AN INTEGRATED TEST ENVIRONMENT FOR DISTRIBUTED APPLICATIONS

Huey–Der Chu and John E Dobson

Centre for Software Reliability, Department of Computing Science

University of Newcastle upon Tyne, NE1 7RU, UK

ABSTRACT

Software testing is an essential component in achieving software quality. 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) for testing distributed applications. To address two crucial issues in software testing, when to stop testing and how good the software is after testing, SITE provides automatic support for test execution, test development, test failure analysis, test measurement, test management and test planning.

Keywords: Software Test Environment, Statistics–based Testing, Distributed Applications

1 INTRODUCTION

Software testing is a very time–consuming and tedious activity and therefore a very expensive process and accounts for over 30% of the cost of software system development [8, 9]. In addition to its high cost, manual testing is unpopular and often inconsistently executed. To achieve the high quality required of software applications, a powerful environment that automates sophisticated testing and analysis techniques is needed. Therefore, Software Testing Environments (STEs) overcome the deficiencies of manual testing through automating the test process and integrating testing tools to support a wide range of test capabilities [5]. The use of STE provides significant benefits as follows [11, 13]. 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.

The Statistics–based Integration Test Environment (SITE) provides a test environment based on statistical testing which secures automated support for the testing process, including modeling, specification, statistical analysis, test data generation, test results inspection and test path tracing. Testing of a distributed application is very complex because such a system is inherently concurrent and non–deterministic. It adds another degree of difficulty to the analysis of the test results. Therefore, a systematic and effective test environment for the distributed applications is highly desirable. To address these problems, the SITE is developed on the Java Development Kit (JDK) which provides Java Application Programming Interface (API) and Java tools for developing distributed client/server applications.

In Section 2 of this paper, an operational environment for testing distributed software is presented. A basic architecture of automated software testing is introduced in Section 3. An overview of our approach is shown in the end of this section. In Section 4, the architecture of SITE is described and the relation of the main components is also shown. A comparison of STEs using the SAAM structure is discussed in Section 5. Section 6 summarizes my research work.

2 AN OPERATIONAL ENVIRONMENT FOR TESTING DISTRIBUTED SOFTWARE

Distributed applications have traditionally been designed as systems whose data and processing capabilities reside on multiple platforms, each performing an assigned function within a known and controlled framework contained in the enterprise. Even if the testing tools were capable of debugging all types of software components, most do not provide a single monitoring view that can span multiple platforms. Therefore, developers must jump between several testing/monitoring sessions across the distributed platforms and interpret the cross–platform gap as best they can. That is, of course, assuming that comparable monitoring tools exist for all the required platforms in the first place. This is particularly difficult when one server platform is the mainframe as generally the more sophisticated mainframe testing tools do not have comparable PC– or Unix–based counterparts. Therefore, testing distributed applications is exponentially more difficult than testing standalone applications.

To overcome this problem, we present an operational environment for testing distributed applications based on the Java Development Kit (JDK) as shown in Figure 1, allowing testers to track the flow of messages and data across and within the disparate platforms.

The primary goal of this operational environment is an attempt to provide a coherent, seamless environment that can serve as a single platform for testing distributed applications. The hardware platform of the testbed at the lowest level in Figure 1, is a network of SUN workstations running the Solaris 2.x operating system which often plays a part in distributed and client–server system. The widespread use of PCs has also prompted an ongoing effort to port the environment to the PC/Windows platform. On the top of the hardware platform is Java Development Kit. It consists of the Java programming language core functionality, the Java Application Programming Interface (API) with multiple package sets and the essential tools such as Remote Method Invocations (RMI), Java DataBase Conncetivity (JDBC) and Beans for creating Java applications. On top of this platform is the SITE which secures automated support for the testing process, including modeling, specification, statistical analysis, test data generation, test results inspection and test path tracing. At the top of this environment are the distributed applications. These can use or bypass any of the facilities and services in this operational environment. This environment receives commands from the users (testers) and produces the test reports back.

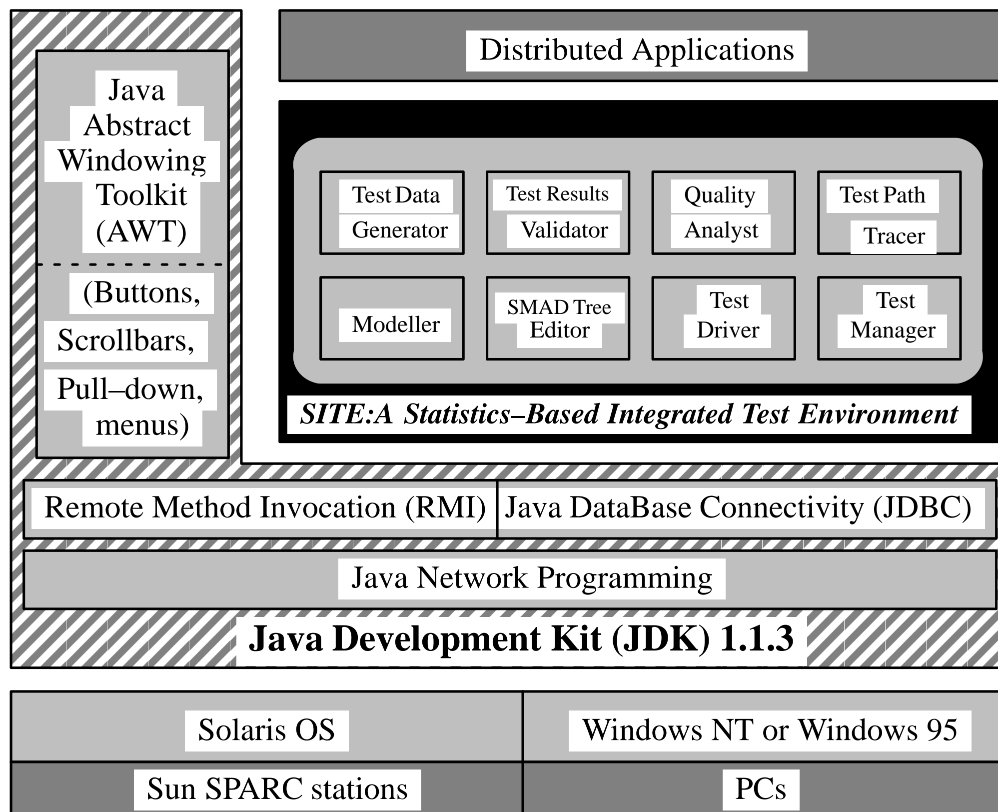

Figure 1: An operational environment for testing distributed applications

The picture given in Figure 1 shows an approximate idea of how the various parts of the operational environment fit together. It gives an indication of the gross structure, so henceforth we will use it as our model.

3 A BASIC ARCHITECTURE OF AUTOMATED SOFTWARE TESTING

In this section, a process of automated testing is described for distributed applications. Testing is a method to validate that the behaviour of an object is conforming to its requirements specification. Therefore, before testing, the requirements specification activity should be to specify the detail input data, expected results and non–deterministic or deterministic behaviour of a distributed application. Formal or semi–formal specification techniques may be appropriate for expressing such a specification which can act as a basis for test data generation, test execution and test result validation. The basic architecture of automated testing is shown as in Figure 2.

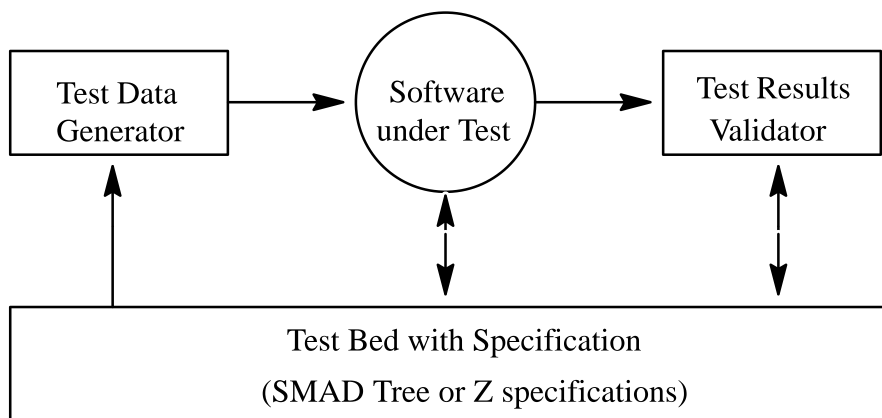

Figure 2: The basic architecture of automated testing

3.1 Requirements Specification

Requirements specification is the activity of identifying all of the requirements necessary to develop the software and fulfill the user's needs [2]. Not only do testers need specified–behaviour information in order to detect whether or not the test results satisfy their requirements but other software developers must have that information. However, the functional requirements is not enough in achieving software quality. What we need to do is to add quality to software during the engineering process. To achieve this, testers must be conscious of quality requirements at the same time they are building in functional requirements[4]. In the

other words, the objective of this paper is to answer the two crucial questions in software testing:

- When to stop testing (whether distributed or not)
- How good the software is after testing

Therefore, requirements specification must include the software requirements, test requirements and quality requirements[2, 3]:

- Software requirements include input, processing and output requirements. The input requirements consist of the types of input, quality characteristics of each type of input, rules for using the input and constraints on using the input. The process requirements contain exhaustive listings of the functions the software must have. The output requirements include man–machine interface and other characteristics of the products to be generated by the software.
- Test requirements consist of a test plan with which the software is to be tested and accepted. They define the product units and product unit defectiveness for statistical sampling, sampling methods for estimating the defect rate of the software population with which to judge software quality, statistical inference methods and confidence level of software output population quality, the acceptable software defect rate and test input unit generation methods.
- Quality requirements is specified by some activities that are followed to analyze the user's needs for quality, to convert the quality needs to requirements and to document the results of the software requirements analysis. These documents must clearly be validated by users since only they know what they want.

A requirements specification presented to a tester could be as informal as a set of notes scribbled during a meeting or as formal as a document written in a specification language. Formal languages such as Z, VDM, LOTOS, etc. [6, 10] have been promoted strongly by the academic community in recent years although their take up in industry has been patchy. They are particularly well suited for specification–based testing. However, the method of test data selection in these approaches was based on the deterministic testing method and two main issues of software testing–when to stop testing and how good the software is after testing–were only briefly discussed in these approaches.

3.2 Test Data Generator

Test data generation is a process of selecting execution path/input data for testing. Most of the approaches dealing with automatic test data generation are based on the implementations code, either using stochastical methods for generation or symbolic execution. This seems quiet natural, since in a traditional software development process this usually is the only ‘‘specification” that has formal semantics allowing detailed, automatic analysis. Using formal specifications these tasks can now be carried out along the specification. Different work with formal specification has been done as well, either describing manual or automatic test data generation [6].

A test data generator is a tool which assists a tester in the generation of test data for a software. It takes formally recorded specification information, treats it as though it were a knowledge based or data base and applies test design rules to this base to automatically create test data. If a requirement changes in the knowledge base, new test data can be designed, generated, documented and traced.

3.3 Test Execution

Test execution is a process of feeding test data to the software and collecting information to determine the correctness of the test run. For a sequential software, this process can be accomplished without difficulty. However, for distributed applications, some test cases can be very hard to execute because by having more than one process executing concurrently in a system, there are non–deterministic behaviours. Repeated executions of a distributed software with the same input may execute different paths in the distributed software and produce different results. This is called the non–reproducible problem. Therefore, a mechanism is required in order to exercise these test cases.

3.4 Test Results Validator

Validation of test results is a process of analyzing the correctness of the test run. For the sequential software, the correctness of an execution can be observed by comparing the expected and software the software outputs. However, for distributed applications again because of non–determinism, there are more than one or possible infinite outputs for one execution. Validation of such test results is much more difficult than that of the sequential test results.

The behaviour of a distributed application can be represented by sequences of communication events. Each such sequence represents a possible interaction of communication events. Generally, the event sequence is long and the number of all possible sequences is usually extremely large. Because of the non–determinism of distributed applications, using breakpoints to validate the execution result is not acceptable. To reduce the interference of testing to the system, it is required that the sequence of events transmitted during the execution be recorded in a so–called execution history file for an off–line analysis. However, it is erroneous and tedious work if this analysis is done by human. Thus, an automated analysis tool is required.

3.5 Our Approach

To guide testers in testing distributed software, the tool, the SMAD tree which is between formal and informal specification, is presented. Extending this concept of the SIAD/SOAD tree in FAST [3], we attempt to specify all possible delivered messages between events by means of the "Symbolic Message Attribute Decomposition" (SMAD) tree. It combines with classification and syntactic structure to specify all delivered messages. In the upper level of the SMAD tree, we classify all delivered messages into three types of message: input message, intermediate message and output message. Each type of message has a syntactic sub–tree describing the characteristics of messages with a happen–before relationship so that it can be determined whether messages were delivered in an order consistent with the potential causal dependencies between messages.

The SMAD tree is used to define the test case, which consists of an input message plus a sequence of intermediate messages, to resolve any non–deterministic choices that are possible during software execution, e.g., the exchange of messages between processes. In other words, the SMAD tree can be used in two ways, firstly to describe the abstract syntax of the test data (including temporal aspects) and secondly to hold data occurring during the test.

A test data input message can be generated based on the input message part of the SMAD tree and rules for setting up the ordering of messages which are incorporated into the tree (initial event) .The intermediate message part of the SMAD tree can trace the test path and record the temporal ordered relationship during the lifetime of the computation. The test results also can be inspected based on the output message part of the SMAD tree (final event), both with respect to their syntactic structure and the causal message ordering

under repeated executions.

For testing distributed applications, the test strategy consists of testing on two levels: component testing and interaction testing. The component testing is based on dynamic testing which combines FAST and other testing techniques. The interaction testing can reveal potential behavioural properties of a distributed software using deterministic testing.

4 SITE: A STATISTICS–BASED INTEGRATED TEST ENVIRONMENT

The objective of SITE is to build a fully automated testing environment with the statistical analysis. The architecture of SITE suggested in Figure 3 consists of computational components , control components and an integrated database. The computational components include the modeller, the SMAD tree editor, the quality analyst, the test data generator, the test paths tracer, the simulator and the test results validator. There are two control components, the test manager and driver.

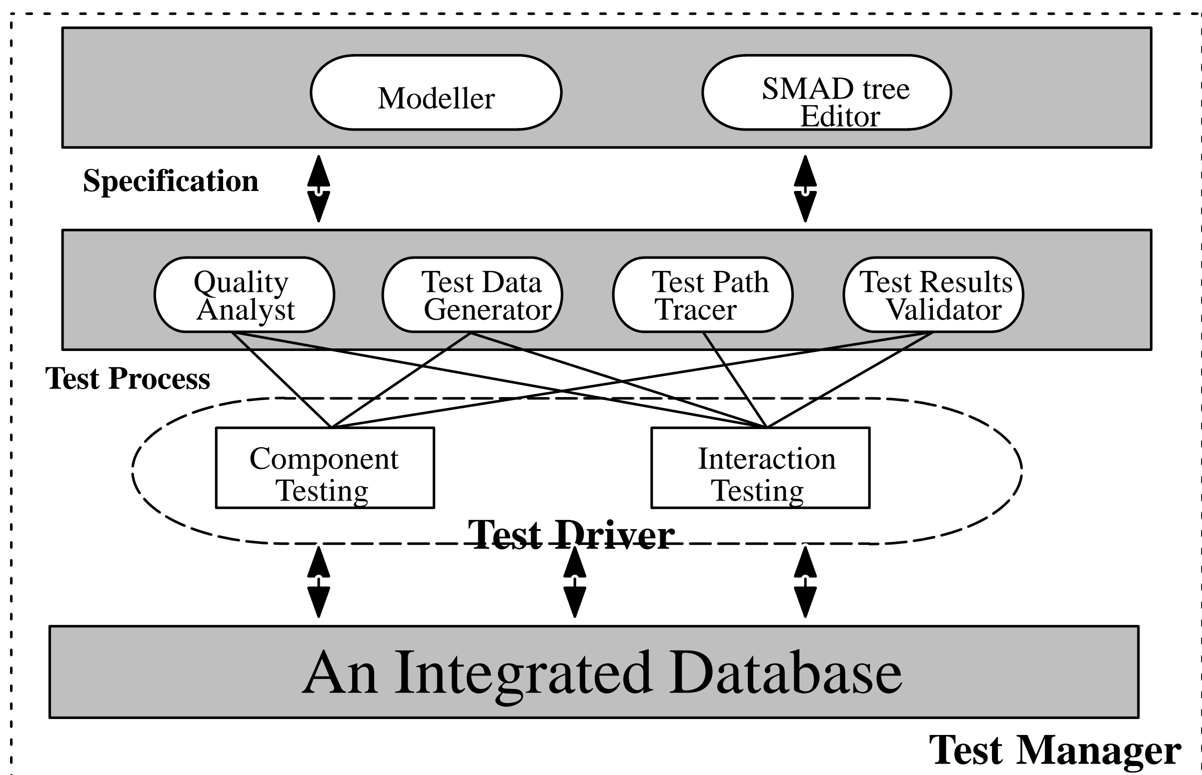

Figure 3: The architecture of SITE

The SITE is designed for distributed applications, according to the test requirements:

- To set up the requirements, including the functional and quality requirements,
- To execute automated testing until it has been sufficiently tested (when to stop testing),
- To re–execute the input units which have been tested (regression testing),
- To execute the component–testing first and the interaction testing second,
- To test all ‘‘interface’’ paths among processes which should be traversed at least once,
- To enhance testing in areas that are more critical,
- To produce test execution, test failure and test quality reports.

For a distributed application, the test environment could model the executing behaviour, edit messages' specification into a SMAD tree file, automatically generate test data based–on statistical testing, receive a test software, run the software with the generated test data, trace the test paths recording in the path records file for re–tests, inspect the test results and finally generate a test report to the tester.

4.1 Test Manager

Software testing is an extremely complicated process consisting of many activities and dealing with many files created during testing. The test manager has two main tasks: control and data management.

The task of control management provides a GUI between tester and SITE. This GUI receives commands from the tester and corresponds with the functional module to execute the action and achieve the test results. 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 as well as 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 message–flow paths file, a SMAD 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 test paths recording file, a defect rate file, a file for the range of defect rate and a sample size file.

A conceptual data model for this database is shown in Figure 4. These data files will be described more fully through this paper as they arise.

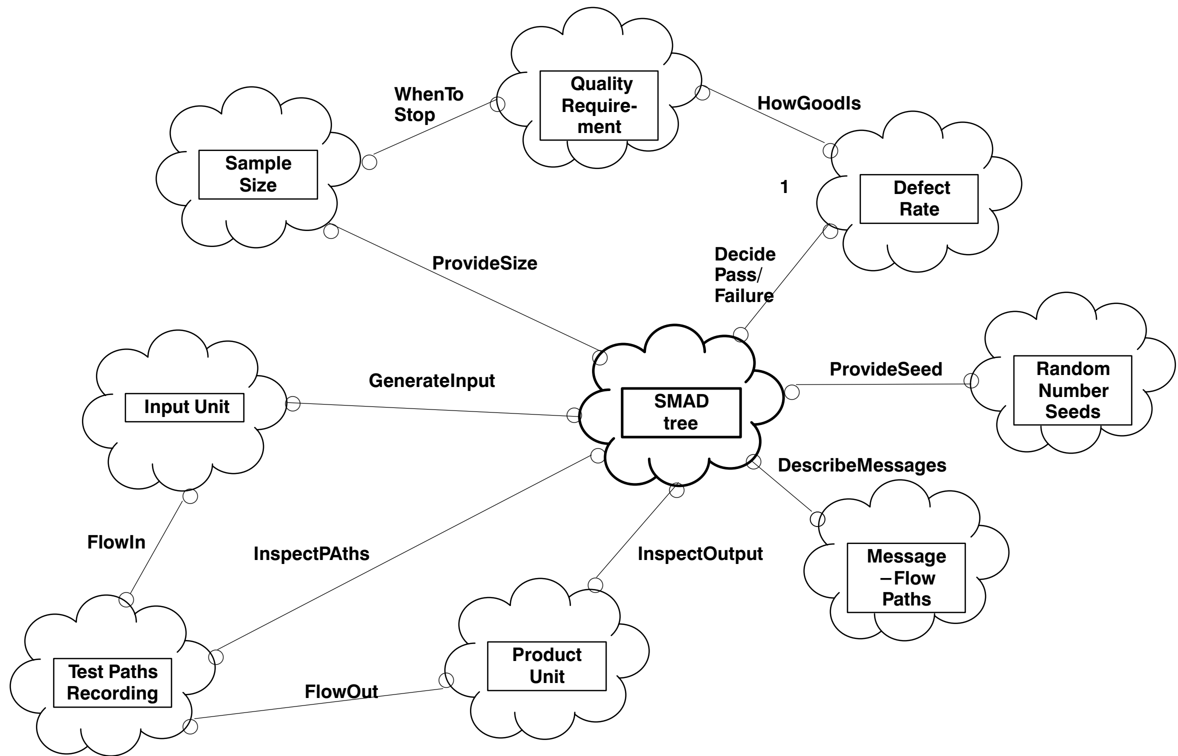

Figure 4: A conceptual data model for SITE

4.2 Modeller

The Modelling activity includes [3]: modelling of inputs and outputs as well as 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. 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.

A distributed application is a system as a set of communicating processes, where each process holds its own local data and the processes communicate by message passing. In SITE, the modelling component describes a set of asynchronous processes in a distributed application to be tested with message–flow routines to gather information about an application's desired behaviour from which all tests are then automatically derived.

This model is used as the basis of a specification in the SMAD 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 message–flow routines will provide an elemental function visible at the system level and constitute the point at which integration and system testing meet, which results in a more seamless flow between these two forms of testing. This information provides support for test planning (a component testing and an interaction testing) to the test driver as well as the SMAD tree editor for specifying messages among events.

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.

4.3 SMAD Tree Editor

Requirements specification is the activity of identifying all of the requirements necessary to develop the software and fulfil user needs. Here, the SMAD tree is a powerful tool to represent the input/output domain in a convenient form for the crucial part of requirements specification.

The SMAD tree editor is a graphical editor that supports editing and browsing of the SMAD tree. The SMAD tree and the model will be built at the same time. The modeller will trigger the SMAD tree editor when each message links two events during the modelling process. The result of editing will be saved in a SMAD 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.

4.4 Test Driver

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

- Set up the environment needed to call the software being tested. This 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.

There are some different activities between the component testing and interaction testing. Therefore, the test driver invokes different computational components/sub–components in different level testing. This difference is shown in Figure 3.

During the component testing, 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 application and produces the product unit to the test results validator for evaluation of the tests and software.

After the component testing, the test driver performs the interaction testing. It starts by calling the call test data generator to generate an input message plus a sequence of intermediate messages which are selected to correspond to the message–flow paths file and sets up the ordering of messages using ‘happened before’ relationships which are incorporated into the SMAD tree. When the test runs, the test driver invokes the test paths tracer to trace the test path and record the temporal ordered relationship into the path recordings file during the lifetime of the computation. The test results also can be saved into the product unit file to the test results validator for inspecting the product unit, both with respect to their syntactic structure and the causal message ordering under repeated executions using the path recordings file.

4.5 Quality Analyst

4.5.1 Statistical Analysis For Component Testing

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’ a product unit from the population which consists of an infinite number of units. The goal of statistics–based testing is to find certain characteristics of the population such as the ratio of the number of defective units in the population to the total number of units 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 dynamically 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 full details can be seen in [3].

The statistical analyst receives quality statements from a quality requirement file. The quality statement defines software quality that is equivalent to p% of the product unit population being non–defective (the acceptance level). The result of the iterating sampling process, sample size n, will be dynamically saved into a sample size file for providing an information to the test data generator. The values of confidence interval also is 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.

4.5.2 Test Coverage Analysis For Interaction Testing

The objective of interaction testing is to to verify the message exchanges among processes. One reasonable cover would be to require that all ‘‘interface” messages between a pair of process should be exercised at least once. The ‘‘interface” message is the message sent out and received from different processes. In SITE, we can use the path recordings file in comparison with the message–flow paths to examine whether or not there are ‘‘interface” messages which do not verify. If so, more tests are added until the test set is sufficient for the quality level required.

4.6 Test Data Generator

After the sample size is determined, the SMAD 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 regression tests according to the test requirements.

For interaction testing, the test generator addresses how to select the input test data plus event sequences from the SMAD tree with the ‘‘happened before” relationship. Due to the unpredictable progress of distributed processes and the use of non–deterministic statements, multiple executions of an application

with the same input may exercise different message–flow paths. Therefore, the input test data plus event sequences are generated with reference to the message–flow paths file.

4.7 Test Path Tracer

The reproducibility of tests is important, particular in testing distributed applications. Therefore, we need a mechanism for tracing and recording test paths during the test. The tracer consists of correlated views that allow the tester to compare different information about a path routing in the software execution. The path tracer records events from currently executing tasks into a path records file, where the trace is played in ‘‘real time”. Once a path record file has been created, the tester can replay the trace for re–tests.

4.8 Test Results Validator

A test results validator in SITE is like a compiler. Much as a compiler reads and analyzes source code, the validator reads and analyzes the test results with the SMAD tree. It introduces the static testing method to inspect the test results during dynamic testing. The main advantage of using the SMAD tree here is that we do not need a test oracle to compute expected results. The SMAD tree can be used directly for automatic inspection whether or not the results produced by the software are correct. In the interaction testing, the validator examines the execution of different test paths which drive from different test data or from the same test data (repeated execution) to test the causal message ordering with the ”happened before” relationships in the SMAD tree.

The 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 validator.

5 COMPARISON WITH OTHER TEST ENVIRONMENTS

5.1 An Overview Of The STEP Model And SAAM

Eickelmann and Richardson [5] developed the Software Test Environment Pyramid (STEP) model which partitioned the STE domain into six canonical functions, test execution, test development, test failure analysis, test measurement, test management and test planning, in a corresponding progression of test process evolution in [5] – the debugging, demonstration, destruction, evaluation and prevention periods.

This correspondence is shown in Figure 5 and the full details can be seen in [5].

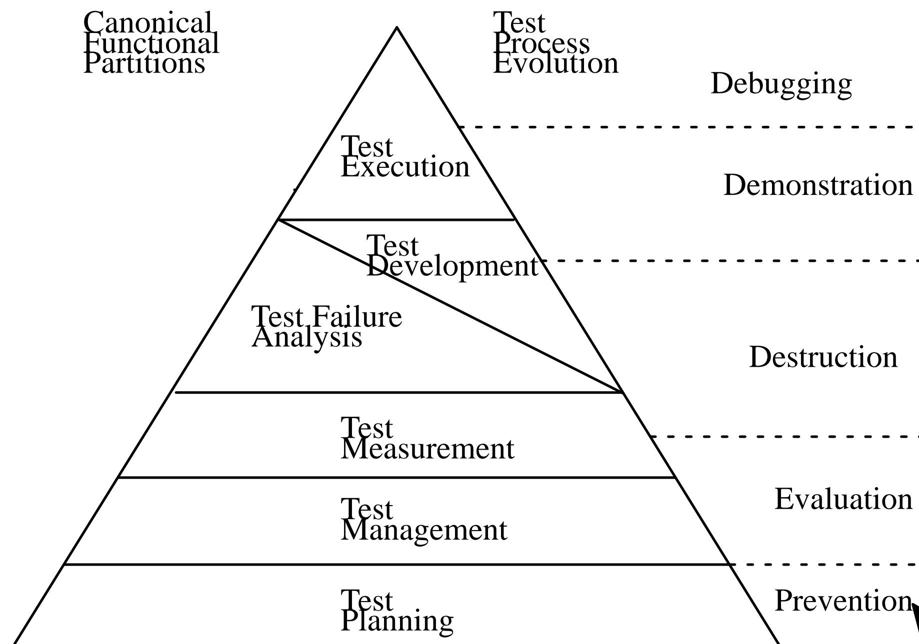

Figure 5: STEP Model, adapted from [5]

Based on the STEP model, a comparison and analysis of three STEs [5], PROTest II (Prolog Test Environment, Version II) [1], TAOS (Testing with Analysis and Oracle Support) [11] and CITE (CONVEX Integrated Test Environment) [13], 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 [5]: 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 [7]: 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 6.

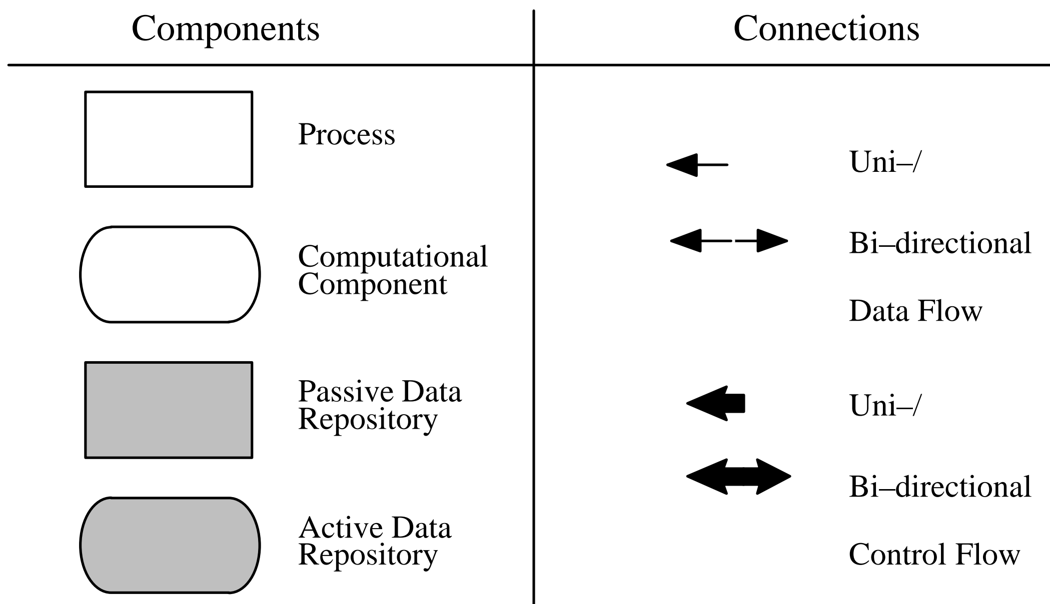

Figure 6: SAAM architectural notations, adapted from [7]

5.2 SITE SAAM Description And Functional Allocation

The SAAM graphical depiction of SITE is shown in Figure 7. 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 SMAD tree editor and the test data generator, test failure analysis by the test results validator, test measurement by the quality analyst, test management by the test manager and test planning by the modeller. These tools are integrated around an object management system [12] which includes a public, shared data model describing the data entities and relationships which are manipulable by these tools.

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

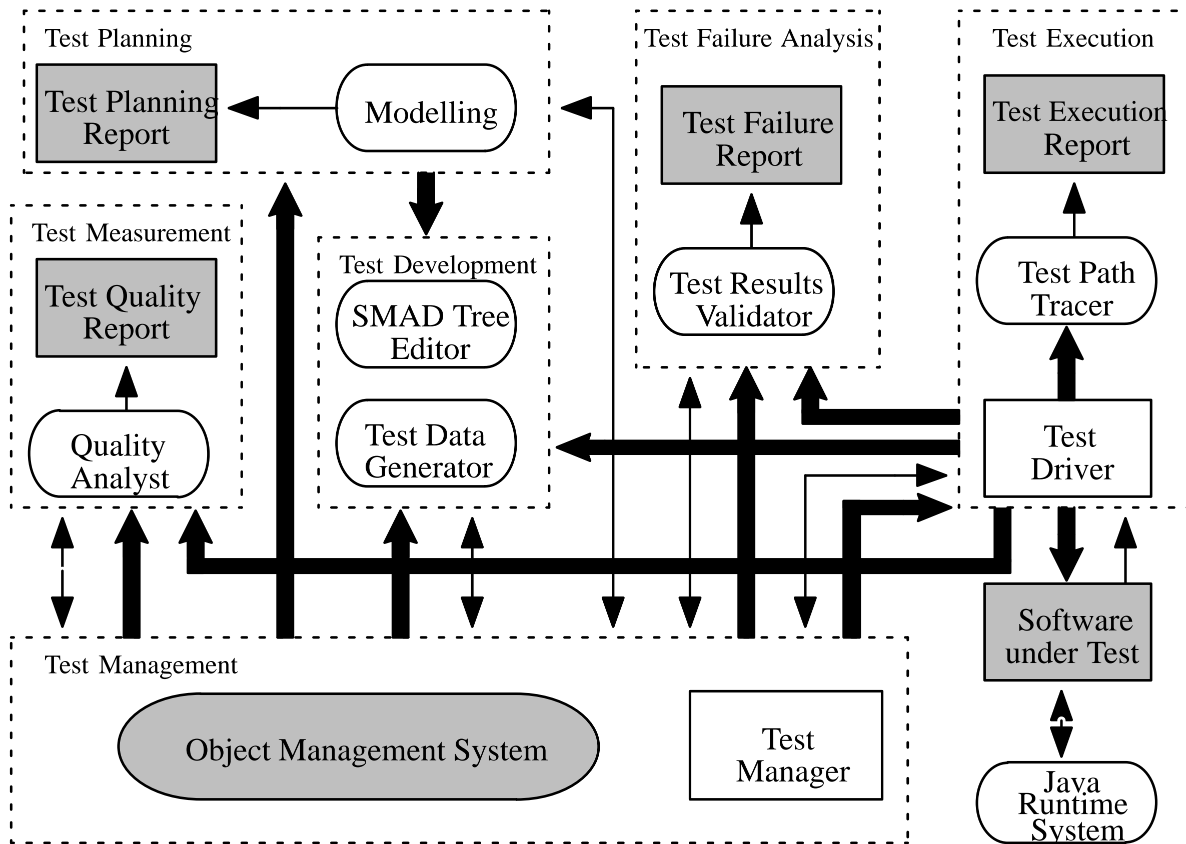

Figure 7: The SITE system structure and functional allocation through SAAM

5.3 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 [5]. A comparison of four STEs, PROTest II, TAOS, CITE and SITE, made by the SAAM is shown in Figure 8.

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

Figure 8: A comparison of four STEs by SAAM

However, test process focus was identified for each STE across the software development life cycle, shown in Figure 9.

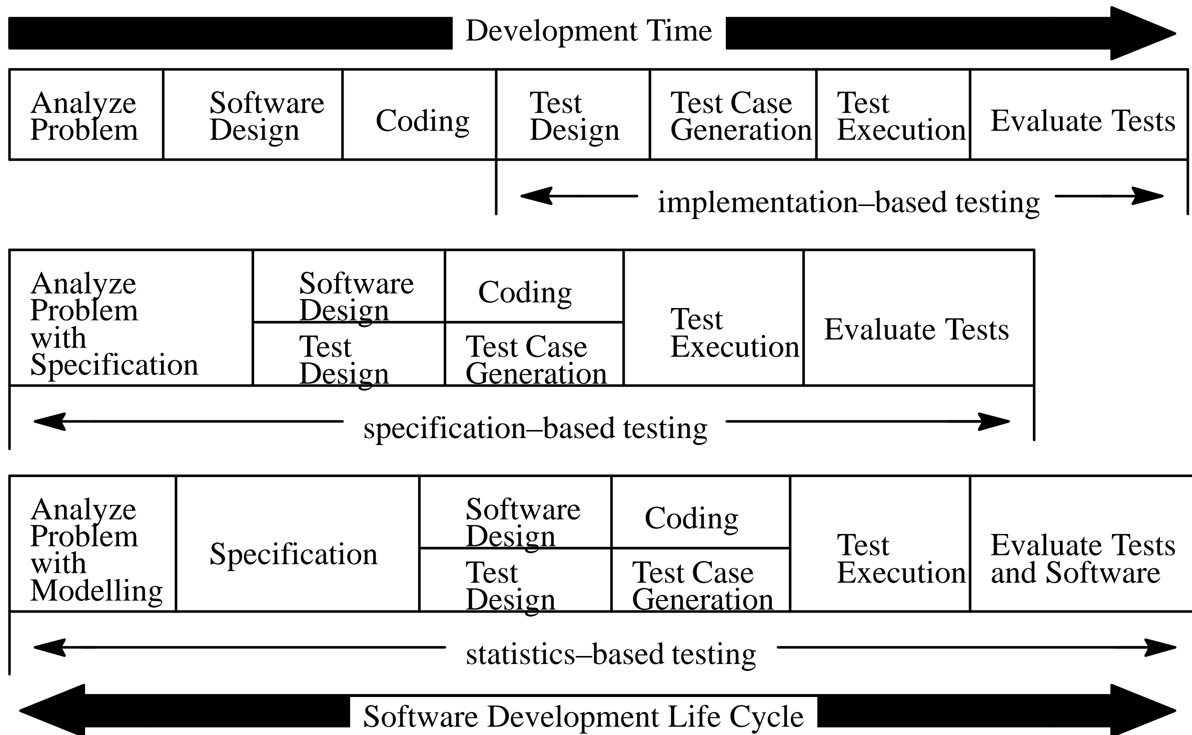

Figure 9: Test process focus appears the life cycle

- 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 can not be derived from the implementation. In this case, the

existence of an oracle (in the human mind) must be assumed and checking the test results against the oracles has to be done. With specification–based testing, both test input data and the expected outputs can be generated from a specification. The statistics–based testing is on the top of specification–based testing with the quality plan before specification.

6 CONCLUSION

The support of fully automated test environment for distributed applications have been a desired significant issue to the software development process. In this paper, an operational environment for testing distributed applications is proposed. An essential component for developing quality software is SITE in this operational environment. It consists of control components (test manager, test driver), computational components (Modeller, SMAD tree editor, quality analyst, test data generator, test paths tracer, test results validator, simulator) and an integrated database. 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 addresses the two crucial requirements, when to stop testing and how good the software is after testing, for software test and provides automated support for test execution, test development, test failure analysis, test measurement, test management and test planning.

ACKNOWLEDGEMENT

The authors would like to thank Dr C.K. Cho of Computa and Ms N. S. Eickelmann for their for their suggestions and corrections which were useful for improving the paper. The work of one author, H.D. Chu, is funded by the National Science Council in Taiwan from whom he received a fellowship to work toward a doctoral degree.

REFERENCES

[1] Belli, F. & Jack, O., Implementation–based analysis and testing of prolog programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, 70–80, Cambridge, Massachusetts, 1993.

[2] Cho, C. K., *Quality Programming: Deleloping and Testing Software with Statistical Quality Control*. John Wiley & Sons, Inc., New York, 1988.

[3] Chu, H., Dobson, J. & Liu, I., FAST: A Framework for Automating Statistics–based Testing. *Software Quality Journal*, 6, 13 – 36, 1997.

[4] Deutsch, M. S. & Willis, R. R., *Software Quality Engineering: A Total Technical and Management Approach*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1988.

[5] Eickelmann, N. S. & Richardson, D. J., An Evaluation of Software Test Environment Architectures. In *Proceedingd of the 18th International Conference on Software Engineering*, 353–364, Berlin, Germany, 1996.

[6] Hörcher, H. M. and Peleska, J., Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 4, 309–327, 1995.

[7] Kazman, R., Bass, L., Abowd, G. & Webb, W., SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*, 81 – 90. Sorrento, Italy, 1994.

[8] Myers, G. J., *The Art of Software Testing*. John Wiley & Sons, New York, 1978.

[9] Norman, S., *Software Testing Tools*. Ovum Ltd, London, 1993.

[10] Poston, R. M., *Automating Specification–Based Software Testing*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[11] Richardson, D.J., TAOS: Testing with Oracles and Analysis Suport. In *Proceedings of the International Symposium on Software Testing and Analysis*, 138–153, Seattle, Washington, 1994.

[12] Sommerville, I., *Software Engineering* (Fifth ed.). Addison–Wesley Publishing Company, Wokingham, England, 1996._

[13] Vogel, P. A., An Integrated General Purpose Automated Test Environment. In *Proceedings of the International Symposium on Software Testing and Analysis*, 61–69. Cambridge, Massachusetts, 1993.