SITE: A Statistics–based Integrated Test Environment

Huey–Der Chu

Centre for Software Reliability, Dept. of Computing Science
Bedson Building, University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, U. K.

Tel: +(44) 191 222 8972
Fax:+(44) 191 222 8788
Email: huey–der.chu@newcastle.ac.uk
URL: http://www.ncl.ac.uk/~n4521677

John E Dobson

Centre for Software Reliability, Dept. of Computing Science
Bedson Building, University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, U.K.

Tel: +(44) 191 222 8228
Fax: +(44) 191 222 8788
Email: john.dobson@newcastle.ac.uk

SITE: A Statistics–based Integrated Test Environment

Huey–Der Chu, John E Dobson

Centre for Software Reliability, Department of Computing Science,

University of Newcastle upon Tyne, NE1 7RU, UK

ABSTRACT

The production of high–quality information applications to secure the reliability of Software is a very important issue in the development of information systems. An essential component for developing quality software is software testing. However, it is a very time–consuming and tedious activity and accounts for over 30% of the cost. In addition to its high cost, manual testing is unpopular and often inconsistently executed. Therefore, a powerful environment that automates testing and analysis techniques is needed. This paper presents a Statistics–based Integrated Test Environment, SITE, which supports statistics–based testing on the top of specification–based testing and addresses the two main issues in software testing, when to stop testing and how good the software is after testing. It provides automatic support for test execution by the test driver, test development by the SIAD/SOAD tree editor and the test data generator, test failure analysis by the test results validator, test measurement by the statistical analyst, test management by the test manager and test planning by the modeller. These tools are integrated around an object management system which includes a public, shared data model describing the data entities and relationships which are manipulable by these tools.

Keywords: Software test environment, deterministic testing, statistical testing, software quality

1 INTRODUCTION

The history of software testing is as long as the history of software development itself. It is an integral part of the software life–cycle and must be structured according to the type of product, environment and language used. The goal of software testing is [14, 17]: Firstly, to reveal that hidden number of defects which are created during the specification, design and coding stages of development, secondly, to provide confidence that failures do not occur and thirdly, to reduce the cost of software failure over the life of a product. This is not only a developmental activity for discovering product defects but also an independent assessment of software execution in an operating environment. It is a very time–consuming and tedious activity and accounts for over 30% of the cost of software development [1, 11, 15]. In addition to its high cost, manual testing is unpopular and often inconsistently executed. Therefore, a powerful environment that automates sophisticated testing and analysis techniques is needed.

Software Test Environments (STE) overcome the deficiencies of manual testing through automating the test process and integrating testing tools to support a wide range of test capabilities [9]. The use of STE provides significant benefits as follows [16, 18]. Firstly, major productivity enhancements can be achieved by automating techniques through tool development and use. Secondly, errors made in testing activities can be reduced through formalizing the methods used. Thirdly, defining testing processes secures more accurate, more complete and more consistent testing than do human–intensive, ad hoc testing processes. Fourthly, automated testing improves the likelihood that results can be reliably reproduced.

To achieve the high quality required of software, a Statistics–based Integrated Test Environment (SITE) is proposed. Based on two main issues in software testing, when to stop testing and how good is the software after testing, it provides support for test execution, test development, test failure analysis, test measurement, test management and test planning. SITE enables early entry of the test process into the life cycle due to the modelling stage. After well–prepared modelling and requirements specification are undertaken, the test process and the software design and implementation can proceed concurrently.

In Section 2 of this paper, we survey the related work in software testing techniques, particularly in statistical software testing. The architecture of SITE for supporting automatic testing is proposed in Section 3. In Section 4, a comparison of STEs using the Software Architecture Analysis Method (SAAM) is discussed. Section 5 summarizes my research.

2 SOFTWARE TESTING TECHNIQUES

2.1 The Classification Of Software Testing Techniques

The software testing techniques can be classified according to the following viewpoints and the classification of software testing techniques is shown in Figure 1.

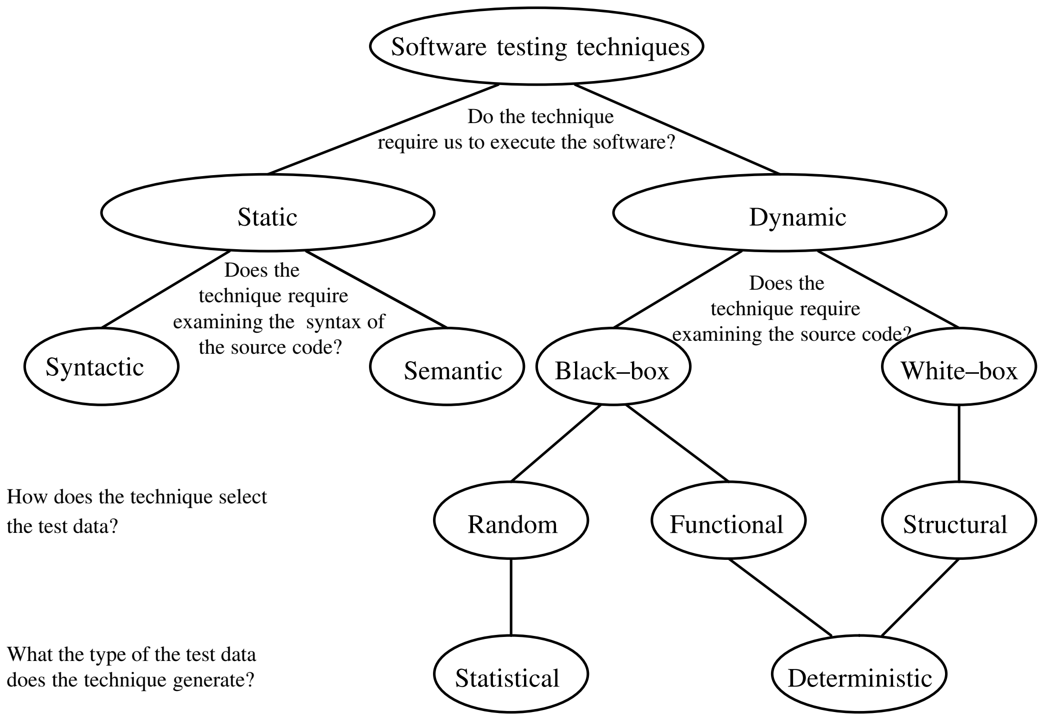


Figure 1: The classification of software testing techniques

• Does the technique require us to execute the software? If so, the technique is *dynamic testing*; if not, the technique is *static testing*.

• Does the technique require examining the source code in dynamic testing? If so, the technique is *white–box testing*; if not, the technique is *black–box testing*.

• Does the technique require examining the syntax of the source in static testing? If so, the technique is *syntactic testing*; if not, the technique is *semantic testing*.

• How does the technique select the test data? Test data is selected depending on whether the technique refers to the function or the structure of the software [3], leading respectively to *functional testing* and *structural testing*, where as test data is selected according to the way in which software is operated with respect to *random testing*.

• What type of test data does the technique generate? In *deterministic testing* [13], test data are predetermined by a selective choice according to the adopted criteria. In *statistical testing*, test data are generated according to a defined probability distributed on the input domain.

In practice, the software development methodologies typically employ a combination of several software testing techniques. That there is no “silver bullet” testing approach and that no single technique is satisfactory on its own has been pointed out by many leading researchers. The need to combine testing techniques is further substantiated when we consider the primary characteristics of each approach and find that each testing strategy addresses only a narrow set of concerns. We must also ensure that we do each of them at the right time on the right work products.

2.2 Statistical Software Testing

It is a well known fact in the software industry that software of any complexity cannot be exhaustively tested and that a sample of the possible inputs must be relied on for the testing performed. The testing techniques which base on the deterministic method ask the tester to select these peculiar inputs to test peculiar cases by means of test criteria. It may discover many errors but may not provide much improvement in the product’s quality. It is also accepted that errors can have significantly different effects on the failure rate of software and that a greater payoff comes from discovering and removing the errors with high failure rates during testing.

Statistical testing is a software testing process in which the objective is to measure the reliability of the software rather than to discover software faults. The user of a system is interested in the probability of failure–free behaviour. Following this line of thought, statistical testing which has a high fault–revealing power provides a method for determining test data sets, in spite of a tricky link between the adopted criterion and the actual faults [13, 17]. The exit criterion can be based on a reliability measure when the test set has been selected randomly from an appropriate probability distribution over the input domain.

The basic procedure of the statistical testing shown in Figure 2 is

- to determine the size n of test set and select test cases from an input distribution,
- to execute the software under test,
- to record the amount of execution time between failure [7] or estimate the defective rate of the output population [4],
- to continue testing until the selected model shows that the required failure–intensity level has been met to the desired level of confidence.

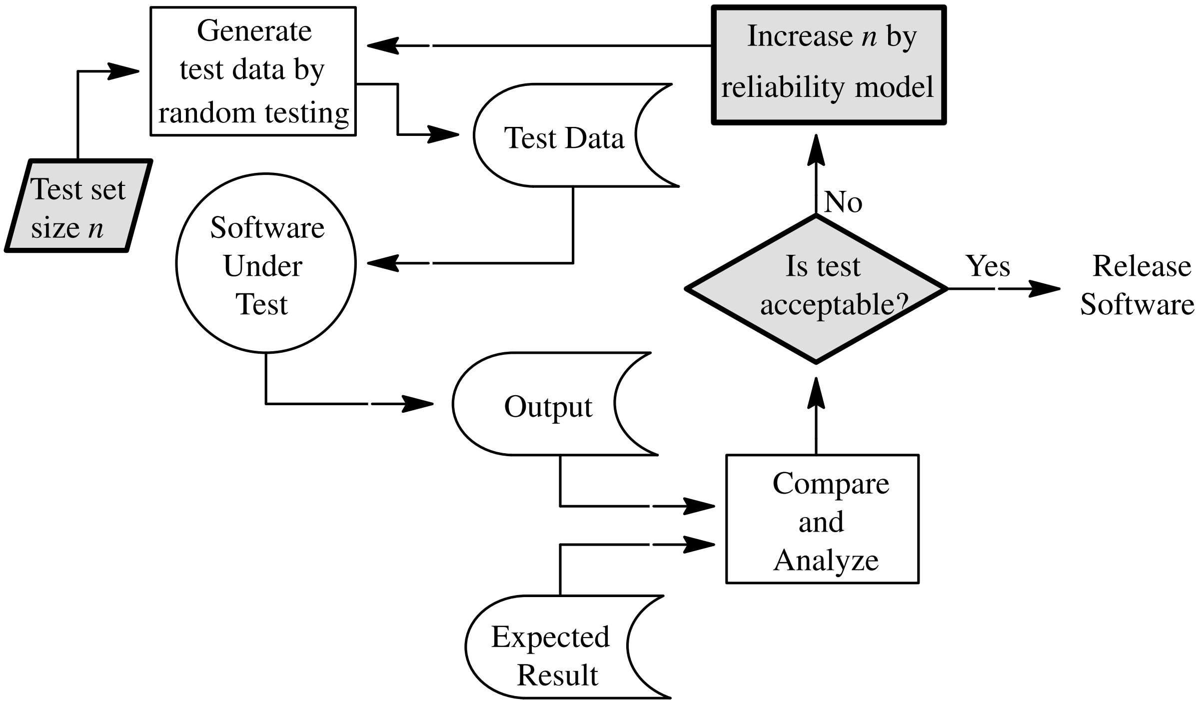


Figure 2: A model of the exit criterion for the statistical testing

Current statistical testing techniques involve exercising a piece of software by supplying it with test data that are randomly drawn according to a single, unconditional probability distribution on the software's input domain [6, 8]. This distribution represents the best estimate of the operational frequency for the use for each input. It provides a scientific basis for making inferences from testing about operational environment. Therefore, if test data are randomly drawn from an input distribution representative of some particular user profile, statistical testing becomes an experimental way to determine whether or not a product meets its dependability requirements. The main benefit of statistical testing is that it allows the use of statistical inference to compute probabilistic aspects of the results of the testing process, such as reliability, *mean time to failure* (*MTTF*) and *mean time between failures* (*MTBF*). However, these techniques are insufficient for many types of software, because the probability of applying an input can change when the software is executed [7].

In order to address this difficulty, we introduced the inverse concept presented by Cho [4]. Each execution of the software is considered equivalent to 'sampling' an output from the output population. The goal of software testing is to find certain characteristics of the population such as the ratio of the number of defective outputs in the population to the total number of outputs in the population. It uses the number of executions of the software to assess the software reliability, which is different from previously mentioned measures which use the execution time. Gathering the data for the error history of a piece of software requires a long period of time, and even then, the reliability measure is often difficult to quantify. However, a piece of software is not subject to deterioration such as wear, tear or burn, that is, the reliability of a piece of software is independent of time but dependent on the frequency and nature of software usage. Cho gives the following definition:

Software reliability is $1 - \theta$, the probability that the software performs successfully, according to software requirements, independent of time.

where θ is the defective rate of the software output population. This definition is a natural consequence of following the principles of software engineering with statistical quality control. From this point of view, determining the defective or non–defective outputs from software requires corresponding input data. The input domain is the source from which input data are constructed for the software. If the input domain is not well defined, the input data will not be properly constructed and will be of a poor quality. Following Cho, the approach adopted in this paper specifies the input domain of a software by means of a SIAD (Symbolic Input Attribute Decomposition) tree which is a syntactic structure representing the input domain of a piece of software in a form that facilitates construction of random test data for producing random output for quality inspection. The SIAD tree enforces the development of well–defined user requirements and imposes discipline in both design and implementation. However, it lacks a clear framework which would indicate how automated testing is to be achieved. As an improvement, a Framework for Automating Statistics–based Testing (FAST) was presented.

2.3 An Overview of the FAST

FAST [5], which is an extension of the testing concept in quality programming, has been proposed to help develop good quality and cost effective software. It addresses the two major software testing issues: when to stop testing and how good the software is after testing. A simplified view of the FAST is shown in Figure 3.

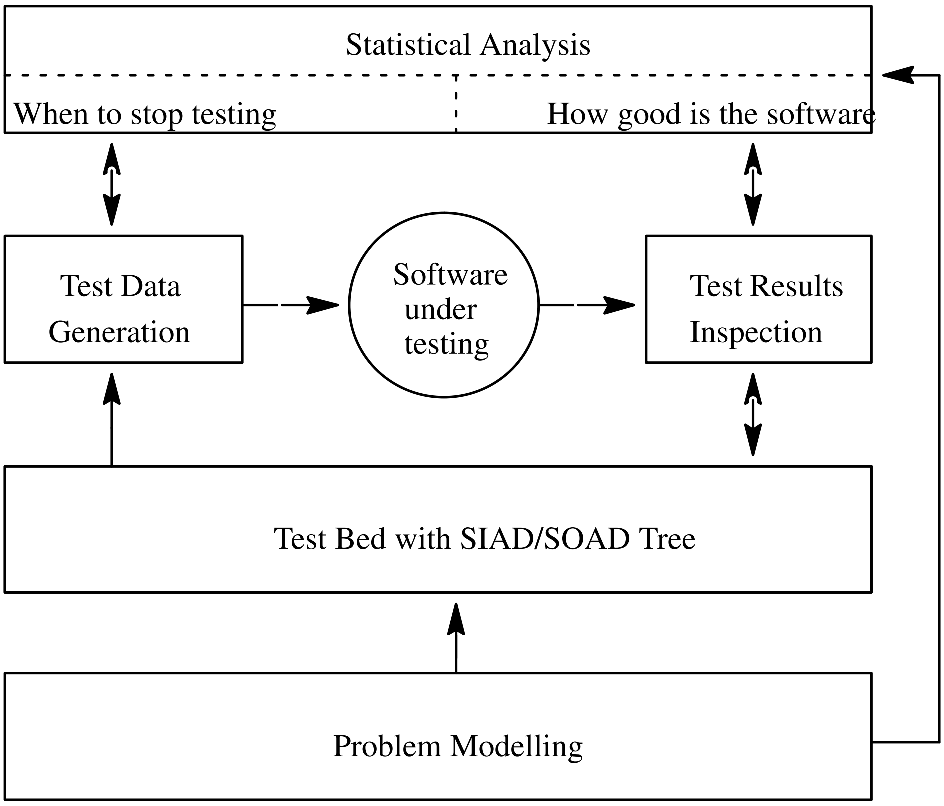


Figure 3: An overview of the FAST

• Problem Modelling: Given a system to be developed, a model is developed to analyze and understand the problem. The Modelling activity includes: modelling of inputs and outputs as well as the modelling of the software. Requirements are then generated as a result of the modelling activity. Included in the requirements are software and test requirements. Software requirements define the functions the software is to perform and the quality characteristics such as response time, throughput, understandability and portability. The test requirement can be divided into two groups: functional and quality requirements. These requirements define the input domain, product units and product unit defectiveness for statistical sampling, sampling methods for estimating the defective rate of the software population with which to judge software quality, statistical inference methods and the confidence levels of software output population quality, the acceptable software defect rate and the generation methods of test input units. From functional requirements, the input domain and the product units are defined. From quality requirements, we can define the product unit defectiveness and specify the quality statement. Without the definitions of input domain, product unit and product unit defectiveness, it is not possible to control the quality of the data produced by a piece of software.

• The SIAD/SOAD tree: One of the major problems in software development is ambiguity in requirements specification, in particular the specification of input domain and product unit. The SIAD tree is a syntax structure representing the input domain of a piece of software in a form that facilitates construction of random test data for producing random output for quality inspection. It is used to represent the hierarchical and "network" relationship between input elements and to incorporate rules into the tree for using the inputs. Input is constructed from data of different characteristics that are called input attributes. Associated with each input attribute is a syntax structure. The structure can be decomposed into a lower level substructure and so on, until further decomposition is not possible. The lowest level substructure is called a basic element. If the basic element is numerical then the lower bound and the upper bound of the element are given under the element. The overall structure is a tree. The tree can be arranged as a linear list with the structure preserved by a set of symbols called the tree symbols. In FAST, the specification of the product unit and product unit defectiveness is addressed by the "Symbolic Output Attribute Decomposition" (SOAD) tree which is similar to the structure of the SIAD tree.

• Statistical Analysis: Testing a piece of software is equivalent to finding the defect rate of the product unit population generated by the software. The defect rate is defined as the ratio of the number of product units that are defective to the total number of product units that the software has generated. The total number of product units, denoted by N, of any non–trivial piece of software ranges from extremely large to infinite, but can still be treated as an object of statistical interest. Although impossible in practice, it can be conceptually assumed that all N units have been produced and analyzed. Each of them can be classified as defective or non–defective. If there are D units that are defective, then the product unit population defect rate is denoted by θ in $\theta = D/N$. Since it is impossible to obtain all N units, the best approach is to estimate by means of statistical sampling.

• Test Process: Based on the SIAD/SOAD tree, FAST can automatically generate test data with an iterative sampling process which dynamically determines the sample size n (when to stop testing); the software quality can be estimated with the inspection of test results which can be automatically achieved by lexical and syntax analysis and the product unit of population defect rate which can be estimated from the sample defect rate which may be imposed on the software as the software quality index (how good is the software after testing).

● The major advantages of FAST are: Firstly, testing can be completely automated using a statistical approach, from the generation of test data based on the SIAD tree to the inspection of test results based on the SOAD tree; secondly, changing distributions do not need to be acknowledged since the SIAD tree is static; thirdly, the software quality can be assessed using statistical techniques (such as sampling or inference); fourthly, the test data do not need to be stored for regression testing, as it only requires a small space in which to keep the random number seeds; fifthly, after the specification of requirements is developed, the generation of test data is independent from the software design and implementation; sixthly, we do not need the test oracle to compute expected results and finally, testing can be performed based on the user's actual execution of the software.

3 SITE: A STATISTICS–BASED INTEGRATED TEST ENVIRONMENT

Based on FAST, SITE is developed to provide a test environment which secures automated support for the testing process. It consists of computational components, control components and an integrated database. The computational components include the modeller for modelling the software as well as the quality plan, the SIAD/SOAD tree editor for specifying the input, product unit and product unit defectiveness, the quality analyst performs statistical analysis for determining the sample size and estimating the software quality during testing, the test data generator for generating test data and the test results validator for inspecting the test results. There are two control components, the test manager and the test driver. The architecture of SITE is shown in Figure 4.

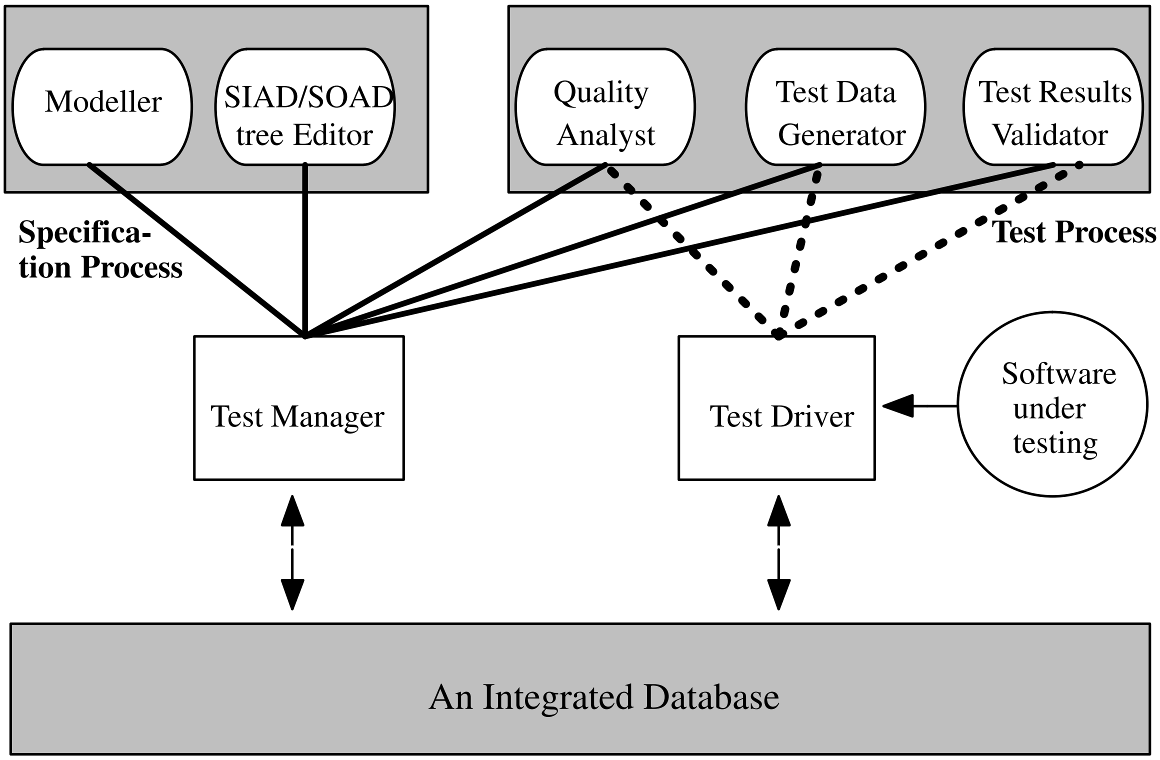


Figure 4: The architecture of SITE

3.1 Test Manager

Software testing is an extremely complicated process consisting of many activities and dealing with many files created during testing. Following this line of thought, the test manager includes two main tasks: control management and data management.

The task of control management provides an application programmatic interface (API) between tester and SITE. This API receives the command from the tester and corresponds with the functional module to execute the action and achieve the test requirements. It will trigger the test driver to start test and get the status report of test execution back which will be saved in the test report repository.

The task of data management provides the support for creating, manipulating and accessing data files plus the relations among these data files which are maintained in a persistent database in the test process. This database consists of static and dynamic data files. The static data files include a SIAD/SOAD tree file, a random number seeds file and a quality requirement file. The dynamic data files include an input unit file, a product unit file, a defect rate file, a file for the range of defect rate and a sample size file. A conceptual data model is shown in Figure 5.

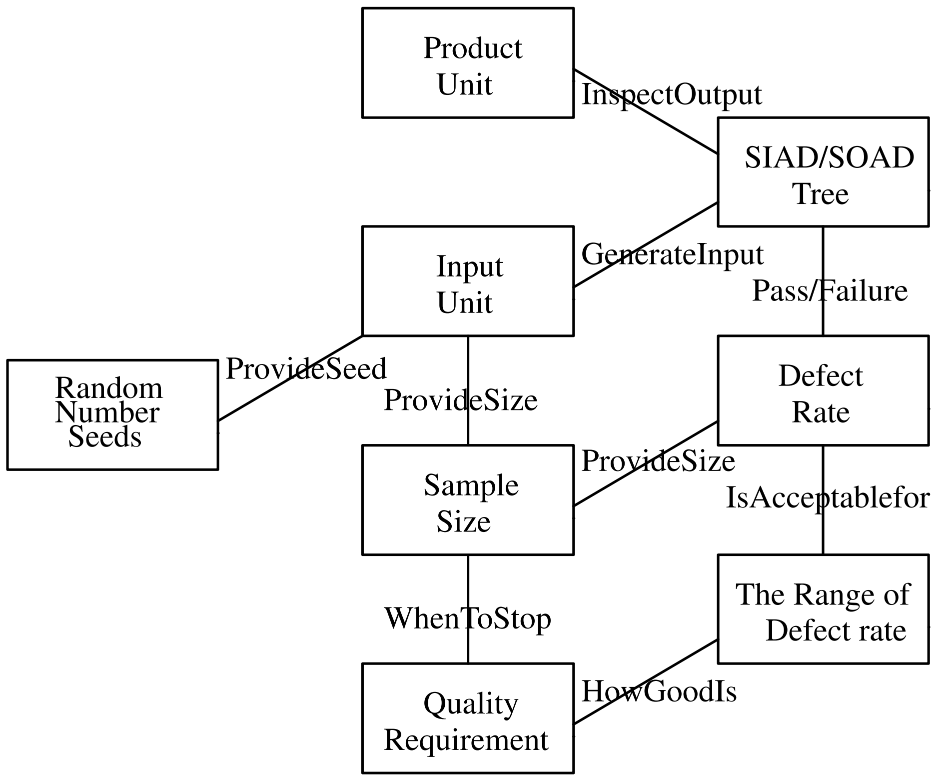


Figure 5: A conceptual data model for SITE

3.2 Test Driver

Test driver calls the software being tested and keep track of how in performs. More specifically, it should

- Set up the environment needed to call the software being tested. It may involve setting up and perhaps opening some files.
- Make a series of calls to operate the dynamic testing. The arguments for these calls could be read from a file or embedded in the code of the driver. If arguments are read from a file, they should be checked for appropriateness, if possible.

When the testing operates, the test driver triggers the test data generator to generate input according to the requirements determined by the statistical analysis of the quality analyst, makes a series of calls to execute the software and produces the product unit to the test results validator for evaluation of the tests and software.

3.3 Modeller

The Modeller operates activities which include modelling of inputs and outputs, modelling of the software. Inputs are modelled in terms of types of input data, rules for constructing inputs and sources of inputs. The modelling of output includes the crucial definitions of product unit and product unit defectiveness on which the design and testing of the software must be based. This information is used as the basis of a specification in the SIAD/SOAD tree that can be used to describe the abstract syntax of the test cases as well as to the trace data occurring during the test. The software itself, as distinct from its output, is modelled in terms of the description of the process being automated, rules for using inputs, methods for producing outputs, data flows, process control and methods for developing the software system.

The modelling of output also includes output quality planning, in which sampling methods and parameters for software testing and the acceptance procedure are determined. These parameters include firstly a definition of the defectiveness of the product unit so that the quality of a product unit can be evaluated and secondly an identification of the tolerance limits in defining the defectiveness of a product unit. This information provide support for test planning and test measurement to the statistical analyst.

3.4 SIAD/SOAD Tree Editor

Requirements specification is the activity of identifying all of the requirements necessary to develop the software and fulfil the user's needs. It covers all input, processing and output requirements. In particular, the input/output domain of the software, that is, the types of input/output, rules for using the input or examining the output and constraints on using the input/output, are identified from the modelling process and refined. Here, the SIAD/SOAD tree is a powerful tool to represent the input/output domain in a convenient form for the crucial part of requirements specification.

The SIAD/SOAD tree editor is a graphical editor that supports the editing and browsing of the SIAD/SOAD tree. The SIAD/SOAD tree and the model will be built at the same time. The modeller will trigger the SIAD/SOAD tree editor when each message (input or output) links two events during the modelling process. The result of editing will be saved in a SIAD/SOAD tree file which allows the test data generator to generate test data by a random method and the test results validator to inspect the product unit.

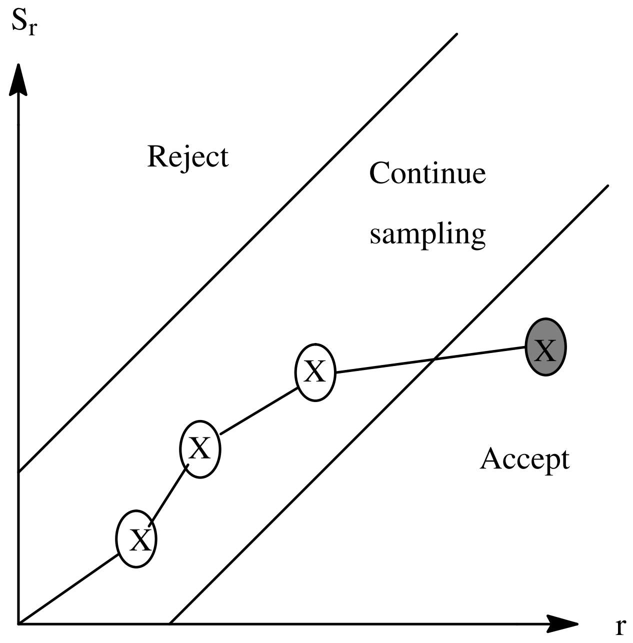
3.5 Statistical Analyst

Testing a piece of software is likely to find the defect rate of the product unit population generated by the software. Therefore, each execution of the software in SITE is considered equivalent to 'sampling' an output from the output population. The goal of statistics–based testing is to find certain characteristics of the population such as the ratio of the number of defective outputs in the population to the total number of outputs in the population. Clearly a mass inspection of the population to find the rate is prohibitive. An efficient method is through statistical random sampling. A sample of n units is taken randomly from the population. If it contains d defective units, then the sample defect rate, denoted by θ^0, is $\theta^0 = d/n$. If n is large enough, then the rate θ^0 can be used to estimate the product unit population defective rate θ.

Addressing the two major testing issues, when to stop testing and how good the software is after testing, the statistical analyst provides an iterative sampling process that determines the sample size n. It also provides a mechanism to estimate the mean, denoted by μ, of the product unit population. Once the value of μ is estimated, the product unit population defect rate θ can be computed by $\mu = n\theta$. If the value of θ is acceptable, then the product unit population is acceptable. The piece of software is acceptable only when the product unit population is acceptable. Therefore, the estimated product unit population defect rate θ can be viewed as the software quality index.

The statistical analyst receives quality statements from a quality requirement file. The quality statement defines software quality that is equivalent to $p\%$ of the output population being non–defective (the acceptance level). The result of the iterating sampling process, sample n, will be dynamically saved into a sample size file for providing information to the test data generator. The values of confidence interval is also computed and will be saved into a file for the range of defect rate for supporting the evaluation of software quality by the test results validator.

To analyse the failure data collected during the statistical testing a reliability model is needed. The model is based on a control chart with three regions, reject, continue and accept, as shown in Figure 6.



S_r: the cumulative number of defectives in the sample of r units

Figure 6: A control chart for statistical analysis

The defect rate is plotted in the chart. As long as the plots fall in the continue sampling region, the testing has to continue. If the plot falls in the rejection region, the software reliability is so bad that it has to be rejected and re–engineered. If the plots fall in the acceptance region, the software can be confidently accepted based on the required quality statement and the testing can be stopped.

3.6 Test Data Generator

After the sample size is determined, the SIAD/SOAD tree file is used for automatically generating input test data through random sampling with a random number seed. The input test data will be temporarily saved in the input unit file for re–executing according to the test requirements.

The process of automated test data generation is as follows:

Step1: Determine the sample size n by statistical analysis,

Step2: Generate the number of test data M for each sample by a random number seed,

Step3: The construction of test data using a SIAD/SOAD tree can be accomplished as follows:

3.1 Let K be the number of elements in the SIAD/SOAD tree. Each element in the tree is indexed by a number ranging from 1 to K. A random number selected from $[1,K]$ is produced by using a random number generator.

3.2 The element with its index equal to the random number is selected.

3.3 If the element has a parent in the SIAD/SOAD tree, then backtrack to select it.

Step4: A total of M elements will be randomly sampled from a tree for designing test data.

3.7 Test Results Validator

A test results validator in SITE is like a compiler. Much as a compiler reads and analyzes source code, the test results validator reads and analyzes the test results with the SIAD/SOAD tree (specification information). It introduces the static testing method to inspect the test results during dynamic testing. The main advantage of using the SIAD/SOAD tree here is that we do not need a test oracle to compute expected results. The SIAD/SOAD tree can be used directly for automatic inspection whether or not the results produced by the software are correct.

The test results validator receives the test results during test execution. After inspecting the test results, it will compute the defect rate and store it in the defect rate file thus providing data to the quality analyst dynamically. According to test requirements, the test failure report is produced by the test results validator.

4 COMPARISON WITH OTHER TEST ENVIRONMENTS

4.1 An Overview of the STEP Model and SAAM

Eickelmann and Richardson [9] presented the Software Test Environment Pyramid (STEP) model which partitioned the STE domain into six canonical functions, test execuation, test development, test failure analysis, test measurement, test management and test planning, in a corresponding progression of test process evolution in [9, 10] – the debugging (1950–56), demonstration (1957–78), destruction (1978–82), evaluation (1983–87) and prevention (1988–95) periods. This correspondence is shown in Figure 7 and the full details can be seen in [9].

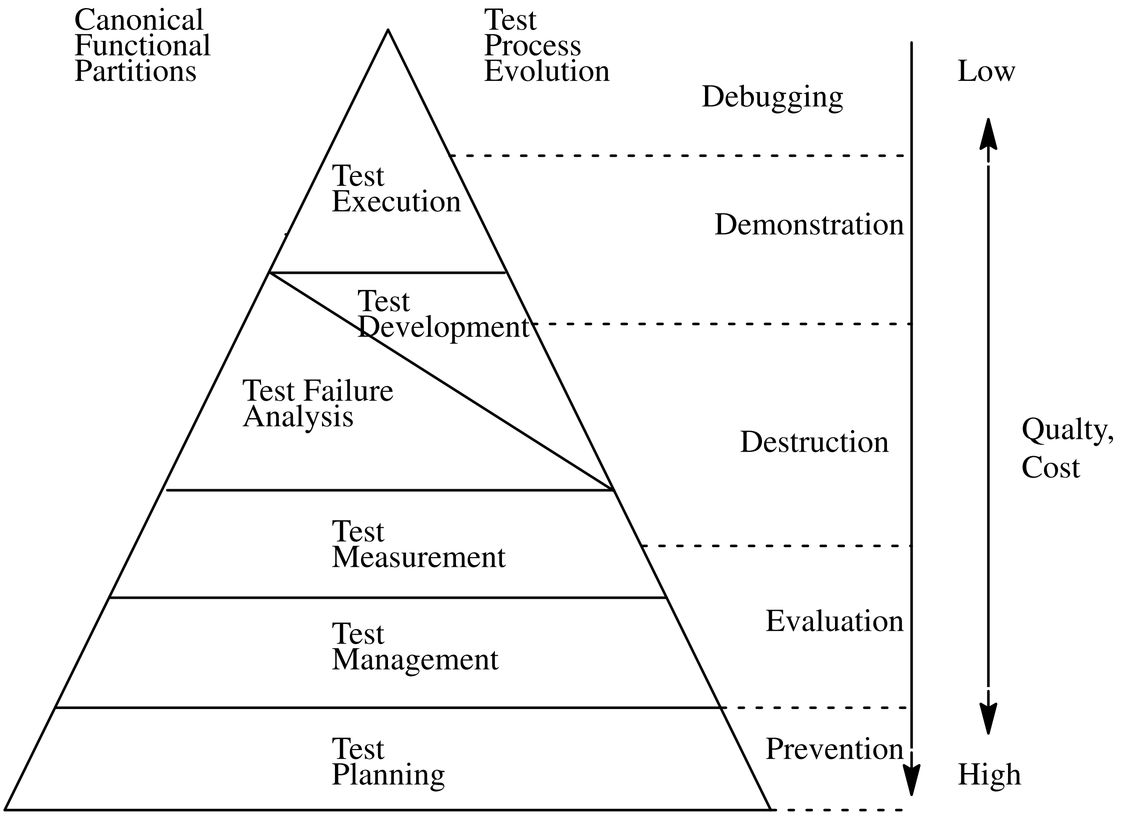


Figure 7: STEP Model, adapted from [9]

Based on the STEP model, a comparison and analysis of three STEs [9], PROTest II (Prolog Test Environment, Version II) [2], TAOS (Testing with Analysis and Oracle Support) [16] and CITE (CONVEX Integrated Test Environment) [18], was made by the Software Architecture Analysis Method (SAAM) which provides an established method for describing and analyzing software architectures. To accomplish this work, there are three main steps in SAAM [9]: firstly, to characterize a canonical functional partition for the domain, secondly, to create a graphical diagram of each system's structure by the SAAM graphical notation and thirdly, to allocate the functional partition onto the system structure. This done, the three STEs were analysed to determine whether the architecture supports specific tasks or not in accordance with the qualities attributable to the system.

In the graphical notation used by SAAM there are four types of components [12]: a process (unit with an independent thread of control); a computational component (a procedure or module); a passive repository (a file) and an active repository (a database). There are two types of connectors, control flow and data flow, either of which may be uni or bi–directional. The notation is a concise and simple lexicon shown in figure 8.

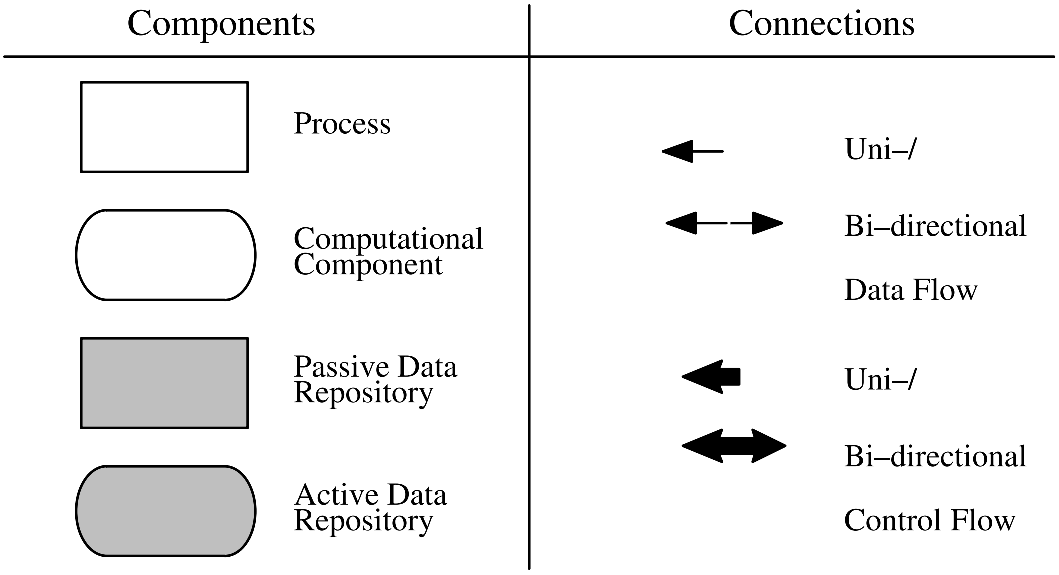


Figure 8: SAAM architectural notations, adapted from [12]

4.2 SITE SAAM Description And Functional Allocation

The SAAM graphical depiction of SITE is shown in Figure 9. SITE supports statistics–based testing on the top of specification–based testing with two main issues in software testing, when to stop testing and how good the software is after testing. It provides automatic support for test execution by the test driver, test development by the SIAD/SOAD tree editor and the test data generator, test failure analysis by the test results validator, test measurement by the statistical analyst, test management by the test manager and test planning by the modeller. These tools are integrated around an object management system which includes a public, shared data model describing the data entities and relationships which are manipulable by these tools.

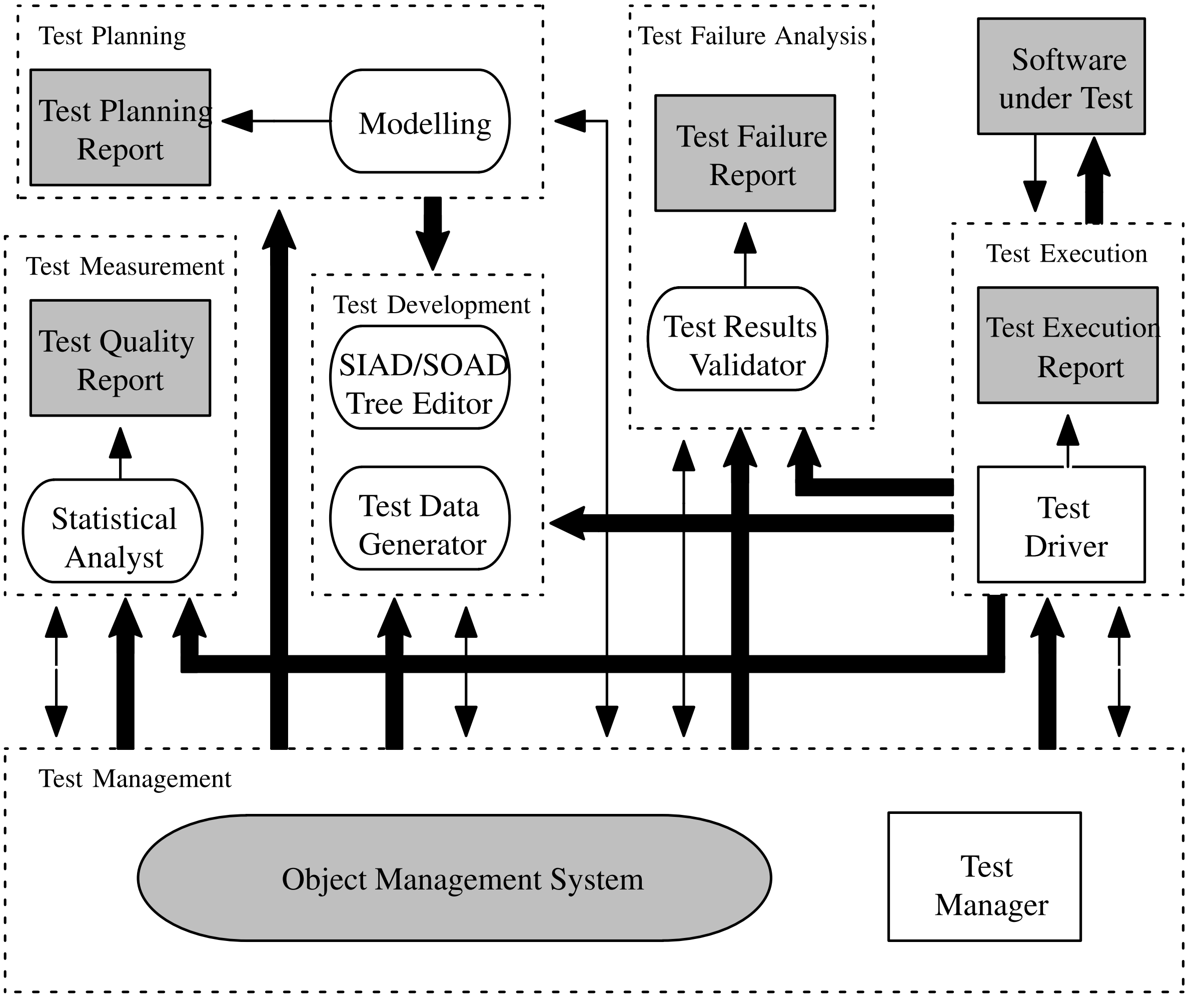


Figure 9: The SITE system structure and functional allocation through SAAM

SITE enables early entry of the test process into the life cycle due to the definition of the quality planning in the modeller. After well–prepared modelling and requirements specification are undertaken, the test process and the software design and implementation can proceed concurrently.

4.3 STE Comparison

The use of SAAM provides a canonical functional partition to characterize the system structure at a component level. The functionalities supported and structural constraints imposed by the architecture are more readily identified when compared in a uniform notation [9]. A comparison of four STEs, PROTest II, TAOS, CITE and SITE, made by the SAAM is shown in Figure 10.

	Test Execution	Test Development	Test Failure Analysis	Test Measurement	Test Management	Test Planning
PROTest II	✓	✓		✓		
TAOS	✓	✓	✓	✓	✓	
CITE	✓	✓	✓	✓	✓	
SITE	✓	✓	✓	✓	✓	✓

Figure 10: A comparison of four STEs by SAAM

However, the test process focus was identified for each STE across the software development life cycle, shown in Figure 11.

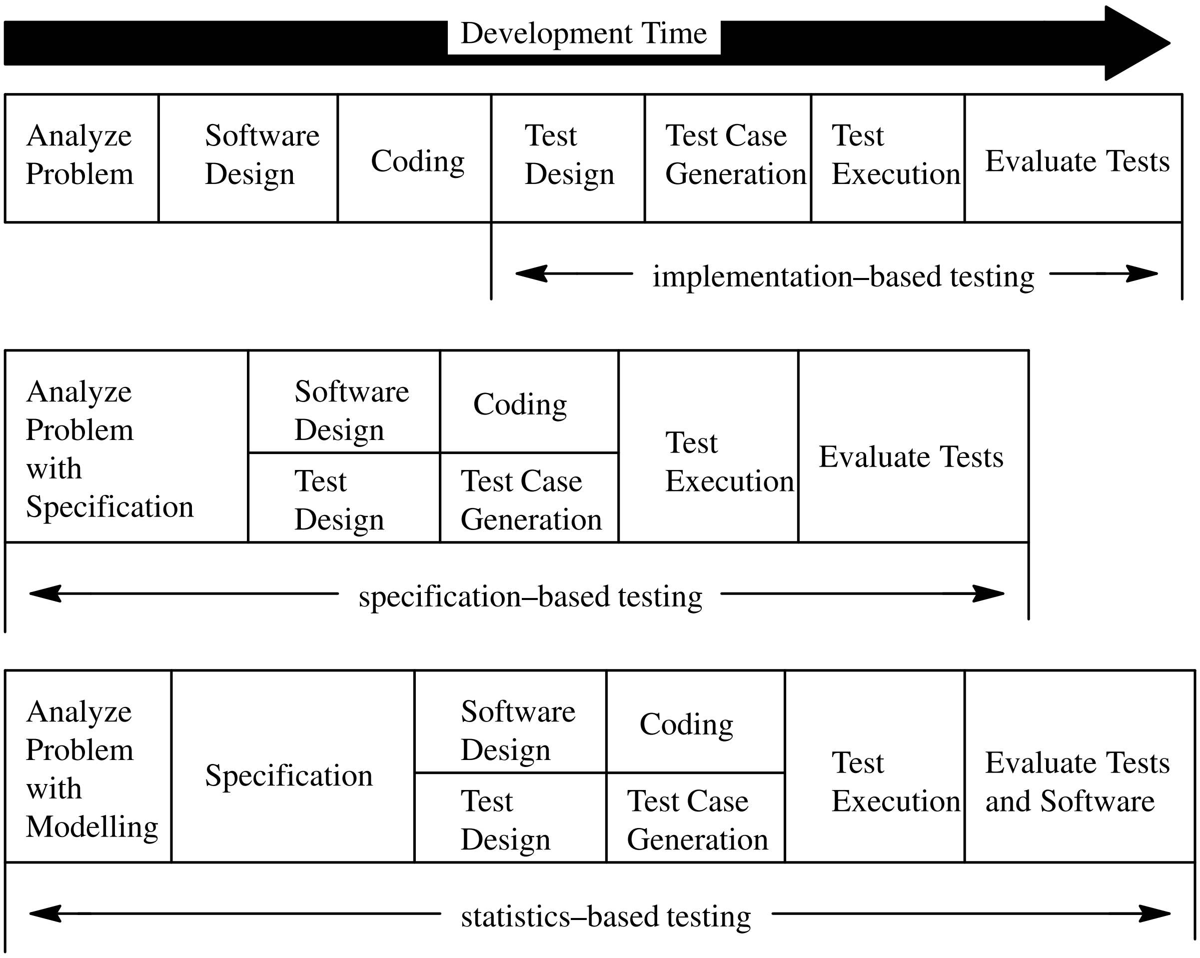


Figure 11: Test process focus appears the life cycle

- PROTest II and CITE support implementation–based testing and have a destructive testing process focus. This focus has a limited scope of life cycle applicability, as it initiates testing after implementation for the purpose of detecting failures.
- TAOS supports specification–based testing and has an evaluative test process focus. An evaluative test process focus provides complete life cycle support, as failure detection extends from requirements and design to code.
- SITE supports statistics–based testing and has a prevention testing process focus. It focuses on fault prevention through parallel development and test processes. SITE uses the way that timely testing improves software specifications by building models that show the consequences of the software specifications.

There exist some differences amongst implementation–based, specification–based and statistics–based testing. With implementation–based testing, only a set of input data can be generated from an implementation but the expected outputs cannot be derived from the implementation. In this case, the existence of an oracle (in the human mind) must be assumed and it is necessary to check the test results against the oracles. With specification–based testing, both test input data and the expected outputs can be generated from a specification. However, it does not provide an index to show how good the software is after testing. The statistics–based testing is on the top of the specification–based testing with the quality plan before specification. The quality plan addresses testing issues such as the testing strategy, test tools needed, acceptance criteria, what reviews are to be performed and risk management (e.g. what tests not to do, when to stop testing). With statistics–based testing the tests are developed according to prevailing standards and at the right time so that the software quality will be achieved.

5. CONCLUSION

The production of high–quality information applications to secure the reliability of the software is a very important issue in information system development. An essential component for developing quality software is software testing. However, It is a very time–consuming and tedious activity and accounts for over 30% of the cost of software development. In addition to its high cost, manual testing is unpopular and often inconsistently executed. Therefore, the support of fully automated test environment for the software have been desired a significant issue to the software development process. In this paper, a statistics–based integrated test environment is proposed. It consists of control components (test manager, test driver) and computational components (Modeller, SIAD/SOAD tree editor, statistical analyst, test data generator, test results validator). The activities of the test process are integrated around an object management system which includes a public, shared data model describing the data entities and relationships which are manipulable by these tools. SITE provides automated support for test execution, test development, test failure analysis, test measurement, test management and test planning. It enables early entry of the test process into the life cycle due to the modelling stage. After well–prepared modelling and requirements specification are undertaken, the test process and the software design and implementation can proceed concurrently.

Acknowledgement

The authors would like to thank Ms Nancy S. Eickelmann for her comments which were useful for improving the paper. The work of one author, H.D. Chu, was funded by the National Science Council in Taiwan from whom he received a fellowship to work toward a doctoral degree.

References

1. B. Beizer: Software Testing Techniques (Second ed.), Van Nostrand Reinhold, New York, 1990.
2. F. Belli and O. Jack: Implementation–based analysis and testing of prolog programs, in In–ternational Symposium on Software Testing and Analysis, 1993, pp. 70–80.
3. A. Bertolino: An Overview of Automated Software Testing. Journal Systems Software, 15, 1991, pp. 133–138.
4. C.K. Cho: Quality Programming – Development and Testing Software wih Statistical Qualty Control, John Wiley & Sons, New York, 1988.
5. H.D. Chu and J.E. Dobson: FAST: A Framework for Automating Statistics–based Testing, Technical Report 564, Department of Computing Science, University of Newcastle upon Tyne, 1996.
6. P.A. Currit, M. Dyer and H.D. Mills: Certifying the Reliability of Software, IEEE Transactions on Software Engineering, SE–12(1), 1986, pp. 3–11.
7. M. Deck: Cleanroom practice: a theme and variations, in Proceedings of the 9th International Software Quality Week, 1996..
8. M. Dyer: The Cleanroom Approach to Quality Software Development, John Wiley & Sons, New York, 1992.
9. N.S. Eickelmann and D.J.Richardson: An Evaluation of Software Test Environment Architec–tures, In 18th International Conference on Software Engineering, 1996, pp. 353–364.
10. D. Gelperin and B. Hetzel: The Growth of Software Testing. Communication of the ACM, 31(6), 1988, pp. 687–695.
11. D. Ince: The automatic generation of test data, Computer Journal, 30(1),, 1987, pp. 63–69.
12. R. Kazman, L., Bass, G. Abowd and W. Webb: SAAM: A Method for Analyzing the Prop–erties of Software Architectures. in 16th International Conference on Software Engineering, 1994, pp. 81–90.
13. B. Marre, P. Thévenod–Fosse, H. Waeselynck, P.L. Gall and Y. Crouzet: An Experimental Evaluation of Formal Testing and Statistical Testing, in Predictably Dependable Computing Systems, Springer, 1995, pp. 273–281.
14. G. J. Myers: The Art of Software Testing,(John Wiley & Sons, New York, 1979.
15. S. Norman: Software Testing Tools, Ovum Ltd, London, 1993.
16. D.J. Richardson: TAOS: Testing with Oracles and Analysis Suport, in International Sympo–sium on Software Testing and Analysis, 1994, pp. 138–153.
17. H.V. Vliet: Software Engineering: Principles and Practice, John Wiley & Sons, New York, 1994.
18. P.A. Vogel: An Integrated General Purpose Automated Test Environment, in International Symposium on Software Testing and Analysis, 1993, pp. 61–69.