland
    Phone: 011 41 22 919 0211
    Fax: 011 41 22 919 0301

Homepage: http://www.iec.ch

Founded in 1906, the International Electrotechnical Commission (IEC) is the world organization that prepares and publishes international standards for all electrical, electronic and related technologies. The IEC was founded as a result of a resolution passed at the International Electrical Congress held in St. Louis (USA) in 1904. The membership consists of more than 50 participating countries, including all the world's major trading nations and a growing number of industrializing countries.

12. IFIP (International Federation for Information Processing)
    Hofstraße 3
    A-2361 Laxenburg, Austria
    Tel.: +43 2236 73616
    Fax: +43 2236 736169
    E-mail: ifip@ifip.or.at
    Homepage: http://www.ifip.or.at

IFIP is a non-governmental, non-profit umbrella organization for national societies working in the field of information processing. It was established in 1960 under the auspices of UNESCO (United Nations Educational, Scientific, and Cultural Organization) as an aftermath of the first World Computer Congress held in Paris in 1959. IFIP's mission is to be the leading, truly international, apolitical organization which encourages and assists in the development, exploitation and application of Information Technology for the benefit of all people.

13. ISA (International Society For Measurement and Control)
    PO Box 12277
    67 Alexander Drive
    Research Triangle Park, NC 27709
    Phone: +1-919-549-8411
    Fax: +1-919-549-8288
    Homepage: http://www.isa.org

14. ISO (International Standards Organization)
    1 rue de Varembé
    Case Postale 56
    CH1211 Geneva 20
    Switzerland
    Phone: 011 41 22 749 0111
    Fax: 011 41 22 733 3430
    Homepage: http://www.iso.ch

    The International Organization for Standardization (ISO) is a worldwide federation of national standards bodies from some 130 countries, one from each country. It is a non-governmental organization established in 1947. The mission of ISO is to promote the development of standardization and related activities in the world with a view to facilitating the international exchange of goods and services, and to developing cooperation in the spheres of intellectual, scientific, technological and economic activity. ISO's work results in international agreements, which are published as International Standards.

15. ITI (Information Technology Industry Council)
    (formerly CBEMA -- Computer and Business Equipment Manufacturers Association)
    1250 Eye Street NW
    Suite 200
    Washington DC 20005 USA
    Phone: +1-202-737-8888
    Fax: +1-202-638-4922
    Homepage: http://www.itic.org

    ITI, the Information technology Industry Council, represents various leading U.S. providers of information technology products and services. ITI member companies are responsible for more than 16% of all U.S. industrially-funded research and development and over 50% of all information technology research and engineering.

16. ITU (International Telecommunication Union)
    (formerly CCITT – international Telegraph and Telephone Consulative Committee)
    United Nations Plaza de Nations

CH 1211 Geneva 20
Switzerland
Phone: 011 41 22 99 511
Fax: 011 41 22 33 7256
Homepage: http://www.itu.int

The ITU, headquartered in Geneva, Switzerland is an international organization within which governments and the private sector coordinate global telecommunication networks and services.

17. NCITS (National Committee for Information Technology Standards)
Information Technology Industry Council
1250 Eye Street NW / Suite 200
Washington, DC 20005
Phone: 202-737-8888
Fax: 202-638-4922
Homepage: http://www.ncits.org

Founded in 1961, NCITS operated under the name of Accredited Standards Committee X3, Information Technology, until 1996. NCITS's mission is to produce market-driven, voluntary consensus standards in the areas of:

- multimedia (MPEG/JPEG),
- intercommunication among computing devices and information systems (including the Information Infrastructure, SCSI-2 interfaces, Geographic Information Systems),
- storage media (hard drives, removable cartridges),
- database (including SQL3),
- security, and
- programming languages (such as C++).

18. NEMA (National Electrical Manufacturers Association)
1300 North 17th Street, Suite 1847
Rosslyn, VA 22209 USA
Phone: (703) 841-3200

Fax: (703) 841-3300
Homepage: http://www.nema.org/

NEMA was formed in 1926, with the aim of to providing a forum for the standardization of electrical equipment.

19. NISO (National Information Standards Organization)
PO Box 1056
Bethesda, MD 20827 USA
Phone: (301) 975-2814
Homepage: http://www.niso.org

The National Information Standards Organization (NISO) develops and promotes technical standards used in a wide variety of information services. NISO is a nonprofit association accredited as a standards developer by the American National Standards Institute. NISO has developed standards for Information Retrieval (Z39.50), 12083 (an SGML Tool), Z39.2 (Information Interchange Format), Codes for Languages and Countries, and Z39.18 (Scientific and Technical Reports).

20. NIST (National Institute of Standards and Technology)
100 Bureau Drive
Gaithersburg, MD 20899-0001
Phone: (301) 975-NIST
Homepage: http://www.nist.gov

The National Institute of Standards and Technology (NIST) was established by Congress "to assist industry in the development of technology ... needed to improve product quality, to modernize manufacturing processes, to ensure product reliability ... and to facilitate rapid commercialization ... of products based on new scientific discoveries."

An agency of the U.S. Department of Commerce's Technology Administration, NIST's primary mission is to promote U.S. economic growth by working with industry to develop and apply technology, measurements, and standards.

Appendix B.2 – Military departments Utilizing DoD T&E Standards

The list of the US military departments, which utilize the Department of Defense (DoD) Test and Evaluation (T&E) Standards for Electronic Warfare Systems is as follows [DefenseLINK, 1999]:

1. AFOTEC – Air Force Operational Test and Evaluation Center
2. OPTEC - Army Operational Test and Evaluation Command
3. COMOPTEVFOR - Navy Commander Operational Test and Evaluation Force
4. DTIC - Defense Technical Information Center
5. ACQ Web - Office of the Under Secretary of Defense for Acquisition and Technology
6. DAU - Defense Acquisition University
7. NAWCWPNS - Naval Air Warfare Center Weapons Division
8. NAWCWPNS: Code 418000D - Naval Air Warfare Center Weapons Division: Survivability Division
9. AFFTC - Air Force Flight Test Center
10. JITC - Joint Interoperability Test Command
11. AAC/46OG/OGM - Air Armament Center/46th Test Wing/ Munitions Test Division

Appendix C.1 – Sample Program for Strategy Comparisons

A sample run using ATTEST is shown below [Infotech, 1979]. The testing criteria used for this run included loop boundary conditions, language dependent conditions, statement coverage, and branch coverage. The predicates generated for the paths are all fairly simple and do not demonstrate the full capabilities of the test data generation component. However, the capabilities of the path selection component are demonstrated.

```
Block                                    Source

1              INTEGER MRCHTB(5,2,),TBSIZ,DEPT,INVNUM,MIN,MAX
1              DATA TBSIZ/5/
0      C
   0   C       INVENTORY NUMBERS
0      C
1             DATA MRCHTB(1,1,)/56965/,MHCHTB(2,1)/31415/,
1      C       MHCHTB(3,1)/14736/,MRCHTB(4,1)/12345/,MRCHTB(5,1)/91789/
0      C
   0   C       MIN AND MAX INVENTORY NUMBERS
0      C
1             DATA MIN/11111/,MAX/99999/
0      C
   0   C       DEPARTMENTS HJANDLING INVENTORY
0      C
   1              DATA MRCHTB(1,2)/3/, MRCHTB(2,2)/1/, MRCHTB(3,2)/3/,
   1   C       MRCHTB(4,2)/2/, MRCHTB(5,2)/1/
0      C
0      C
   2              READ (5,100) INVNUM
   3              CALL LINSCH(INVNUM, MRCHTB,TBSIZ,MIN,MAX,DEPT)
4              WRITE (6,100) INVNUM,DEPT
5              STOP
```

```
0       C
    0   100     FORMAT
1               END


    1           SUBROUTINE LINSCH(INVNUM,
    1     C         MRCHTB,TBSIZ,MIN,MAX,DEPT)
    1                 INTEGER INVNUM,TBSIZ, MRCHTB(TBSIZ,2),MIN,MAX,DEPT
    0   C
    0   C
    2                 IF(INVNUM.LT.MIN)
    3               $GOTO 900
    4                 IF(INVNUM.GT.MAX)
    5               $GOTO 900
    0   C
    6                 DO 10 INDEX=1,TBSIZ
    7                 IF(MRCHTB(INDEX,1),EQ.INVNUM)
    8               $GOTO 20
    9   10      CONTINUE
    10          DEPT=0
    11          RETURN
    0   C
    12  20      DEPT= MRCHTB(INDEX,2)
    13          RETURN
    0   C
    0   C       ERROR RETURN
    0   C
    14  900     DEPT=-1
    15          RETURN
    1           END
```

Path 1:

Predicates:

(INVNUM < 11111)
(INVNUM > 99999)
(56965 = INVNUM)
(31415 = INVNUM)
(14736 = INVNUM)
(12345 = INVNUM)
(91789 = INVNUM)

Test Data:

INVNUM = 11111

Path Output:

DEPT = 0

Path 2:

Predicates:

INVNUM < 11111

Test Data:

INVNUM = 0

Path Output:

DEPT = 0

Path 3:

Predicates:

(INVNUM < 11111)
(INVNUM > 99999)

Test Data:

INVNUM = 100000

Path Output:

DEPT = -1

Path 4:

Predicates:

(INVNUM < 11111)
(INVNUM > 99999)
(56965 = INVNUM)

Test Data:

INVNUM = 56965

Path Output:

DEPT = 3

Path 5:

Predicates:

(INVNUM < 11111)

(INVNUM > 99999)

(56965 = INVNUM)

(31415 = INVNUM)

(14736 = INVNUM)

(12345 = INVNUM)

(91789 = INVNUM)

Test Data:

INVNUM = 91789

Path Output:

DEPT = 1

Appendix D.1 – Software Testing Plan Version 1.0

SOFTWARE TESTING PLAN v1.0

This testing plan was written in accordance with the ANSI/IEEE Std 1008-1987 Standard for Software Unit Testing. It consists of the following parts:

1. Scope of Plan
2. Definitions Related to Unit Testing
3. Unit Testing Activities
4. General Approach to Unit Testing
5. Input and Output Characteristics
6. Procedures for Testing / Tasks
7. Schedule for 2nd Term Unit Testing
8. References

1. Scope of Plan

The Testing Plan v1.0 aims to cover all testing activities to be performed in the software development of the later versions of CAIRO v2.0, during the second term of the 1998-1999 academic year. It includes a schedule which shows when all necessary tasks will be performed and when additional reports, namely two testing reports and the final testing report will be given out, as well as a description of methods to be used, and theories, requirements that apply.

This plan is a specific description of the methods selected for the realization of the black box unit testing of CAIRO versions 2.1 and up. Software Unit Testing, as it is used here, can be defined as a process that includes the performance of test planning, the acquisition of a test set, and the measurement of a test unit against its requirements. In all three cycles existing in the proposed management plan for the development of CAIRO, unit testing is to include only the newly added component. Only the final stage is to include the whole package. It should be noted

that although this plan includes all testing activities to be performed during the term, it does not include any standards or processes of debugging. Furthermore, the plan does not address issues such as reviews, walkthroughs, or inspections, which have been included in the Quality Assurance Plan.

The requirements are often expressed by referencing to a previous document prepared within the scope of the project. In the case of this Testing Plan, all relevant subjects shall be referenced to Requirements Document version 3.0.

2. Definitions Related to Testing

This section defines some of the key terms used in the IEEE standard.

CAIRO- Collaborative Agent Interaction and synchROnization

Data characteristic- An inherent trait, quality, or property of data (for example, arrival rates, formats, value ranges, or relationships between field values).

Software characteristic- An inherent trait, quality, or property of software (for example, functionality, performance, attributes, design constraints, number of states, lines of branches).

Software testing case (test case)- Any event occurring during the execution of a software test that requires investigation.

Test objective- An identified set of software features to be measured under specified conditions by comparing actual behavior with the required behavior described in the software documentation.

Test unit- A set of one or more computer program modules together with associated control data, (for example, tables), usage procedures, and operating procedures that satisfy the following conditions:

- All modules are from a single computer program.

- At least one of the new or changed modules in the set has not completed the unit test.
- The set of modules together with its associated data and procedures are the sole object of a testing process.

Mouse listener- A listener is an object in a program code that gets called when the desired event happens. For a mouse listener, the event is the clicking (single or double click, according to design) of the mouse on a selected point on the screen, such as a button or an icon.

Casual contact- Unplanned, spontaneous social interaction among workgroup or class participants occurring outside of planned lectures, sessions and meetings.

Affective State- Internal dynamic state of an individual, when he/she has an emotion.

3. Unit Testing Activities

This section presents as a list, the activities involved in the unit testing process, including future steps for revision of the general software-testing plan. They are determined as follows:

1. General Approach, Resources, Schedule

1.1 Plan Inputs

(1) Collect all project plans

(2) Study the Software Requirements Documentation

1.2 Plan Tasks

(1) Specify a general approach to Unit Testing

(2) Specify completeness requirements

(3) Specify termination requirements

(4) Determine resource requirements

(5) Specify a general schedule

1.3 Plan Outputs

(1) Gather General Unit Test Planning information

(2) Create Unit Testing General Resource Requests

2. Determine the Features to be Tested

2.1 Determine Inputs

(1) Study Unit Requirements Documentation

(2) Study Software Architectural Design Documentation

2.2 Determine Tasks

(1) Study the functional requirements

(2) Identify additional requirements and associated procedures

(3) Identify states of the unit

(4) Identify Input and Output data characteristics

(5) Select elements to be included in the Unit Testing

2.3 Determine Outputs

(1) Create a list of elements to be included in the Unit Testing

(2) Create Unit Requirements Clarification Requests as necessary

3. Refine the General Plan

3.1 Refine Inputs

(1) Revise list of elements to be included in the Unit Testing

(2) Revise General Unit Test Planning Information

3.2 Refine Tasks

(1) Refine the approach to Unit Testing

(2) Specify Special Resource Requirements as necessary

(3) Specify a detailed schedule

3.3 Refine Outputs

(1) Gather specific Unit Test Planning Information

(2) Create Unit Test Special Resource Requests as necessary

4. Design the Set of Tests

4.1 Design Inputs

(1) Study the Unit Requirements Documentation

(2) Study the list of elements to be included in the Unit Testing

(3) Study the Unit Test Planning Information

(4) Study the Unit Design Documentation

(5) Study the specifications/ results from previous testings

4.2 Design Tasks

(1) Design the architecture of the Test Set

(2) Obtain explicit test procedures

(3) Obtain the Test Case Specifications

(4) Augment the set of Test Case Specifications based on Design Information

(5) Complete the Unit Test Design Specification

4.3 Design Outputs

(1) Study the Unit Test Design Specification

(2) Separate Test Procedure Specifications

(3) Separate Test Case Specifications

(4) Create Unit Design Enhancement Requests as necessary

5. Implement the Refined Plan and Design

5.1 Implement Inputs

(1) Study the Unit Test Design Specification
(2) Study the Software Data Structure Descriptions
(3) Test support resources
(4) Test items
(5) Test data from previous testing activities

5.2 Implement Tasks

(1) Obtain and verify test data
(2) Obtain special resources
(3) Obtain test items

5.3 Implement Outputs

(1) Verify test data
(2) Configure test items
(3) Create initial summary information

6. Execute the Test Procedures

6.1 Execute Inputs

6.2 Execute Tasks

(1) Run tests
(2) Determine results

6.3 Execute Outputs

7. Check for Termination

7.1 Check Inputs

(1) Check completeness and termination requirements

(2) Check execution information

(3) Check test specifications

7.2 Check Tasks

(1) Check for normal termination of the testing process

(2) Check for abnormal termination of the testing process

(3) Supplement the test set

7.3 Check Outputs

8. Evaluate the Test Effort and Unit

8.1 Evaluate Inputs

8.2 Evaluate Tasks

(1) Describe testing status

(2) Describe Unit's status

(3) Complete the Test Summary report

8.3 Evaluate Outputs

4. General Approach to Unit Testing

The general approach to unit testing is specified by the explanation of completeness, termination, and termination requirements.

a. General Approach (Unit Testing)

Black-box unit testing shall be used in this term's testing procedures. In this approach, the tester shall not be familiar with the program code or the design of the program. This is a more efficient method compared to white-box unit testing, since the tester shall not know for which exact purpose each component of the program is written, and thus shall attempt to require and test every function that a component of the program may seem to offer.

The Unit Testing of next generation CAIRO shall be carried out on three steps, excluding the preparation of the Testing Plan document. The first step shall be carried out at the end of the first software development cycle, which includes the design and implementation of the desired VRML environment setting, as a separate unit. It shall aim to fully test the implemented VRML tool according to the standards and the requirements specified in this document. The second step of Unit Testing shall be carried out at the end of the second cycle in the software development, which consists of the design and implementation of a fully viewable/useable VRML environment inside the CAIRO interface. This second step of Unit Testing shall aim to fully test the package of CAIRO and VRML combined, in terms of user interface functionality.

The final stage of the Unit Testing shall be carried out at the end of the third and final cycle of the software development. This last cycle will consist of the design and implementation of the fully functional CAIRO, with the awareness tool linked to the canvas. The third stage of Unit Testing shall aim to fully test this final version of CAIRO, as a complete package having VRML-environment and awareness displays inside the user interface. Pre-tested components shall also be re-tested for interference with realized modifications to the software during the three cycles.

b. Specification of Completeness Requirements

All completeness assumptions in the Unit Testing of CAIRO shall be made according to the requirements set below:

1. For testing of the navigation controls in cycle 1, in the instant response case, the response given by the VRML program running inside the canvas of CAIRO to a given command for navigation should be the commencement of the desired navigation sequence in the direction specified with the command.
2. For testing of the navigation controls in cycle 1, in the long term response case, the response given by the VRML program running inside the canvas of CAIRO to a given command for navigation should start as specified in (1), and the navigation should continue without any breaks in the desired direction.
3. For the visual response to navigation through the same environment, the environment should not change shape or color, and should maintain integrity throughout.
4. For the visual response to navigation through different environments, all environments should keep their change shape and color, and maintain integrity throughout. Moreover, there should not be any breaks or distortions in the graphics when going in or out of one environment.
5. For general user friendliness of the VRML interface, the black-box tester is to decide subjectively on the user friendliness of the interface and report to the analyst with his comments, if the interface is not found to be sufficiently user friendly. If found necessary by either the tester, or the analyst, both shall test only the general interface together to reach the final decision.
6. For the logical correctness of environments, or participants, the black-box tester is to decide subjectively and report to the analyst with his comments, if there are some logic errors. If found necessary by either the tester, or the analyst, both shall test only this aspect of the tool together to reach the final decision.
7. For participant images, whiteboard, and mouse listener functions in cycle 2, the program should be able to start related functions or display related menus, when the corresponding images inside the canvas are single or double clicked by the tester.
8. For the status displays (talk, side-talk, busy), correct status should be displayed for corresponding actions, such as being able to see that someone is side-talking with you.
9. For the interaction displays (request chat, accept), correct messages should be displayed for corresponding actions, such as being able to see that someone is requesting a side-talk permission from you.
10. For the installing, quitting, restarting, saving, and loading tests, these functions should work, the same as in the original product, proving that the installation of the VRML meeting environment tool, and the awareness tool has not interfered with their operation.

11. For the correctness of the data output by the awareness tool, the data should match with the real life data, either measured, or known from experience.
12. For creating, or logging into a forum, the whiteboard, the talk, and the side-talk, these functions should work, the same as in the original product, proving that the installation of the VRML meeting environment tool, and the awareness tool has not interfered with their operation.
13. For the case of general program performance, since no performance requirements have been included in the requirement document, the black-box tester is to decide subjectively and report to the analyst with his comments, if there are some problems with the operation of the whole program or some of its components. The problems can range from long loading times to many crashes per day of CAIRO. If found necessary by either the tester, or the analyst, both shall test only this aspect of the tool together to reach the final decision.

c. Specification of Termination Requirements

Termination of the testing on a certain component of the software program shall occur in two cases. The first case shall be when the test for completeness test is passed successfully. The second case shall be when the completeness test is failed. For the "passed" cases, the output shall be written in the next tester's report and submitted. For the "failed" case, the certain component of the software program will remain labeled incomplete and an immediate report shall be made to the programmers at first for modification and correctness for full functionality, and to the project manager, analyst, and designers as deemed necessary. The testing will not be completed terminally but will restart as soon as the reported bugs or errors have been fixed. In the case of more critical errors, to be specified by the analyst as deemed necessary, such as applications of defense or security, different and strict requirements for the termination of testing may apply.

The status of the test case, as to whether it has passed or failed, shall be determined as follows:

1. The designed test case shall be carried out, and the results recorded. Test cases shall be repeated three, six, or nine times, according to the rules given below.
2. Unless otherwise specified by the quality assurance or the analyst (for applications of security and defense) the test case shall be labeled "failed" if it fails at least once in the given set of

consecutive trials. If it passes all trials, in the given number of consecutive repetitions, it shall be labeled "passed".

3. The number of consecutive trials for tests related to areas (1) and (2) in section [b. Specification of completeness requirements] shall be 9. For cases (5), and (6) the result shall be decided on all of the total trials carried out with other tasks, since these do not need a dedicated task. For the remaining cases, this number shall be 3.
4. All steps shall be repeated for SGI, Sun, and Windows NT/95 machines, and failures shall be classified with the corresponding platform name.
5. All steps shall be repeated for client as well as the server machine, and failures shall be classified with the corresponding user type.

d. Specification of Resource Requirements

The required resources are as follows:

1. Requirements Document 3.0 – reference 1.
2. Quality Assurance Document 2.0 – reference 2.
3. The ANSI/IEEE Std 1008-1987 Standard for Software Unit Testing.
4. The original "Stable CAIRO" CAIRO v2.0.
5. The VRML component.
6. VRML viewer for phase 1 testing, such as Cosmo Player. To be decided by the designers.
7. The VRML implemented version of CAIRO for cycle 2.
8. The VRML and awareness implemented version of CAIRO for cycle 3.
9. Requirements from the analyst regarding the need for strict testing termination requirements.
10. Draft user's manual from the designers for the second and third stage testing operations.
11. Draft user's manual from the User Interaction and Awareness Hardware Group for the third stage testing of the awareness tool.
12. Three PC's connectable to the Internet, running on Windows NT/95.
13. One SGI and one Sun machine, connectable to the Internet.

5. Input and Output Characteristics

a. Input Characteristics

The means of input of data into CAIRO by the user are determined as:

1. Keyboard
2. Mouse
3. Disk drive (floppy / zip)
4. CD-ROM
5. Hard Disk
6. Internet downloading
7. (Internet) Document forwarding by other users

In which cases (1), and (2) in fact trigger (3), (4), (5), (6), and (7).

Input with the keyboard may form the text for the text windows (such as chat), which is simply placed in a text box and transferred to the other user. This data shall not be interpreted by the program. Exceptions in the use may be caused during installation and setup, or during acceptance or rejection of requests coming from other users. Furthermore, the VRML environment may be designed to accept keyboard inputs as well, which, in such a case, shall be specified in the draft user's manual presented by the designers.

Input with the mouse may be in the form of directions or single or double clicks. Single or double clicks shall be tested according to their attached functions in the draft user's manual to be presented by the designers.

Other means of specified input shall be put into use through commands given by using the mouse or the keyboard. These input means shall be checked for loading, and downloading functions intended for CAIRO.

b. Output Characteristics

Output from CAIRO shall be in the forms given below:

1. Output to screen (screen display).
2. Output to printer (print out)
3. Output to files in disk drives (floppy, zip), CD-R drives, and hard disk drives.

4. Output to a connected participant's computer (to be viewed by the participant in either of the 3 above methods).

Output to screen shall be the most commonly used, as well as the best-tested means of output in CAIRO. Item (2) shall be tested for various cases inside the whiteboard, chat, and the awareness tool, according to the draft user's manual. Item (3) shall be tested for saving functions. Item (4) shall form the basis of the use of CAIRO over the Internet, since it depends on information transfer over the net, both in the form of downloading from an external source, as well as from one component of CAIRO into another running on a different machine.

6. Procedures for Testing / Tasks

a. Components to be Tested

The components to be tested in the total three stages of the Unit Testing of CAIRO are as listed below:

CYCLE 1 – Testing of the individual VRML interface

1. Navigation controls (instant response)
2. Navigation controls (long term response)
3. Visual response to navigation (same environment)
4. Visual response to navigation (switching environments)
5. General user friendliness according to requirements
6. Logical correctness of environments/participants
7. Other functions

CYCLE 2 – Testing of VRML environment inside CAIRO canvas

1. All of the tests in CYCLE 1 are to be repeated
2. Participant images / mouse listener functions
3. Whiteboard / mouse listener functions
4. Status displays (talk, side-talk, busy)
5. Interaction displays (request chat, accept)
6. Installing, quitting, restarting

7. Saving, loading

CYCLE 3 – Testing of CAIRO with VRML and awareness tool

1. All tests in CYCLES 1 and 2
2. Correctness of the data output by awareness tool
3. Creating, logging into a forum
4. Whiteboard
5. Chat – talk, side-talk
6. Protocols, different meeting rooms

In addition to the above steps, the Unit Testing for CAIRO shall be performed for multiple users for the second and the third steps. This multi-user testing case shall require two different data sets before interpretation for errors; namely data gathered from the client-side, and from the host-side.

Furthermore, the final stage of the Unit Test on CAIRO shall be applied on multiple platforms, namely on SGI, Sun machines, as well as PC's running on Windows NT/95 OS's.

The procedure shall consist of a direct application of the tasks specified above, with the detailed descriptions as given below, in a number of times as specified by the testing termination requirements. The result shall be placed in the matching of the three Tester's Reports for the three cycles, and an immediate report shall be made to the programmers, and at times to analysts, project manager, and the quality assurance in the case of a discovered error or bug.

b. Tasks for Testing

Tasks are given in detail below:

CYCLE 1 – Testing of the individual VRML interface

1. Navigation controls (instant response) shall be tested for the instant response. The task is to input various commands into the VRML interface via the input instruments specified earlier. In order to pass this testing task successfully, the VRML tool has to respond by starting the navigation action in the desired direction and manner. (What is desired by the command shall be judged according the draft user's manual to be supplied by the testers.)

2. Navigation controls (long term response) shall be a continuation task (1). It shall check to see if the commenced navigation activity (given that it is initially correct) continues in the correct path and form. The reason for separating the two tasks is mainly to apply a method of debugging. By using this method, it shall be possible to identify whether the inputs are matched to wrong navigation paths, which are among themselves correct, or whether the navigation paths are causing the errors.
3. Visual response to navigation (same environment) shall be tested subjectively by the tester, as mentioned previously. In the case of dissatisfaction by the visual performance of the VRML tool, the tester has the right to comment to the analyst, requesting a joint test by the two. The result shall be determined then on.
4. Visual response to navigation (switching environments) is a similar test case, which is applied to the visual performance when switching environments such as passing from one room to another, or entering or leaving a building. The result shall be determined similar to case (3).
5. General user friendliness according to requirements shall be decided upon by the tester, based on his/her use of the VRML tool throughout the application of other test cases, and through the use of the draft user's manual. It shall be reported to the analyst in the case of dissatisfaction, and action shall be taken by decision of the two parties then on.
6. Logical correctness of environments/participants shall be decided upon by the tester, based on his use of the VRML tool throughout the application of other test cases. It shall be reported to the analyst in the case of dissatisfaction, and action shall be taken by decision of the two parties then on. An example to a failure in such a case may be given as bodies without heads, or a chair floating in the air.
7. Other functions that may be added later by the designers shall be tested according to their task design, which shall be carried out following such a case. In the case of a high number of late additions, a separate Tester's Report shall be prepared and submitted.

CYCLE 2 – Testing of VRML environment inside CAIRO canvas

1. All of the tests in CYCLE 1 are to be repeated, with the same procedures given for cycle 1. The aim of these steps shall be to prove that addition of VRML to the CAIRO canvas has not interfered with the functioning of the VRML tool.
2. Mouse listener functions for the images of participants shall be tested simply by clicking or double clicking on the VRML images of different participants in meetings, and checking for desired actions. (As specified by the draft user's manual.)

3. Mouse listener functions for the whiteboard shall be tested by clicking on the whiteboard in the VRML environment and checking to see if the action activates the whiteboard. (Clicking action may be altered as specified by the draft user's manual.)
4. Status displays (talk, side-talk, busy) shall be tested by applying cases to create various different status, and checking the related status display windows. (For example, the tester may login two users and have them side-talk. Then he may check the screen of one of these users to see if the status that he/she is "side-talking" with the other user is shown.)
5. Interaction displays (request chat, accept) shall be checked by carrying out actions requiring the appearance of an interaction display window. The tester shall check if the related window appears or not, as well as whether or not the data input into and sent to the other user via this display tool reaches the other end and gets displayed.
6. Installing, quitting, restarting shall be tested by carrying out a number of these activities in the possible locations, as specified by the draft user's manual.
7. Saving, and loading shall be tested by carrying out a number of these activities in the possible locations, as specified by the draft user's manual

CYCLE 3 – Testing of CAIRO with VRML and awareness tool

1. All tests in CYCLES 1 and 2 shall be carried out for reasons of checking the interaction of the awareness tool with the pre-existing components of CAIRO as well as the added VRML environment.
2. Correctness of the data output by the awareness tool shall be checked by comparing the data to real values known through experimentation, previous research papers, or through experience. The test case for the use of the awareness tool shall be obtained by the user's manual submitted by the User Interaction and Awareness Hardware Group.
3. Creating, logging into a forum shall be tester in a similar fashion to the tests carried out during the first term. Different users, including one chairman and various others, shall be created and logged into CARIO. Their interactions with each other, as well as their images, and messages on each other's screens shall be checked.
4. Whiteboard shall be tested in a similar fashion to the tests carried out during the first term. The tester shall operate the whiteboard, use all buttons and functions to draw, edit, load, and save images. The tester also shall check whether or not these modifications to the image are updated on the screens of the other users.
5. Chat – talk, and side-talk shall be tested in a similar fashion to the tests carried out during the first term. The tester shall request the selected talk option from one or all of the users (both as

chairman and as client). Upon acceptance, the tester shall use all buttons and functions to type, edit, load, and save messages. The tester also shall check whether or not these modifications to messages are updated on the screens of the other users.

6. The protocols inside CAIRO shall be checked to see if they apply or not. The test case shall be to join late into all different meeting room types and to accept or reject by the chairman of the meeting. The next case shall be to request side-talk to everyone or to individual users during a meeting and to check for their authorization requests. Different meeting rooms shall be checked by switching from one room to another and checking to see if all of the existing images show up correctly, and if all buttons inside the canvas work correctly. The tester shall also check to see whether or not the agent appears to make suggestions about meeting rooms and the agenda as necessary.

Test Cases

Specific test cases determined for the testing of CAIRO, from the use cases specified in Requirements document v3.0, as well as the original use cases are as given below:

1. A casual contact capability must be provided.
2. Sensing devices must provide a measure of attitude **(or affective state)** of the users.
3. A viewing environment must provide feedback a user about the state of the other users.
4. A message board capability must be provided.
5. A private messaging capability must be provided.
6. A security infrastructure must be provided.

Using the above use cases, the following test cases shall apply:

Test Case #1

1. Chairman wants to create a forum. He/she logs in as the chairman. He/she is asked his/her password and username. He/she inputs an incorrect password. Permission is denied.
2. Chairman wants to create a forum. He/she logs in as the chairman. He/she is asked his/her password and username. He/she inputs the correct password. Permission is granted.
3. Chairman creates a forum for the **round table meeting**.

4. Chairman creates the agenda for the meeting.
5. The image of the chairman exists in the image database. It is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
6. The agent comments about the meeting environment selection.
7. Chairman views the images of other participants, who join the meeting on time. All images are displayed correctly. Every participant has a chair to sit in.
8. Chairman sends a greeting message to all participants, by talking to everyone simultaneously.
9. The chairman loads a previously saved chat from a previous meeting.
10. Chairman starts the meeting. Agenda is displayed and monitored.
11. Emotion (awareness) sensors connected to one of the participants displays his/her emotions. The chairman sees this display. (Both 2D and 3D-VRML cases). In the 2D case, the chairman sees an image representing the emotional state of the participant. In the 3D case, the chairman sees the change in the 3D image of the participant according to the sensed emotion.
12. The chairman side-talks with one of the participants.
13. One of the participants requests a side-talk with the chairman. He/she accepts, and they perform the side-talk operation.
14. One of the participants requests a side-talk with the chairman. He/she rejects.
15. The chairman opens the whiteboard. He/she draws a figure, and then edits it.
16. One of the participants edits the image drawn on the whiteboard. The chairman is able to see the modifications.
17. The chairman saves the image for future reference, and deletes it. He/she then loads a previously saved image. He/she edits the image, and saves it for future reference.
18. A late arriving participant requests permission to enter the meeting. The chairman accepts permission. He/she is successfully added to the meeting environment, and his/her image appears correctly. (Both 2D and 3D cases). The agent comments on the meeting environment selection again.
19. A passing by user requests permission to enter the meeting. (Casual contact).The chairman accepts permission. He/she is successfully added to the meeting environment, and his/her image appears correctly. (Both 2D and 3D cases). The agent comments on the meeting environment selection again.
20. A passing by user requests permission to enter the meeting. (Casual contact). The chairman rejects the request.
21. The chairman offers a change of the meeting environment. They change to **rectangular table meeting**.

22. The chairman offers a change of the meeting environment. They change to **conference hall meeting**.
23. The chairman offers a change of the meeting environment. All but two participants leave the meeting. The remaining change to **personal meeting**.
24. The agenda monitor warns the chairman of time running out.
25. The chairman saves the chat for future reference.
26. The chairman exits CAIRO successfully. (logout)

Test Cases #2,3,4

These test cases are similar to Test Case #1. The only difference is that in these cases, the chairman creates forums in the **rectangular table meeting**, **conference hall meeting**, and **personal meeting** modes respectively.

Test Case #5

1. User wants to join a forum in the form of a **round table meeting**. He/she logs in as a participant. He/she is asked his/her password and username. He/she inputs an incorrect password. Permission is denied.
2. User wants to join a forum in the form of a **round table meeting**. He/she logs in as the chairman. He/she is asked his/her password and username. He/she inputs the correct password. Permission is granted.
3. The forum has not been created, and an appropriate error message is displayed.
4. The forum has been created, and the user logs in successfully.
5. The image of the user exists in the image database. It is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
6. The image of the user does not exist in the image database. He/she makes a request to add an image to the meeting environment of choice. (Both 2D and 3D-VRML cases). This is performed successfully. The selected image is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
7. User views the images of other participants, who join the meeting on time. All images are displayed correctly. Every participant has a chair to sit in.
8. Chairman sends a greeting message to all participants, by talking to everyone simultaneously. User sees the message.

9. The chairman loads a previously saved chat from a previous meeting. User sees this chat session.
10. Chairman starts the meeting. Agenda is displayed and monitored. User can see this.
11. Emotion (awareness) sensors connected to one of the participants displays his/her emotions. The user sees this display. (Both 2D and 3D-VRML cases). In the 2D case, the user sees an image representing the emotional state of the participant. In the 3D case, the user sees the change in the 3D image of the participant according to the sensed emotion.
12. The chairman side-talks with the user.
13. The user requests a side-talk with the chairman. The chairman accepts, and they perform the side-talk operation.
14. The user requests a side-talk with the chairman. The chairman rejects.
15. The chairman opens the whiteboard. He/she draws a figure, and then edits it. The user is able to see it.
16. The user modifies the image on the whiteboard.
17. The chairman saves the image for future reference, and deletes it. He/she then loads a previously saved image. He/she edits the image, and saves it for future reference. The user can see this.
18. A late arriving participant requests permission to enter the meeting. The chairman accepts permission. He/she is successfully added to the meeting environment, and his/her image appears correctly. (Both 2D and 3D cases). The agent comments on the meeting environment selection again. The user sees this.
19. A passing by user requests permission to enter the meeting. (Casual contact).The chairman accepts permission. He/she is successfully added to the meeting environment, and his/her image appears correctly. (Both 2D and 3D cases). The agent comments on the meeting environment selection again. The user sees this.
20. A passing by user requests permission to enter the meeting. (Casual contact). The chairman rejects the request. The user sees this.
21. The chairman offers a change of the meeting environment. The user accepts. They change to **rectangular table meeting**.
22. The chairman offers a change of the meeting environment. The user accepts. They change to **conference hall meeting**.
23. The chairman offers a change of the meeting environment. All but two participants leave the meeting. The user stays. The remaining change to **personal meeting**.
24. The agenda monitor warns the chairman of time running out. The user sees this.

25. The chairman saves the chat for future reference. The user sees this.
26. The user exits CAIRO successfully. (logout)

Test Cases #6,7,8

These test cases are similar to Test Case #5. The only difference is that in these cases, the user joins forums in the **rectangular table meeting**, **conference hall meeting**, and **personal meeting** modes respectively.

Test Case #9

1. User wants to join a forum in the form of a **round table meeting**. He/she logs in successfully.
2. The forum has been created, the user, however, is late.
3. He/she request permission from the chairman of the meeting to join the meeting. The chairman denies permission. A message is displayed indicating the denial of requested permission.
4. He/she request permission from the chairman of the meeting to join the meeting. The chairman gives permission. A message is displayed indicating the acceptance of requested permission. The user joins the meeting.
5. The image of the user exists in the image database. It is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
6. The image of the user does not exist in the image database. He/she makes a request to add an image to the meeting environment of choice. (Both 2D and 3D-VRML cases). This is performed successfully. The selected image is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
7. User views the images of other participants, who join the meeting on time. All images are displayed correctly. Every participant has a chair to sit in.
8. User views the chat box to see the previous conversation.
9. Agenda is displayed and monitored. User can see this.
10. The user requests a side-talk with the chairman. The chairman rejects.
11. The user exits CAIRO successfully. (logout)

Test Cases #10,11,12

These test cases are similar to Test Case #9. The only difference is that in these cases, the user joins forums in the **rectangular table meeting**, **conference hall meeting**, and **personal meeting** modes respectively.

Test Case #13

1. User logs into CAIRO successfully.
2. User starts browsing through existing meetings.
3. User sees a meeting he may want to join.
4. User chooses to join a forum in the form of a **round table meeting**.
5. The forum has been created, the user, however, is not among the invited participants.
6. He/she request permission from the chairman of the meeting to join the meeting. The chairman denies permission. A message is displayed indicating the denial of requested permission.
7. He/she request permission from the chairman of the meeting to join the meeting. The chairman gives permission. A message is displayed indicating the acceptance of requested permission. The user joins the meeting.
8. The image of the user exists in the image database. It is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
9. The image of the user does not exist in the image database. He/she makes a request to add an image to the meeting environment of choice. (Both 2D and 3D-VRML cases). This is performed successfully. The selected image is automatically added to the meeting environment of choice. (Both 2D and 3D-VRML cases).
10. User views the images of other participants, who join the meeting on time. All images are displayed correctly. Every participant has a chair to sit in.
11. User views the chat box to see the previous conversation.
12. Agenda is displayed and monitored. User can see this.
13. The user requests a side-talk with the chairman. The chairman rejects.
14. The user exits CAIRO successfully. (logout)

Test Cases #14,15,16

These test cases are similar to Test Case #13. The only difference is that in these cases, the user joins forums in the **rectangular table meeting**, **conference hall meeting**, and **personal meeting** modes respectively.

Test Case #17

*For use with the VRML environment as a component.

1. User enters the corridor for meeting room selection. All doors are visible, with created meeting names on the doors.
2. User chooses a door. The view closes up to the door by navigating automatically, and enters the selected room. The room is empty. All empty chairs are seen around the meeting table. (Or ordered inside the room for the conference hall case).
3. The view changes to a general view of the selected meeting room, as seen from an approximate 5 to 6 inches height from the floor.
4. The user picks a chair to sit in. The view navigates to the chair, and the chair disappears, to display a view of the table and the other chairs around the table.
5. User looks left, right, up, or down by using corresponding buttons. The view turns in the desired directions.
6. Another participant joins the meeting. His/her avatar is displayed by the door, as standing up.
7. The new participant selects the chair occupied by the user. He/she is denied permission.
8. The new participant selects an empty chair. The avatar moves towards the chair, and sits in the chair. It is then displayed as sitting in the chair.
9. User clicks on whiteboard in the room. The whiteboard tool launches.
10. User clicks on avatar of participant. Pull-down menu appears.

Test Case #18

*For use with the VRML environment as a component.

1. User enters the corridor for meeting room selection. All doors are visible, with created meeting names on the doors.

2. User chooses a door. The view closes up to the door by navigating automatically, and enters the selected room.
3. The view changes to a general view of the selected meeting room, as seen from an approximate 5 to 6 inches height from the floor. All chairs are seen in their appropriate positions. Avatars of existing participants are viewed in their chairs.
4. The user picks a chair to sit in. The chair is already occupied by someone else.
5. The user picks up another chair. The chair is empty. The view navigates to the chair, and the chair disappears, to display a view of the table and the other chairs around the table. All avatars sitting in their chairs are seen.
6. Another participant joins the meeting. His/her avatar is displayed by the door, as standing up.
7. The new participant selects the chair occupied by a user. He/she is denied permission.
8. The new participant selects an empty chair. The avatar moves towards the chair, and sits in the chair. It is then displayed as sitting in the chair.

Test Case #19

*For use with the VRML environment as a component.

1. User enters the corridor for meeting room selection. All doors are visible, with created meeting names on the doors.
2. User chooses a door. The view closes up to the door by navigating automatically, and enters the selected room.
3. The view changes to a general view of the selected meeting room, as seen from an approximate 5 to 6 inches height from the floor. All chairs are seen in their appropriate positions. Avatars of existing participants are viewed in their chairs.
4. The user picks a chair to sit in. The chair is already occupied by someone else.
5. The user picks up another chair. The chair is empty. The view navigates to the chair, and the chair disappears, to display a view of the table and the other chairs around the table. All avatars sitting in their chairs are seen.
6. Another participant joins the meeting. His/her avatar is displayed by the door, as standing up.
7. The new participant selects the chair occupied by a user. He/she is denied permission.
8. Since there are new chairs, the participant (or the chairman, depending on the protocol to be created) requests that another chair be added to the room.

9. A new chair appears near the table, and all other chairs, along with their avatars move sideways to create even spacing with the new chair.
10. The new participant selects the new chair. The avatar moves towards the chair, and sits in the chair. It is then displayed as sitting in the chair.
11. Another new user comes in. He/she requests another chair. The program warns the users that the capacity of the room has been reached and a room change is necessary for the addition of another chair.
12. Chairman chooses to move to another room.
13. All users are viewed as sitting in chairs (the order will be selected either randomly, or by keeping the same order either to the left, or to the right of the chairman). The existing user will see the view from his/her own chair.
14. The new participant is viewed as standing next to the door of the new room.
15. The new participant selects an occupied chair. He/she is denied permission.
16. The new participant selects an empty chair. The avatar moves towards the chair, and sits in the chair. It is then displayed as sitting in the chair.

Test Case #20

1. User leaves the meeting.
2. The meeting room view is replaced by a view of the corridor, with doors and all meeting names written on the corresponding door.
3. User chooses to continue into the next corridor. He/she clicks on the door at the end of the corridor.
4. User is taken to the new corridor, which is identical, except for the names of meetings on the doors. New meetings are visible.
5. User selects a new door, and joins a new meeting.
6. User selects to go back.
7. User goes to the previous corridor by clicking a dedicated button.
8. The previous corridor is seen with all original meeting names. If a meeting has ended, the door is either empty, or carries the name of a new meeting taking place in the room.
9. The user goes through all the corridors. When the next corridor will not have any available meetings, he/she is not let into that corridor, and is warned by a message instead.

7. Schedule for 2nd Term Unit Testing

The general schedule for the testing operations to be carried on in this term is as given below:

1. Testing of VRML Environment; 18 - 21 Feb 1999
2. Submission of the first Tester's Report; 23 Feb 1999
3. Audit of the first Tester's Report; 25 Feb 1999
4. Testing of the connection of VRML to CAIRO; 4 - 10 March 1999
5. Submission of the second Tester's Report; 11 March 1999
6. Audit of the second Tester's Report; 16 March 1999
7. Testing of connection of awareness + VRML to CAIRO, and complete CAIRO: 18 March – 10 April 99
8. Submission of the third Tester's Report; 13 March 1999

8. References

1. Requirements Document 3.0.
2. Quality Assurance Document 2.0.
3. The ANSI/IEEE Std 1008-1987 Standard for Software Unit Testing.
4. DeMillo, McCracken, Martin, Passafiume. *Software Testing and Evaluation.* 1987, Benjamin/Cummins Publishing Company, Inc.
5. B. Chandrasekaran, S. Radicchi. *Computer Program Testing.* 1981, North-Holland Publishing Company.

Appendix E.1 - Glossary of Terms

A

- **Acceptance test.** Formal tests conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept a system. This particular kind of testing is performed with the STW/Regression suite of tools.
- **Ada.** The DoD standard programming language.
- **ATS.** A SMARTS user-designed description file which references a test suite. Test cases are referenced in a hierarchically organized structure and can be supplemented with activation commands comparison arguments, pass/fail evaluation criteria, and system commands. When SMARTS is run on either a X Window or UNIX system, the ATS is written in SMARTS' Description Language (which is similar to C language syntax). The ATS file is written in SMARTS C-Interpreter Language when SMARTS is run on a MS Windows system.
- **AUT.** Application-under-test.
- **Automated Test Script.** See ATS.

B

- **Back-to-back testing.** For software subject to parallel implementation, back-to-back testing is the execution of a test on the similar implementations and comparing the results.
- **Basis paths.** The set of non-iterative paths.
- **Black-Box testing.** A test method where the tester views the program as a black box, that is the test is completely unconcerned about the internal behavior and structure of the program. Rather the tester is only interested in finding circumstances in which the program does not behave according to its specifications. Test data are derived solely from the specifications without taking advantage of knowledge of the internal structure of the program. Black-box testing is performed with the STW/Regression suite of tools.
- **Bottom-up testing.** Testing starts with lower level units. Driver units must be created for units not yet completed, each time a new higher level unit is added to those already tested. Again a set of units may be added to the software system at the time, and for enhancements the software system may be complete before the bottom up tests starts. The test plan must reflect the approach, though. The STW/Coverage suite of tools supports this type of testing.

- **Built-in testing.** Any hardware or software device which is part of an equipment, subsystem of system and which is used for the purpose of testing that equipment, subsystem or system.

C

- **C.** The programming language C. ANSI standard and K&R C are normally grouped as one language. Certain extensions supported by popular C compilers are also included as normal C.
- **C++.** The C++ object oriented programming language. The current standard is ANSI C++ and/or AT&T C++. Both are supported by TCAT/C++.
- **C0 coverage.** The percentage of the total number of statements in a module that are exercised, divided by the total number of statements present in the module.
- **C1 coverage.** The percentage of logical branches exercised in a test as compared with the total number of logical branches known in a program.
- **Call graph.** The function call tree capability of S-TCAT. This utility show caller-callee relationship of a program. It helps the user to determine which function calls need to be tested further.
- **Coding rule.** A rule that specifies a particular way in which a program is to be expressed.
- **Coding style.** A general measure of the programming nature of a system; abstractly, the way the programming language is used in a real system.
- **Compiler-based testing.** is carried out through the source language compiler, either as an additional compilation activity, or to implement a language feature designed to assist in the detection of errors.
- **Complexity.** A relative measurement of the "degree of internal complexity" of a software system, expressed possibly in terms of some algorithmic complexity measure.
- **Component.** A part of a software system smaller than the entire system but larger than an element.
- **Connected directed graph.** A directed graph is connected if there is at least one path from every entry node to every exit node.
- **Control statement.** A statement that involves some predicate operation. For example: an if statement or a while statement.
- **Correctness proof.** A mathematical process which demonstrates the consistency between a set of assertions about a program and the properties of the program, when executed in a known environment.

- **Coverage testing.** Coverage testing is concerned with the degree to which test cases exercise or cover the logic (source code) of the software module or unit. It is also a measure of coverage of code lines, code branches and code branch combinations.
- **Cross-reference.** An indication, for a selected symbol, of where instances of that symbol lie in a software system.
- **Cycle.** A sequence of logical branches that forms a closed loop, so that at least one node is repeated.

D

- **DD-path.** See Logical branch.
- **De-instrumentation.** When certain parts of your code have already been tested, you can use TCAT's and S-TCAT's de-instrumentations utilities to exclude those parts from instrumentation. For large programs, this can save time.
- **Debug.** After testing has identified a defect, one "debugs" the software by making certain changes that repair the defect.
- **Decision node.** A node in the program directed graph which corresponds to a decision statement within the program.
- **Decision statement.** A decision statement in a module is one in which an evaluation of some predicate is made, which (potentially) affects the subsequent execution behavior of the module.
- **Decision-to-decision path.** See Logical branch.
- **Defect.** Any difference between program specifications and actual program behavior of any kind, whether critical or not. What is reported as causing any kind of software problem.
- **Deficiency.** See Defect.
- **Delay multiplier.** The multiplier used to expand or contract playback rates.
- **Directed graph.** A directed graph consists of a set of nodes, which are interconnected with oriented arcs. An arbitrary directed graph may have many entry nodes and many exit nodes. A program directed graph has only one entry and one exit node.
- **Domain testing.** is designed specifically to detect domain errors. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program.

- **Dynamic analysis.** A process of systematically demonstrating properties of programs by a series of constructed executions. The STW/Coverage suite of tools performs dynamic analysis.
- **Dynamic directed graph display.** An organic diagram showing the connection between logical branches in a program, where the logical branches are "animated" based on behavior of the instrumented program being tested.

E

- **Edge.** In a directed graph, the oriented connection between two nodes.
- **End-to-end testing.** Test activity aimed at proving the correct implementation of a required function at a level where the entire hardware/software chain involved in the execution of the function is available.
- **Error.** A difference between program behavior and specification that renders the program results unacceptable. See Defect.
- **Essential paths.** The set of paths that include one essential edge, that is an edge that lies on no other path.
- **Executable statement.** A statement in a module which is executable in the sense that it produces object code instructions. A non-executable statement is not the opposite: it may be a declaration. Only comments can be left out without affective program behavior.
- **Execution verifier.** A system to analyze the execution-time behavior of a test object in terms of the level of testing coverage attained.
- **Explicit predicate.** A program predicate whose formula is displayed explicitly in the program text. For example: a single conditional always involves an explicit program predicate. A predicate is implicit when it is not visible in the source code of the program. An example is a program exception, which can occur at any time.

F

- **Filter.** A stage in a software process that attempts to identify defects so they can be removed.
- **Filter Efficiency.** The percentage of state-detectable defects vs. the actual average number of defects detected. Typical filter efficiencies range from 10% (not often of much practical use) to 90% (nothing is perfect)>

- **Flow control.** When a terminal emulation program establishes communications with a mainframe application, it establishes flow control to prevent characters being lost. In some cases the mainframe application (or cluster controller) locks out the keyboard. This prevents the user from typing ahead; however, when CAPBAK is being used to record terminal sessions, the user is expected to wait for a response from the mainframe. The user, thus, imposes manual flow control to prevent data from being lost in cases where the keyboard is not locked. When CAPBAK is being run in terminal emulation mode, a record of the manual flow control is stored in the keysave and response files. When CAPBAK is transmitting keys in playback, flow control is maintained by using the information saved in these files. See also Automatic flow control.
- **Flow graph.** The oriented diagram, composed with nodes and edges with arrows, the shows the flow of control in a program. Also called a flow chart or a directed graph.
- **Function call.** A reference by one program to another through the use of an independent procedure-call or functional-call method. Each function call is the "tail" of a caller-callee call-pair.
- **Function Keys.** During a recording session with CAPBAK/X or CAPBAK/MSW, you can issue commands via function keys (i.e. your F1 to F10 keyboard function keys). During a recording session, you can use the function keys to bring up the Hotkey window (see Hotkeywindow), add comments to the keysave file, select an image or window for or the entire screen, pause, resume or terminate the session. CAPBAK/X also has additional function keys that allow you to synchronize on a character string or extract characters from an image or the entire screen. During playback, function keys can be used to slow or to quicken the speed of playback, to insert or to append new keysave records into a keysave file, to pause, to resume or to terminate a playback session.
- **Functional specifications.** A set of behavioral and performance requirements which, in aggregate, determine the functional properties of a software system.
- **Functional test cases.** A set of test case data sets for software which are derived from structural test cases.

G

- **GUI (Graphical User Interface).** A interface system, e.g. X11 or Windows '95 that communicates between a user and an application.

H

- **History report.** A SMARTS summary report which indicates all test outcomes maintained in the log file, providing an overview of test regression throughout the testing process.
- **Hotkey window.** When recording a test session with CAPBAK/X or CAPBAK/MSW, this window pops up when the hotkey function key is pressed (defaulted to F1). It allows you to issue commands, including inserting comments into the keysave file, to save an image or window for synchronization, to capture an image, a window, or the entire screen, to resume, and to end a recording session.

I

- **In-degree.** In a directed graph, the number of in-ways for a node.
- **Incompatible logical branch.** Two segments in one program are said to be incompatible if there is no logically feasible execution of the program, which will permit both of them to be executed in the same test. See also Essential logical branch.
- **Infeasible path.** A logical branch sequence is logically impossible if there is no collection of setting of the input space relative to the first branch in the sequence, which permits the sequence to execute.
- **Inspection/review.** A process of systematically studying and inspecting programs in order to identify certain types of errors, usually accomplished by human rather than mechanical means.
- **Instrumentation.** The first step in analyzing test coverage, is to instrument the source code. Instrumentation modifies the source code so that special markers are positioned at every logical branch or call-pair or path. Later, during program execution of the instrumented source code, these markers will be tracked and counted to provide data for coverage reports.
- **Integration Testing.** Exposes faults during the process of integration of software components or software units and it is specifically aimed at exposing faults in their interactions. The integration approach could be either bottom-up (using drivers), top-down (using stubs) or a mixture of the two. The bottom up is the recommended approach.
- **Interface.** The informational boundary between two software systems, software system components, elements, or modules.

- **Invocation point.** The invocation point of a module is normally the first statement in the module.
- **Invocation structure.** The tree-like hierarchy that contains a link for invocation of one module by another within a software system.
- **Iteration level.** The level of iteration relative to the invocation of a module. A zero-level iteration characterizes flows with no iteration. A one-level iteration characterizes program flow, which involves repetition of a zero-level flow.

J

- **Java.** A programming language, not unlike C++, that is used to program applets that are interpretively executed by Java applet viewers associated with several Internet browsers. Some say that Java := ((C++)--)++.

L

- **Linear histogram.** A dynamically updated linear-style histogram showing accumulating C1 or S1 coverage for a selected module.
- **Log file.** An established or default SMARTS file where all test information is automatically accumulated.
- **Logical branch.** The set of statements in a module which are executed as the result of the evaluation of some predicate (conditional) within the module. The logical branch should be thought of as including the outcome of a conditional operation and the subsequent statement execution up to and including the computation of the value of the next predicate, but not including its evaluation in determining program flow.
- **Logical units.** A logical unit is a concept used for synchronization when differencing two files with the EXDIFF system. A logical unit may be a line of text, a page of text, a CAPBAK screen dump, or the keys (or responses) between marker records in a keysave file.
- **Loop.** A sequence of logical branches in a program that repeats at least one node. See Cycle.

M

- **Manual analysis.** The process of analyzing a program for conformance to in-house rules of style, format, and content as well as for correctly producing the anticipated output and results. This process is sometimes called code inspection, structured review, o formal inspection.
- **METRIC.** The software metrics processor/generator component of STW/Advisor. METRIC computes several software measures to help you determine the complexity properties of your software.
- **Module.** A module is a separately invocable element of a software system. Similar terms are procedure, function, or program.
- **Multi-unit test.** A multi-unit test consists of a unit test of a single module in the presence of other modules. It includes: (1) a collection of settings for the input space of the module and all the other modules invoked by it and (2) precisely one invocation of the module under test.

N

- **Node.** (1) A position in a program assigned to represent a particular state in the execution space of that program. (2) Group or test case in a test tree.
- **Node number.** A unique node number assigned at various critical places within each module. The node number is used to describe potential and/or actual program flow.
- **Non-executable statement.** A declaration or directive within a module which does not produce (during compilation) object code instructions directly.

P

- **Path, path class.** An ordered sequence of logical branches representing one or more categories of program flow.
- **Path predicate.** The predicate that describes the legal condition under which a particular sequence of logical branches will be executed.
- **Past test report.** This report lists information from the stored archive file for TCAT and S-TCAT. It summarizes the percentage of logical branches/call-pairs hit in each module listed, giving the C1/S1 value for each module and the program as a whole.
- **Predecessor logical branches.** One of many logical branches that precede a specified logical branch in normal (structurally implied) program flow.
- **Predicate.** A logical formula involving variables/constants known to a module.

- **Process.** The sequence of steps that are performed in developing a software product, system, upgrade, etc. Software processes involve multiple stages, multiple types of activities, many of which may be concurrent.
- **Program.** See Module.
- **Program directed graph.** See Directed graph.
- **Program mutation.** is based on the idea that professional programmers usually produce code, which is very close to being correct. The goal is to establish a program, which is either correct, or radically incorrect (as opposed to simple errors). A program, which is almost correct, is called a mutant of the correct program.
- **Program predicate.** See Predicate.
- **Pseudo code.** A form of software design in which programming actions are described in a program-like structure; not necessarily executable but generally held to be humanly readable.

Q

- **Qualification.** The process that assures that a given software component at the end of its development is compliant with the requirements. The qualification shall be performed at appropriate and defined software components and sub software systems, before integrating the software to the next higher level. The techniques for qualification is testing, inspection and reviewing.

R

- **Regression Testing.** Testing which is performed after making a functional improvement or repair of the software. Its purpose is to determine if the change has regressed other aspects of the software. As a general principle, software unit tests are fully repeated if a module is modified, and additional tests which expose the fault removed, are added to the test set. The software unit will then be re-integrated and integration testing repeated.
- **Return variable.** A return variable is an actual or formal parameter for a module, which is modified within the module.

S

- **Segment.** A [logical branch] segment or decision-to-decision path is the set of statements in a module which are executed as the result of the evaluation of some predicate (conditional)

within the module. The segment should be thought of as including the outcome of a conditional operation and the subsequent statement execution up to and including the computation of the value of the next predicate, but not including its evaluation in determining program flow.

- **Segment instrumentation.** The process of producing an altered version of a module which is logically equivalent to the unmodified module but which contains calls to a special data collection subroutine which accepts information as to the specific logical branch sequence incurred in an invocation of the module.
- **Software subsystem.** A part of a software system, but one which includes many modules. Intermediate between module and system.
- **Software system.** A collection of modules, possibly organized into components and subsystems, which solves some problem or performs some task.
- **Spaghetti code.** A program whose control structure is so entangled by a surfeit of GOTO's that is flow graph resembles a bowl of spaghetti.
- **Statement complexity.** A complexity value assigned to each statement which is based on (1) the statement type, and (2) the total length of postfix representations of expressions within the statement (if any). The statement complexity values are intended to represent an approximation to potential execution time.
- **Static analysis.** The process of analyzing a program without executing it. This may involve wide range of analyses. The STW/Advisor suite of tools performs static analyses.
- **Status report.** This SMARTS' report presents the most recent information about executed tests. It contains: test case name, outcome (pass/fail), activation date, execution time (seconds), and error number.
- **Successor logical branch.** One or more logical branches that (structurally) follow a given logical branch.
- **Symbolic Execution (Evolution) Tree.** It is used to characterize the execution of a program for the symbolic testing of that program. It consists of nodes associated with the executed statements, as well as directed arcs, which indicate the direction of program flow.
- **System Testing.** Verifies that the total software system satisfies all of its functional, quality attribute and operational requirements in simulated or real hardware environment. It primarily demonstrates that the software system does fulfill requirements specified in the requirements specification during exposure to the anticipated environmental conditions. All testing objectives relevant to specific requirements should be included during the software system

testing. Software system testing is mainly based on black-box methods. The STW/Coverage suite of tools supports this type of testing.

T

- **Test.** A [unit] test of a single module consists of (1) a collection of settings for the inputs of the module, and (2) exactly one invocation of the module. A unit test may or may not include the effect of other modules, which are invoked by the undergoing testing.
- **Test adequacy.** It refers to the ability of the data to insure that certain errors are not present in the program under test. A test data set is adequate if the program runs successfully on the data set and if all incorrect programs run incorrectly.
- **Testbed.** See Test Harness.
- **Test coverage ceasure.** A measure of the testing coverage achieved as the result of one unit test usually expressed as a percentage of the number logical branches within a module traversed in the test.
- **Test data set.** A specific set of values for variables in the communication space of a module which are used in a test.
- **Test harness.** A tool that supports automated testing of a module or small group of modules.
- **Test object, object under test.** The central object on which testing attention is focused.
- **Test path.** A test path is a specific (sequence) set of logical branches which is traversed as the result of a unit test operation on a set of test case data. A module can have many test paths.
- **Test purpose.** The free-text description indicating the objective of a test, which is usually specified in the source clause of a SMARTS ATS file.
- **Test stub.** A testing stub is a module, which simulates the operations of a module, which is invoked within a test. The testing stub can replace the real module for testing purposes.
- **Test target.** The current module (system testing) or the current logical branch (unit testing) upon which testing effort is focused.
- **Test target selector.** A function which identifies a recommended next testing target.
- **Testability.** A design characteristic which allows the status (operable, inoperable, or degrade) of a system of any of its sub-system to be confidently determined in a timely fashion. Testability attempts to qualify those attributes of system designs which facilitate detection and isolation of faults that affect system performance. Testability can be defined as the

characteristic of a design which allows the status of a system of any of its subsystems to be confidently determined in a timely fashion.

- **Testing.** Testing is the execution of a system in a real or simulated environment with the intent of finding faults.
- **Testing Techniques.** Can be used in order to obtain a structured and efficient testing which covers the testing objectives during the different phases in the software life cycle.
- **Top-Down Testing.** The testing starts with the main program. The main program becomes the test harness and the subordinated units are added as they are completed and testing continues. Stubs must be created for units not yet completed. This type of testing results in retesting of higher level units when more lower level units are added. The adding of new units one by one should not be taken too literary. Sometimes a collection of units will be included simultaneously, and the whole set of units will serve as test harness for each unit test. Each unit is tested according to a unit test plan, with a top-down strategy.

U

- **Unit test.** See Test.
- **Unit Testing.** Unit testing is meant to expose faults on each software unit as soon as this is available regardless of its interaction with other units. The unit is exercised against its detailed design and by ensuring that a defined logic coverage is performed. Informal tests on module level which will be done by the software development team and are informal tests which are necessary to check that the coded software modules reflect the requirements and design for that module. White-box oriented testing in combination with at least one black box method are used.
- **Unreachability.** A statement (or logical branch) is unreachable if there is no logically obtainable set of input-space settings which can cause the statement (or logical branch) to be traversed.

V

- **Validation.** The process of evaluation software at the end of the software development process to ensure compliance with software requirements. The techniques for validation is testing, inspection and reviewing.

- **Verification.** The process of determining whether of not the products of a given phase of the software development cycle meet the implementation steps and can be traced to the incoming objectives established during the previous phase. The techniques for verification are testing, inspection and reviewing.

W

- **White-box testing.** A test method where the tester views the internal behavior and structure of the program. The testing strategy permits one to examine the internal structure of the program. In using this strategy, the tester derives test data from an examination of the program's logic without neglecting the requirements in the specification. The goal of this test method is to achieve a high test coverage, that is examination of as much of the statements, branches, paths as possible.

Bibliography

American National Standards Institute (ANSI) homepage. (http://www.ansi.org) Last updated: May 1999.

The ANSI/IEEE Std 1008-1987 Standard for Software Unit Testing. American National Standards Institute.

B.W. Boehm. *Software Engineering*, IEEE Trans. On Computers, C-25. 12 Dec 1976.

B. Chandrasekaran, S. Radicchi. *Computer Program Testing*. 1981, North-Holland Publishing Company. Amsterdam, Holland.

Computing History web page, IEEE Computer Society. (http://www.computer.org/50/history). Copyright © 1997.

G. Cruz. DiSEL-98 Team *Quality Assurance Document* Version 2.0. MIT, February 1999.

D. Deborah, N. T. Snyder. Strategies, Tools, and Techniques that Succeed. *Mastering Virtual Teams*. 1999, Jossey-Bass Publishers. San Francisco, USA.

DefenseLINK homepage – Official site for USA Department of Defense, USA Department of Defense. (http://www.defenselink.mil) Last updated: May 1999.

DeMillo, McCracken, Martin, Passafiume. *Software Testing and Evaluation*. 1987, Benjamin/Cummins Publishing Company, Inc. Menlo Park, California, USA.

Establishment of Software Engineering as a Profession web page, IEEE Computer Society. (http://www.computer.org/tab/seprof/history.htm). Last updated: Feb 1999.

General Principles of Software Validation, General Draft Version 1.1, Food & Drug Administration, Center for Devices and Radiological Health, Guidance for Industry web page, FDA. (http://www.fda.gov/cdrh/comp/swareval.html). Last updated: June 1997.

M. Haywood. Practical Techniques for High-Technology Project Managers. *Managing Virtual Teams*. 1998, Management Strategies, Inc. Boston, USA.

J. Hor, C. Kuyumcu. DiSEL-98 Testing Report Version 1.0. MIT, 11/17/1998.

C. S. Horstmann, G. Cornell. *Core Java*, Volume I – Fundamentals. 1997, Sun Microsystems Press, Inc. California, USA.

Infotech, *Software Testing*, Volume 1: Analysis and Bibliography. 1979, Infotech International Ltd. Oakland, California, USA.

Infotech, *Software Testing*, Volume 2: Invited Papers. 1979, Infotech International Ltd. Oakland, California, USA.

Institute of Electrical and Electronics Engineers (IEEE) homepage. (http://www.ieee.org). Last updated: April 1999.

Institute of Electrical and Electronics Engineers (IEEE) Computer Society homepage. (http://www.computer.org). Last updated: March 1999.

International Federation of Information Processing (IFIP) homepage. (http://www.ifip.or.at). Last updated: April 1999.

International Society for Measurement and Control (ISA) homepage. (http://www.isa.org/index). Last updated: May 1999.

IV&V (Independent Verification and Validation) index page, NASA. (http://www.ivv.nasa.gov/services/ivv/index.html) Last updated: Nov 1998.

H. Kopetz, *Software Reliability*, Springer-Verlag, NY, 1979.

H. Krawczyk, B. Wiszniewski. *Analysis and Testing of Distributed Software Applications*. 1998, Research Studies Press, Inc. Baldock, Hertfordshire, England.

C. Kuyumcu. DiSEL-98 Testing Report Version 1.1. MIT, 03/16/1999.

G. Landel, S. Vadahvkar. DiSEL-98 Team *Software Requirements Document* Version 3.0. MIT, February 1999.

D. Lumsden, G. Lumsden. Communicating in Groups and Teams, *Sharing Leadership*. 1993, Wadsworth Publishing Company, Inc., Belmont, California.

C. Manasseh, *CDI – Software Project Management Plan*, Version 2.3. MIT, March 1999.

E. F. Miller. *Program testing technology in the 1980's*. Proc IEEE Conf on Problems in Computing in the 1980s Oregon. April 1978.

National Aeronautics and Space Administration (NASA) homepage. (http://www.nasa.gov). Last updated: April 1999.

NASA Ames Research Center homepage, NASA. (http://ivv.nasa.gov). Last updated: Nov 1998.

Online Standards Resources web page, IEEE Computer Society. (http://www.computer.org/standard/othstand.htm). Copyright © 1997.

Pentium III Xeon web page, Intel Corporation. (http://www.intel.com/PentiumIII/Xeon/home.htm). Last updated: Feb 1999.

Processor Hall of Fame web page, Intel Corporation. (http://www.intel.com/intel/museum/25anniv/hof/hof_main.htm). Last updated: Feb 1999.

Science Program web page, NATO. (http://www.nato.int/science/index.html). Last updated: May 1999.

Single Stock Point for Military Specifications, Standards, and Related Publications web page, USA Department of Defense. (http://www.dodssp.daps.mil). Last updated: Jan 1999.

United Nations Educational, Scientific, and Cultural Organization (UNESCO) homepage. (http://www.unesco.org). Last updated: April 1999.

B.Yang. M.Eng Thesis, *Managing a Distributed Software Engineering Team.* The DiSEL-98 Project. MIT, May 1998. © Bob Yang, MCMXCVIII.