An Evaluation of Software Test Environment Architectures

Nancy S. Eickelmann and Debra J. Richardson
Information and Computer Science Department
University Of California Irvine
Irvine, California 92715 USA
ike@ics.uci.edu and djr@ics.uci.edu

Abstract

Software Test Environments (STEs) provide a means of automating the test process and integrating testing tools to support required testing capabilities across the test process. Specifically, STEs may support test planning, test management, test measurement, test failure analysis, test development, and test execution. The software architecture of an STE describes the allocation of the environment's functions to specific implementation structures. An STE's architecture can facilitate or impede modifications such as changes to processing algorithms, data representation, or functionality. Performance and reusability are also subject to architecturally imposed constraints. Evaluation of an STE's architecture can provide insight into modifiability, extensibility, portability and reusability of the STE. This paper proposes a reference architecture for STEs. Its analytical value is demonstrated by using SAAM (Software Architectural Analysis Method) to compare three software test environments: PROTest II (Prolog Test Environment, Version II), TAOS (Testing with Analysis and Oracle Support), and CITE (CONVEX Integrated Test Environment).

1 Introduction

Software testing is a critical activity in the development of high quality software. When testing is performed manually it is highly error-prone, time consuming and costly. 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. Industrial use of STEs provide significant benefits by reducing testing costs, improving test accuracy, improving software quality, and providing for test reproducibility [17, 20].

Despite the critical importance of STEs in the development of quality software, rigorous evaluation of their capabilities has been largely ignored. Comparisons are frequently made among STEs using a taxonomic approach [6, 12, 14, 5, 18]. These illustrate whether a system supports a given task or has a specific attribute.

How well an STE meets its intended goals is usually drawn from textual descriptions. A uniform graphical depiction complements textual descriptions, thereby facilitating evaluation of developers claims concerning qualities of STEs. Examples of typical claims are:

> The rule-based approach offers many advantages over more traditional test execution systems...which require that each test case carry with it a script or a program to perform the necessary executions and results verification [20].

> Developing sophisticated testing capabilities through composition of less complex, more general components proved to be an extremely effective approach; it facilitates rapid prototyping and lower development costs for new tools as well as tool integration [17].

It has been demonstrated that some system qualities can be evaluated through an analysis of the system's software architecture, where an architecture consists of both system structure and functionality. A uniform graphical depiction of system architectures facilitates such an analysis [8,10].

The Software Architecture Analysis Method (SAAM) provides an established method for describing and analyzing software architectures [10]. SAAM is used in this paper to examine three STEs: PROTest II (Prolog Test Environment, Version II), TAOS (Testing with Analysis and Oracle Support), and CITE (CONVEX Integrated Test Environment). The terminology used to describe SAAM in this paper is consistent with that of [10], where SAAM is defined as consisting of the following steps:

1. Characterize a canonical functional partition for the domain.
2. Create a graphical diagram of each system's structure using the SAAM graphical notation.
3. Allocate the functional partition onto each system's structural decomposition.
4. Choose a set of quality attributes with which to assess the architectures.
5. Choose a set of concrete tasks which test the desired quality attributes.
6. Evaluate the degree to which each architecture provides support for each task, thus indicating satisfaction of the desired quality.

0270-5257/96 $5.00 © 1996 IEEE
Proceedings of ICSE-18

To accomplish this analysis, the SAAM method takes three perspectives: a canonical functional partition of the domain, system structure, and allocation of functionality to system structure.

The first perspective, a canonical functional partition of the application domain, may be provided by a reference architecture. If one is not available, a domain analysis can be done to determine the necessary partitioning of functionality.

In this paper, the three STEs are presented as originally published and then their system structure is recast in the SAAM architectural notation. Using a uniform notation permits a common level of understanding on which to base the analysis.

For each STE, the canonical functional partition is then allocated to its structural decomposition. The architecture can then be evaluated with respect to specific quality attributes. To ascertain if a system has a particular quality, concrete tasks are chosen to demonstrate the presence or absence of that quality in the STE. The analysis determines whether the architecture supports specific tasks or not in accordance with the qualities attributable to the system.

1.1 Overview

In section 2, the three architectural analysis perspectives used by SAAM are described. The canonical functional partition is a result of a domain analysis of the test process and test process automation. This results in a reference architecture for STEs. Then, the graphical notation that is used by SAAM to describe system structure is provided. Finally, the allocation of functionality to system structure is described. This section sets the groundwork for evaluating the architecture of STEs.

Section 3 describes each of the three STEs to be evaluated: PROTest II, TAOS, and CITE. Each is first described as originally diagramed and discussed by the authors and then recast in the graphical notation used by SAAM along with an allocation of the canonical functional partition.

Section 4 defines the quality of reusability of test artifacts analyzed in this paper. The test artifacts and the functional partition to which they are relevant are described. Each of the three STEs is then evaluated with respect to test artifact reusability.

Section 5 concludes with a summary of results and contributions as well as a discussion of future work.

Specific contributions of the paper include an initial reference architecture for Software Test Environments (STEs), a comparison of three STEs using the Software Architectural Analysis Method, an evaluation of architectural constraints on test artifact reusability, and further evaluation of SAAM as an analysis method for software architectures.

2 Architectural Analysis Perspectives

Using SAAM, architectural analysis is approached from three perspectives: a canonical functional partition, system structure, and allocation of functionality to system structure. This section describes these three perspectives.

2.1 Canonical Functional Partition

A canonical functional partition of the application domain provides the functional perspective required by SAAM. Reference architectures, such as the Arch/Slinky Meta-model for UIMSs (User Interface Management Systems) and the Toaster model for SDEs (Software Development Environments), provide functional characterizations for their domains. In mature domains such as these, a canonical functional partition has been accomplished through extensive domain analysis by researchers in the field.

Software Test Environments do not yet have a reference architecture to characterize their domain specific functionality. A domain analysis is needed to accomplish this. The domain analysis and resulting canonical functional partition for STEs requires that two aspects be evaluated: the specific testing activities inclusive in an ideal test process [9] and the functions specific to the automation of that process.

STEs pose a significant challenge for domain analysis. This is in part due to the rapid advances in test technology as well as the evolution of the test process that should be automated by an STE. Gelperin and Hetzel point out that over the years the test process has increased in scope across the life cycle and has been refocused as to its primary goals and objectives [9]. STEs have not all kept up with the state-of-the-art in test process or its automation. To describe the STE's process focus, we provide a history of test process evolution.

Test process evolution is shown in figure 1. The test process began with a debugging focus in the early 50's, where the process took an ad hoc approach of simply finding bugs. The test process then evolved to a demonstration period, which focused on making sure the program ran and solved the intended problem. This period was followed by the destruction period, which focused on executing a program to find failures through implementation-based testing. Next, an evaluation-oriented period addressed the integration of methods to provide evaluative support throughout the software lifecycle, as described in the Federal Information Processing Systems (FIPS) guidelines [13]. The evaluation process focuses on detecting require

ments, design, and implementation faults, which requires an early life cycle testing effort. The current test process is a life cycle prevention model, which focuses on fault prevention through parallel development and test processes. The change of emphasis has been from detection of faults to one of prevention of faults.

1950-1956 The Debugging-Oriented Period
1957-1978 The Demonstration-Oriented Period
1979-1982 The Destruction-Oriented Period
1983-1987 The Evaluation-Oriented Period
1988-1995 The Prevention-Oriented Period

Figure 1. Test process evolution, adapted from [9]

Our domain analysis evaluated the software test process, test process automation, and current STEs. The analysis resulted in a partitioning of the STE domain into six canonical functions: test execution, test development, test failure analysis, test measurement, test management, and test planning.

- ***Test Execution*** includes the execution of the instrumented source code and recording of execution traces. Test artifacts recorded include test output results, test execution traces, and test status.
- ***Test Development*** includes the specification and implementation of a test configuration. This results in a test suite, the input related test artifacts, and documentation. Specific artifacts developed include test oracles, test cases, and test adequacy criteria.
- ***Test Failure Analysis*** includes behavior verification and documentation and analysis of test execution pass/fail statistics. Specific artifacts include pass/fail state and test failure reports.
- ***Test Measurement*** includes test coverage measurement and analysis. Source code is typically instrumented to collect execution traces. Resulting test artifacts include test coverage measures and test failure measures.
- ***Test Management*** includes support for test artifact persistence, artifact relations persistence, and test execution state preservation. Test process automation requires a repository for test artifacts. A passive repository such as a file serves the basic need of storage. However, an active repository is needed to support relations among test artifacts and provide for their persistence.
- ***Test Planning*** includes the development of a master test plan, the features of the system to be tested, and detailed test plans. Included in this function are risk assessment issues, organizational training needs, required and available resources, comprehensive test strategy, resource and staffing requirements, roles and responsibility allocations, and overall schedule.

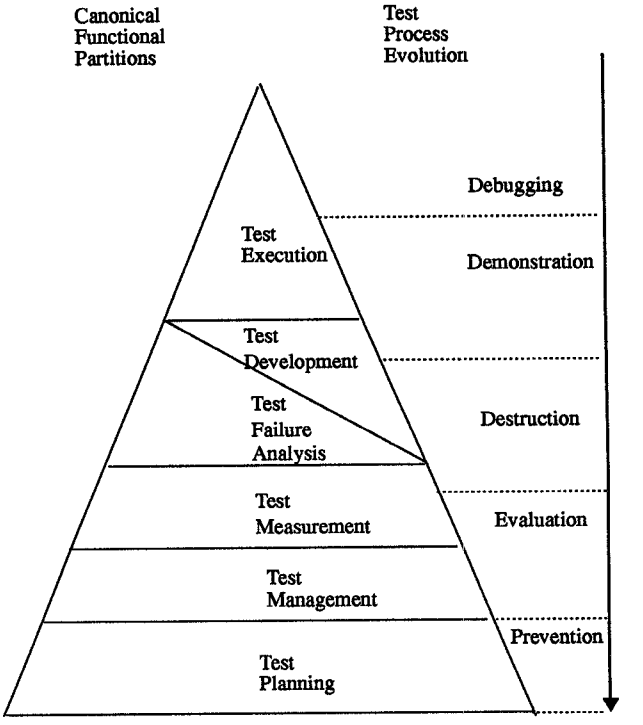

Figure 2. STEP Model

The test process evolution and canonical functional partition resulting from the STE domain analysis provide the foundation for the Software Test Environment Pyramid (STEP) model. The STEP model, shown in figure 2, stratifies test functionalities from the apex of the pyramid to its base in a corresponding progression of test process evolution as described in figure 1. Each period represented in the pyramid includes the functionalities of previous periods as you descend from the apex to the base.

The top section of the pyramid represents the function of test execution. Test execution is clearly required by any test process. The test process focus of the debugging-oriented period was solely on test execution.

The second segment of the pyramid, from the top, is divided into two scalene triangles. The smaller scalene triangle represents test development. The larger scalene triangle represents test failure analysis. The relative positions and sizes have semantic significance.

Test development played a more significant role to the overall test process during the demonstration-oriented and destruction-oriented periods due to the manual intensive nature of test development at that time. Test development methods have not significantly changed, although they have improved in reliability and reproducibility with automation. Thus, their role in test process has diminished in significance as you move ahead in test process evolution.

Test failure analysis was less important when performed manually, as interactive checking by humans added little benefit for test behavior verification. The methods applied to test failure analysis have increased in their level of sophistication, making test failure analysis more significant to the overall test process. One of the most significant advances are specification-based test oracles which enables early entry of the test process into the life cycle. This is a key difference in test process focus as it progresses towards the prevention-oriented period.

Test measurement is represented by the third segment in the pyramid. Test measurement is required to support the evaluation-oriented period, which represents the point of departure from a phase approach to a life cycle approach. A significant change in the test process focus is that testing is applied in parallel to development, not merely at the end of development. Test measurement also enables evaluating and improving the test process.

Approaching the base of the pyramid, the fourth segment represents test management, which is essential to the evaluative test process due to the sheer volume of information that is created and must be stored, retrieved, and reused. Test management is critical for test process reproducibility.

The base, or foundation, of the pyramid is test planning. Test planning is the essential component of the prevention-oriented period. Test planning introduces the test process before requirements, so that rather than being an afterthought, testing is preplanned and occurs concurrently with development.

The Software Test Environment Pyramid reference architecture presented here is an initial attempt at a diagrammatic representation of the canonical functional partition of the Software Test Environment domain. We will continue to refine and improve the STEP model and welcome other researchers to do the same. To avoid overuse of the term architecture, canonical functional partition is used in the remainder of the paper in deference to reference architecture.

2.2 SAAM Structure

Structure represents the decomposition of the system components and their interconnections. The graphical notation used by SAAM is a concise and simple lexicon, which is shown in figure 3. In the notation used by SAAM there are four types of components: 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 (database). There are two types of connectors: control flow and data flow, either of which may be uni or bi-directional.

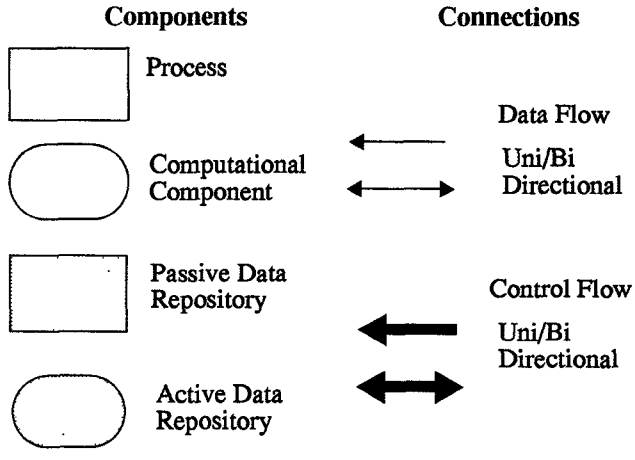

Figure 3. SAAM graphical architectural notation

SAAM's goal is to provide a uniform representation by which to evaluate architectural qualities. The simplicity of the notation is achieved by abstracting away all but a necessary level of detail for a system level comparison. The field of software architecture, not unlike hardware design, recognizes a number of distinct design levels, each with its own notations, models, componentry, and analysis techniques. A more detailed and complex lexicon would be required for a more focused analysis. This is not within the scope or objectives of this paper but is planned for future work.

2.3 Allocation of Functional Partitions to Structure

The allocation of domain functionality to the software structure of an STE completes the graphical representation of the STE. The allocation provides the mapping of a system's intended functionality to the concrete interpretation in the implementation. SAAM focuses most closely on allocation as the primary differentiating factor among architectures of a given domain.

3 Architectural Descriptions of STEs

In this section we evaluate three test environments from the domain of Software Test Environments.

We first duplicate the system level graphical diagram for each STE as published by their respective authors. We discuss components in this diagram briefly. The reader is referred to the authors' papers for detailed information. Next, we recast each STE in the graphical notation used by SAAM.

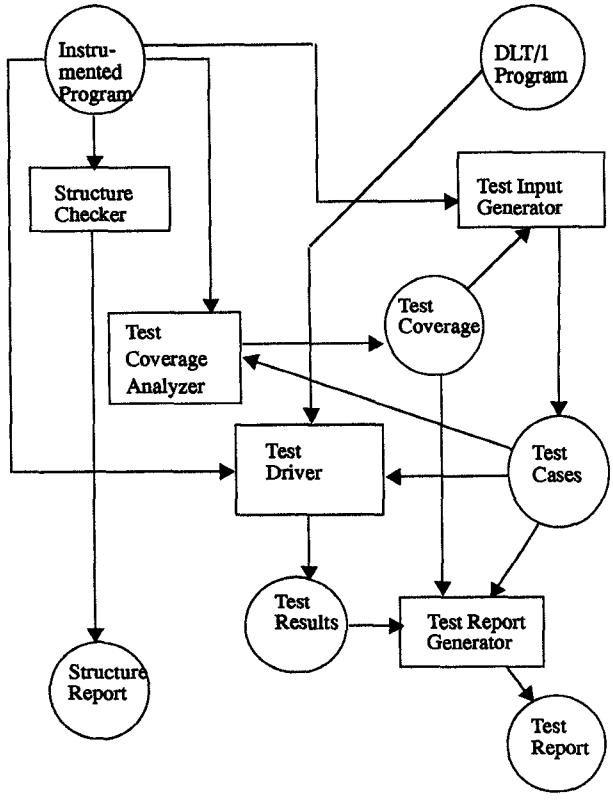

Figure 4. The PROTest II system, adapted from [2]

Here, we name components of all three systems of similar functionality the same and indicate what components in the authors' original descriptions correspond to these like-named components. The re-characterization of the architectures in a uniform, evenly abstract graphical diagram and common terminology provides a sound basis for understanding and comparison. Each STE is evaluated in terms of its stated goals and compared with the STEP model to determine its process focus. Finally, the three STE architectures are compared as represented in the notation used by SAAM.

3.1 PROTest II

PROTest II Author's Description. The PROTest II system, as depicted by [2], is shown graphically in figure 4. PROTest II is composed of five components: a test driver, structure checker, test input generator, test coverage analyzer, and a test report generator. PROTest II also provides a graphical user interface. A detailed textual description of the system can be found in [2].

PROTest II SAAM Description and Functional Allocation. As shown in figure 5, PROTest II includes three of the canonical functional partitions: test execution, test development, and test measurement. Test failure analysis, test management, and test planning are missing, which is often typical of interactive test environments [11].

The SAAM re-characterization of the PROTest II architecture facilitates determining if system goals are achieved. In particular, a major deficiency highlighted is

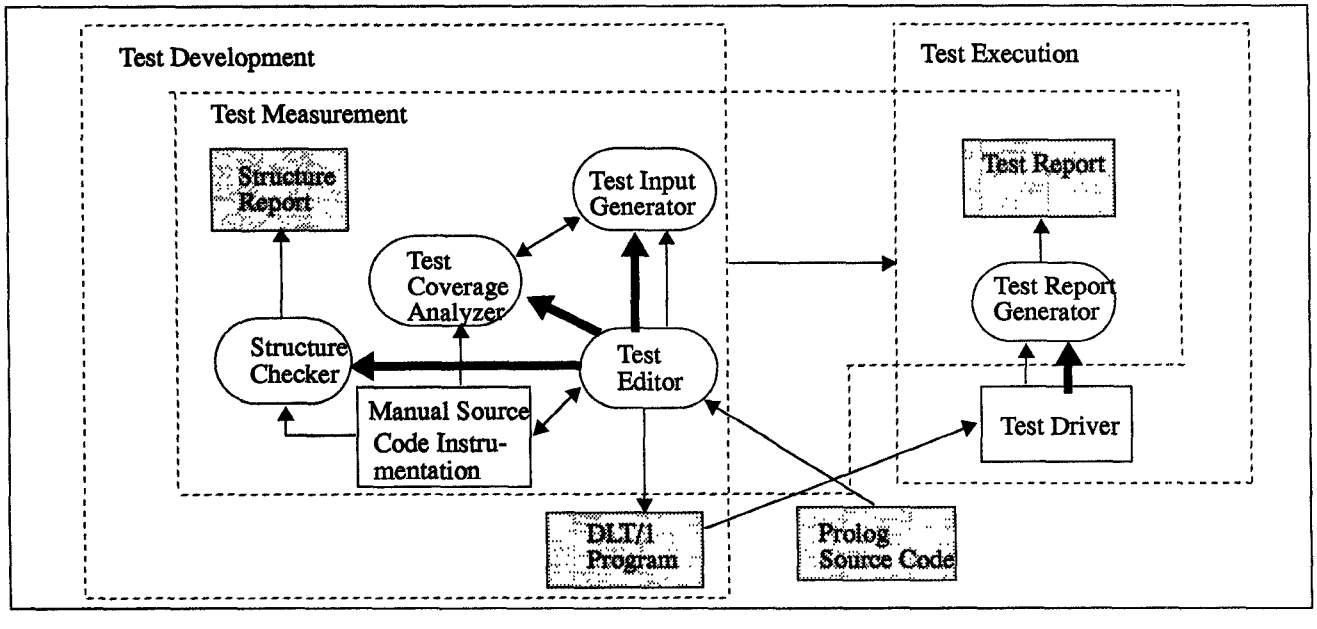

Figure 5. The PROTest II system structure and functional allocation through SAAM.

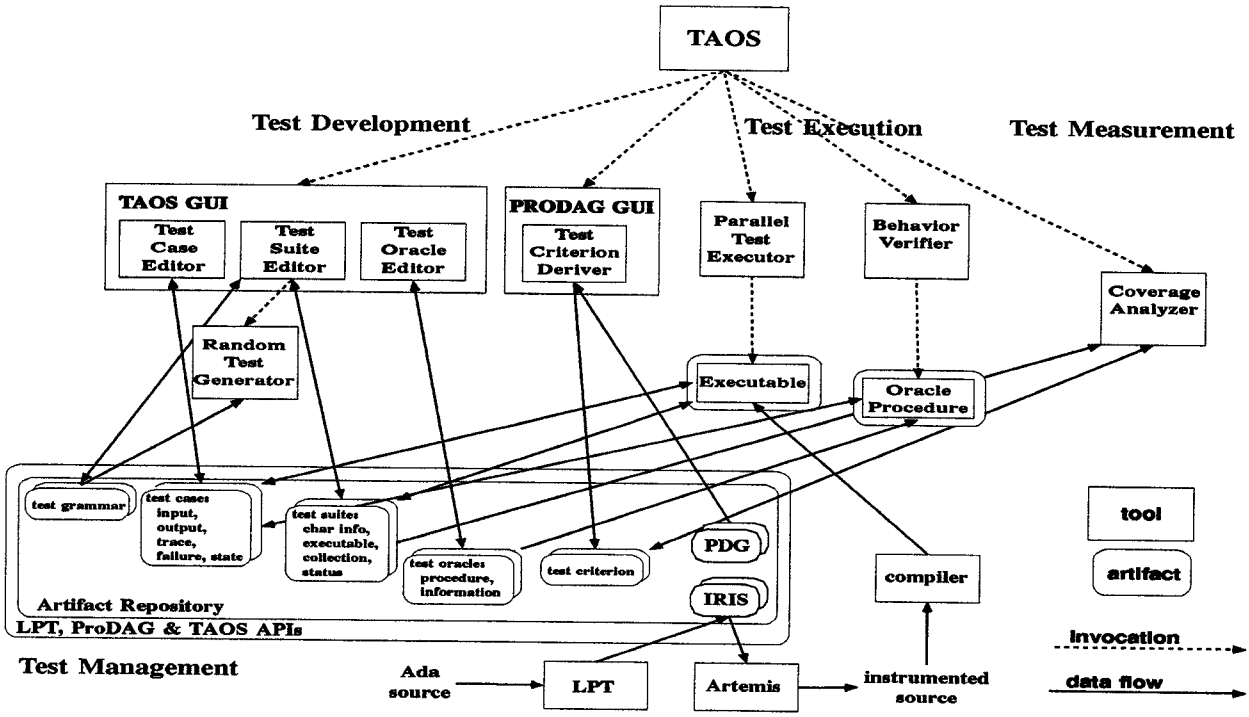

Figure 6. The TAOS system, source [17]

that the PROTest II architecture does not have structural separation between the functions of test development and test measurement. The test input generation process is interdependent with the test coverage process. This is not evident in the author's diagram but is explicit in the SAAM architectural description. PROTest II accomplishes the function of test measurement by analyzing instrumented source code to determine what clauses will be covered given the test cases that are chosen. The predicted coverage of clauses is reported along with the test cases that did not execute (called the failed to execute report). These structural choices create interdependencies of test measurement with test development and test execution. The resulting architecture has highly coupled components, as is clear in the functional overlays in figure 5.

The objective of PROTest II is to provide a fully automated test environment for Prolog programs [2]. PROTest II does not achieve this goal but rather provides an interactive test environment that partially automates test execution, test development, and test measurement. The developers state that the strength of PROTest II as a system stems from the idea of defining coverage in real logic programming terms, rather than adapting imperative programming ideas. It appears that this language focus led to coupling of components, since generic capabilities were not developed.

PROTest II Process Focus. The SAAM diagram of PROTest II highlights the system's structural composition. It also provides a basis to compare the STE to the process evolution of the STEP Model. The PROTest II system supports implementation-based testing; thus, the life cycle entry point for PROTest II is post-implementation. The PROTest II system supports executing a program to cause failures, and thus the test process has a destructive focus [9]. PROTest II does not support the test process across the development life cycle. However, the PROTest II STE is part of a larger system, mentioned but not described in [2], which may extend its range of support.

3.2 TAOS

TAOS Author's Description. The TAOS system, as depicted by [17], is shown graphically in figure 6. TAOS is composed of several components: a test generator, test criterion deriver, artifact repository, parallel test executor, behavior verifier, and coverage analyzer. TAOS also integrates several tools: ProDAG (Program Dependence Analysis Graph), Artemis (source code instrumentor), and LPT (Language Processing Tools). The TAOS system is shown in figure 6. A complete detailed textual description of TAOS and its integration with the Arcadia SDE is given in [17]. Specification-based test oracles as automated by TAOS are described in [16].

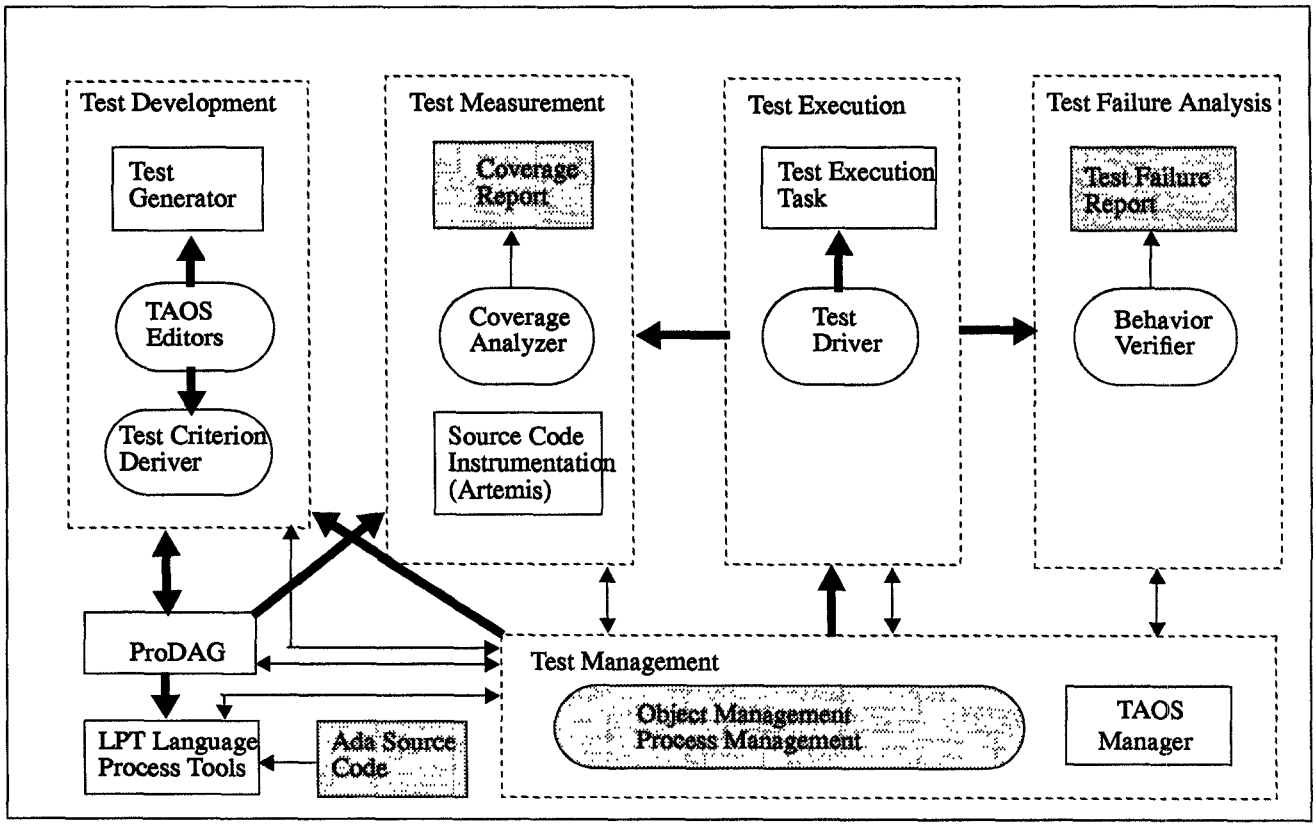

Figure 7. The TAOS system structure and functional allocation through SAAM

TAOS SAAM Description and Functional Allocation. As shown in figure 7, TAOS provides support for test execution, test development, test failure analysis, test measurement, and test management. Test planning, however, is not supported.

As evident in the SAAM architectural description, TAOS achieves loosely coupled components through separation of concerns with respect to system functionalities. The composition of each functional partition is accomplished through the use of highly cohesive components. Test generation is a process with an independent thread of control and is not dependent on components from other functional areas. Test execution is achieved by an independent process for program execution. Test failure analysis and test measurement are invoked by the test driver, but remain highly cohesive componentry within their respective functional partitions.

This high degree of separation of concerns is facilitated by the primary component integration mechanism, which is data integration. Data integration is facilitated by abstract programmatic interfaces provided by the PLEIADES object management system, which supports persistence of data and data relations in an active repository [19].

The TAOS system was built using a development strategy identified during the TEAM project [4] with the goal of building components that provide generic capabilities and abstract interfaces to support varied testing activities and processes. TAOS integrates several multipurpose components, including ProDAG, LPT, and Artemis. Effective data integration is achieved through a consistent internal data representation in the Language Processing Tools (LPT), which is also used by ProDAG and Artemis. Integration of generic analysis tools supports multiple test and analysis purposes. ProDAG, for instance supports the use of program dependence graphs for test adequacy criteria, test coverage analysis, and optimizing efforts in regression testing (by identifying code affected by program changes). The TAOS architecture clearly achieves its goal through generic capabilities and abstract programmatic interfaces.

TAOS Process Focus. TAOS supports specification-based testing and thus an entry point into the life cycle after the requirements phase. TAOS supports a test process with an evaluative focus.

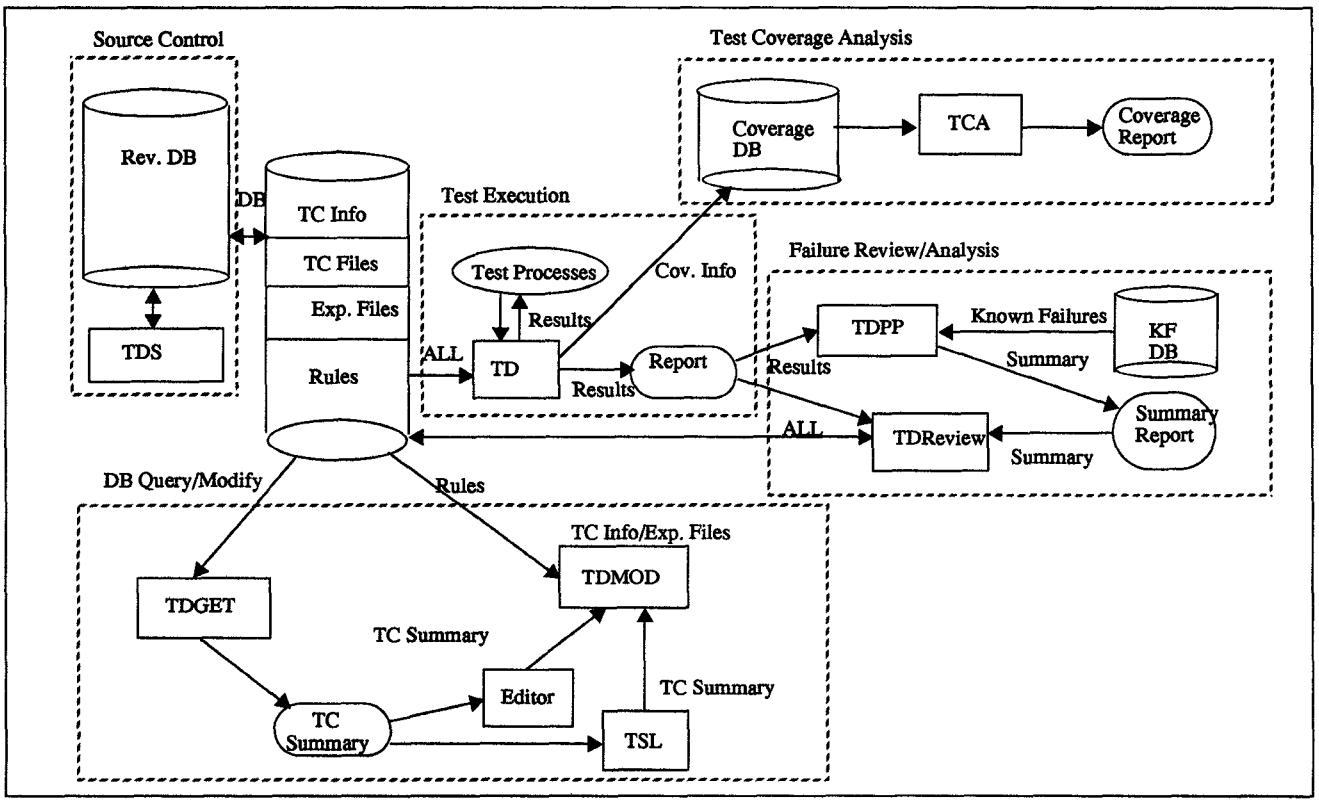

Figure 8. The CITE system, adapted from [20]

3.3 CITE

CITE Author's Description. The CITE system, as depicted by [20], is shown graphically in figure 8 (only the components of CITE developed by Convex are shown). The CITE architecture consists of several components: a test driver (TD), test coverage analyzer (TCA), test data reviewer (TDReview and TDPP), test generator, and databases and a rule base. Additional tools have been integrated for use with the system. Specifically, the GCT test coverage tool expands the types of test coverage measured by the environment and the EXPECT simulator provides for special purpose interactive testing. A complete textual description of CITE is given in [20].

CITE SAAM Description and Functional Allocation. The SAAM graphical depiction of CITE, as shown in figure 9, reflects the structural separation desired in a system that controls the test process automatically. CITE provides support for test execution, test development, test failure analysis, test measurement, and test management. CITE does not support test planning.

CITE has achieved the desired functionalities through allocation to structurally cohesive components. The process control exercised by the rule-base is evidenced in direct control to each functional area. This clean separation of concerns supports the primary goal of a fully automated, general purpose software test environment.

CITE Process Focus. CITE supports implementation-based testing. CITE has a destructive test process focus that is, it executes a program to detect failures, which is a limited scope for automation of the test process. However, all testing functionalities supported by CITE are fully automated.

3.4 STE Comparison

The three STEs share stated goals of automating the test process. The use of SAAM clarifies how well each STE achieves this goal and to what degree. SAAM uses 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.

Test process focus was identified for each STE. PRO-Test II and CITE support implementation-based testing and have a destructive test 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 de-

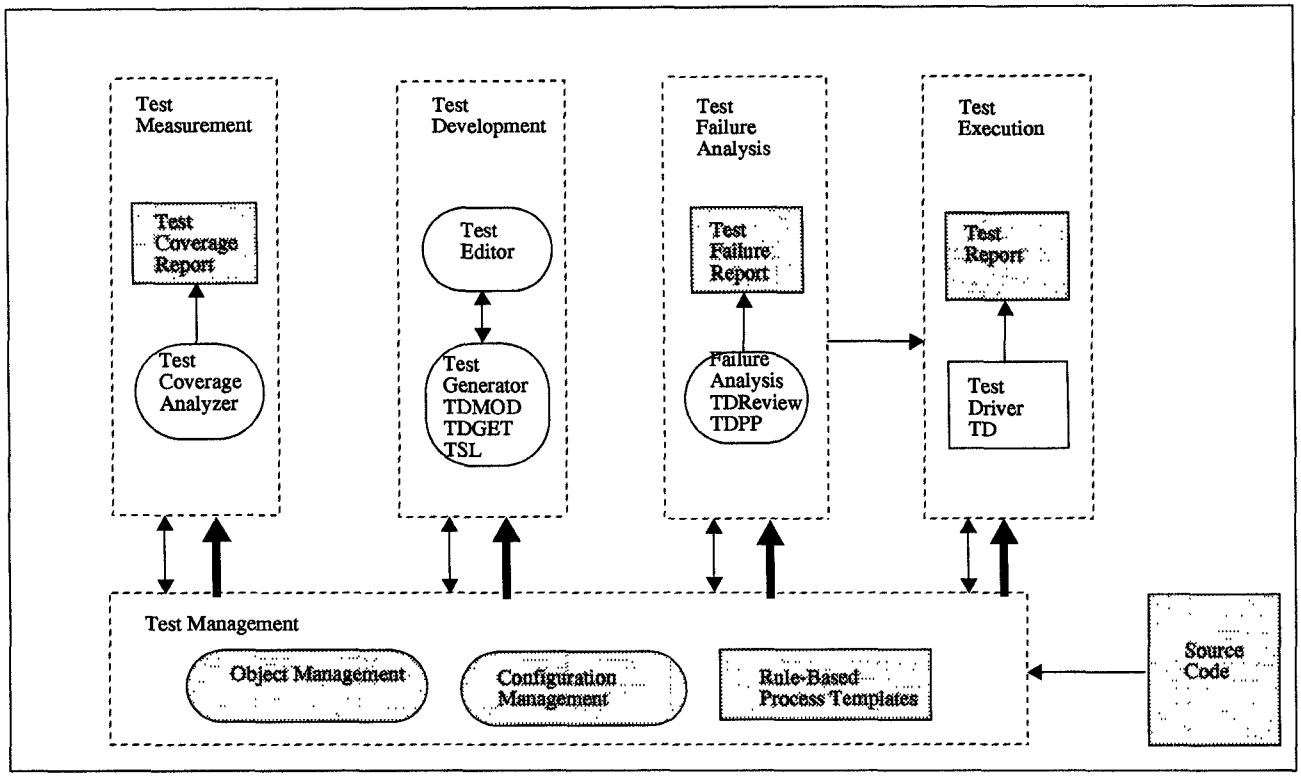

Figure 9. The CITE system structure and functional allocation through SAAM

tection extends from requirements and design to code. Test process focus appears to dictate the scope of automation achieved across the life cycle and the types of functionality supported by an STE.

Test planning is not supported by any of the three STEs. Test planning functionality specifically supports a preventative test process focus. We could not find an STE that explicitly supports a preventative process. Some tools address planning as a separate issue but fail to integrate test artifacts with a parallel development effort. A preventative process focus may be best achieved through integrating an STE with an SDE.

Test Management is not supported in PROTest II. CITE and TAOS support test management with an active repository. TAOS provides support for object management and process management. CITE provides support for object management, rule-based test process templates and configuration management. Test management is essential for full automation. The test process spans a temporal boundary of the life cycle and generates voluminous amounts of test artifacts with complex relations that must be supported by test management. Coordinating the test process over time is not possible without automated test management.

Test Measurement in PROTest II is tightly coupled with test development and test execution. PROTest II only partially automates measurement, specifically source code instrumentation is done manually. PROTest II addresses test measurement and test development by focusing on the language constructs of Prolog. The resulting monolithic, highly coupled architecture reflects this focus. CITE and TAOS provide for full automation of test measurement. CITE supports block coverage and TAOS supports program dependence coverage as well as statement and branch coverage.

Test Failure Analysis requires that the expected output be documented for test case inputs. The actual outputs must be verified with respect to correct or expected behavior. Verification can be accomplished through a test oracle. The most accurate of oracles are specification-based test oracles; the most error prone are interactive human oracles. Test failure analysis has increased in importance as the test process focus has changed from demonstrative to evaluative; it is also critical in achieving a preplanned preventative process. PROTest II does not support failure analysis, as it does not store expected outputs; rather behavior verification relies on an interactive human oracle. CITE stores expected outputs with test case inputs, yet this

approach still relies on a human as oracle. The process is not interactive, however, and is thus less error prone. CITE provides additional leverage for the human oracle in that stored expected outputs are reused extensively adding to the confidence that they are correct. TAOS supports specification-based test oracles, which provide the greatest degree of oracle accuracy and reproducibility.

Test development is partially automated in PROTest II. It is an interactive process that manually develops an augmented Prolog program for test execution, manually instruments the source code, and manually checks the structure report. CITE and TAOS provide full automated support for test development including test data generation.

Test execution is fully automated by all three STEs.

All three STEs share the goal of providing a fully automated software test environment. PROTest II does not achieve this goal. The author's claim their focus on language constructs is a strength despite the architectural implications in lack of support for separation of concerns and development of highly cohesive components. TAOS and CITE have highly cohesive components that are loosely coupled. TAOS clearly achieves its goal of full automation and provides broader support across the life cycle with an evaluative process focus. Test failure analysis is optimized in TAOS through the use of specification-based test oracles. CITE meets its goal of full automation. A particular strength of the CITE STE is its provision for automated configuration management (CM) and versioning control. Automated CM is foundational to successful process control.

4 Analyzing Software Architectural Qualities

The goal of a software development process is to produce software having a specified set of qualities, hopefully quantifiable. The eleven ...ilities of software quality, delineated by [3], are correctness, reliability, efficiency, integrity, usability, maintainability, flexibility, testability, portability, reusability and interoperability. These qualities are also desirable for Software Test Environments.

Software architectural analysis is used to determine if a specific structural decomposition and the functional allocation to system structures supports or impedes certain qualities. Parnas identified changes to data representation and changes to processing algorithm as particularly sensitive to system architectural constraints [15]. Garlan, Kaiser, and Notkin further identified enhancements to system functionality, improvements to performance (space and time), and reuse of components [7]. This paper examines software architectural constraints in relation to reusability. An examination of system modifications, functional enhancements, and performance improvements are to be examined in future work.

Component reusability is the attribute of facilitating multiple and/or repeated use of components. This applies to both computational and data components. With respect to STEs, we might evaluate reuse of the components in the STE or reuse of the components managed and manipulated by the STE. Here, we choose the latter, and thus will evaluate each STE with respect to its support for reuse of the artifacts of the test process managed by the STE. An STE's ability to store, retrieve, and reuse test plans, test scripts, test suites, test cases, test criteria, test oracles, test results, and the like is central to their ability to effectively automate the test process. Note that some of these artifacts are indeed computational components (e.g., test scripts) while others are data components.

Reuse is further delineated by [Bie91] as verbatim or leveraged reuse. Verbatim reuse is reuse without modification to the code. Leveraged reuse is reuse through modifying some portion of the code. Verbatim reuse has the clear advantage of reducing test effort. Leveraged reuse is of value as long as it takes less effort to locate, select, and modify the test artifact, than to simply reproduce it.

The next section evaluates test artifact reusability for each of the three STEs. The presence or absence of the quality of reusability in PROTest II, TAOS, and CITE is evaluated in regards to the STE's support for test artifact reusability.

4.1 Reusability of Test Artifacts in PROTest II.

PROTest II does not support test artifact reuse. The lack of provision for test management is a primary factor in this deficiency. Test management is necessary to support artifact persistence and thus reuse.

The structure of PROTest II also impedes test artifact reusability. Test inputs are generated based on test coverage. Test execution results provide clauses actually covered in the execution. Failed cases are test cases that did not execute. This functional interdependency among structural components inhibits test artifact reuse. The high degree of artifact interdependence impedes reuse as well.

4.2 Reusability of Test Artifacts in TAOS.

TAOS supports test artifact persistence and maintains test artifact relations through an active object repository. TAOS test management support provides the ability to store, retrieve, and modify test artifacts. Thus, TAOS adequately supports test artifact reuse.

Primarily, TAOS provides support for verbatim reusability. As an example, ProDAG provides dependence graphs for many uses, including derivation of test adequacy criterion and test coverage analysis. The same

dependence graphs can be reused in debugging, where code slices leading to a failure can be determined. These same graphs can be used later in the life cycle to facilitate software maintenance, where code slices affected by a change can be determined, analyzed and related test artifacts reused for regression testing. Thus, we see that TAOS provides for reuse in diverse functional capacities as well as in disbursed temporal contexts.

TAOS also provides support for leveraged reuse as persistent test artifacts may be modified for use in slightly different contexts, such as during software evolution and maintenance.

4.3 Reusability of Test Artifacts in CITE.

CITE provides extensive support for leveraged reusability. Leveraged reuse in CITE includes reuse of test rules and test templates. The rules may be reused with slight modifications to execute tests including new capabilities. Test information can be modified for reuse by editing templates. As an example, Convex discusses the reuse of the compiler test suites for a family of related compiler products, where in each instance the rule required a simple modification allowing 90% of the test suite to be reused.

Verbatim reuse in CITE includes the reuse of entire test suites as well as individual test cases.The test management provides support for version control, which supports reusing tests during maintenance and updates of previously released or tested code.

5 Conclusions and Future Work

This paper laid the groundwork for future evaluation of software testing environments by developing the STEP Model for STEs, validating the model by representing three different systems using the architecture, comparing three STEs using SAAM, and evaluating the architectural constraints for one quality attribute, test artifact reusability, for all three STEs. This work has also provided insight into the application of SAAM for architectural analysis of STEs.

STEP Model. The Software Test Environment Pyramid (STEP) model was introduced in section 2 as an initial attempt at providing a much needed canonical functional partition for STEs in a semantically significant model that incorporates process evolution and focus. The six functions identified - test execution, test development, test failure analysis, test measurement, test management, and test planning - provide orthogonal partitions for all STE functionality. The pyramid model provides semantic significance in the ordering of the sections through the test process evolution progression from apex to base. This model is unique in its incorporation of both functionality and test process evolution. The STEP model implicitly recognizes the importance of test process focus for STEs. STEP also provides insight into an STE's range of life cycle support in the test process focus. The STEP model's integration of process evolution recognizes the importance of test process automation as well as its evolutionary nature in applicability across the life cycle. The STEP model attempts to capture the complexity of test process automation in a semantically rich model with intuitive explanatory powers.

Comparison of Three STEs. PROTest II, TAOS, and CITE all share the goal of a fully automated test environment. Claims made by the respective authors of the STEs were: the PROTest II focus on declarative language constructs would provide an advantage for such a system, CITE was the most powerful and complete automated test environment in use, and TAOS provided an automated STE based on integrated tools and generic component functionality.

PROTest II provides a partially automated environment. CITE is indeed powerful and complete, although automated oracle support for test failure analysis could be improved. A particular strength of CITE is its provision for configuration management and versioning control. TAOS achieves its goal of a fully automated test environment and covers test activities across the life cycle. A significant strength of TAOS is its provision for automated specification-based test oracles to support test failure analysis.

Test Artifact Reusability. Test artifact reusability is of particular importance for STEs. Reuse of test artifacts with little or no modification is critical to providing full automation across the test process and the software life cycle. Reuse is also an effective determinant for delimiting the amount of information that must be stored and therefore managed by an STE. To support artifact reuse, an STE must provide test management utilities, specifically persistence of test artifacts and test artifact relations. Persistence alone, however, is not sufficient to support leveraged reusability. The artifacts must be configured to enable efficient storage, retrieval, and modification for leveraged reuse. If this capability is not supported, leveraged reuse becomes more costly and possibly prohibitive.

This paper examined PROTest II, TAOS, and CITE for the quality of reusability, in particular test artifact reusability. PROTest II does not support such reuse. TAOS and CITE support both verbatim and leveraged reuse. Support for test artifact reuse serves as the litmus test for an STE's support for full automation of the test process across the life cycle.

Evaluation of SAAM Method. The SAAM architectural analysis is approached from three perspectives: a canonical functional partition, system structure, and allocation of functionality to structural decomposition of the

system. SAAM uses a simple lexicon to describe all the STEs in a uniform notation to depict their system structure so as to permit a common level of understanding. The systems are further evaluated in their support for some concrete task that would determine whether or not the system has a quality such as modifiability or reusability.

SAAM enabled our analysis of all three STEs and revealed system strengths and weaknesses inherent to structural decompositions and choices concerning functional allocations. The differences noted were not apparent from the original author's descriptions but were revealed by using SAAM.

A more detailed lexicon in SAAM would support more in-depth analysis of test artifact reusability. The SAAM lexical notation supports analysis of data and control flow among system components but does not address complex data relationships. Future work will look at augmenting the SAAM lexical notation to provide greater detail and support complex analysis of data components.

Future Work. The STEP model reference architecture for STEs is to be further refined. Specific improvements planned are adding a third dimension to the pyramid and providing the required semantic significance to more closely correlate the canonical function partitions to test process evolution.

Additional STEs will be compared to the current work and evaluated with SAAM. This will provide additional insight into STEs beyond that gained by taxonomic approaches.

The three STEs analyzed in this paper will be further evaluated for architectural impact on enhancements to system functionality. Such analysis is supported by SAAM and will be the next area of investigation. In addition, system modifiability as evidenced in changes to data representation, processing algorithms, and performance optimization will be examined. However, SAAM's simple lexicon does not support such analysis and will therefore require a more detailed lexical notation be developed.

Acknowledgment

Thank you to the referees for their insightful suggestions and to Peter Vogel for his corrections to the CITE diagrams. This work was sponsored in part by the Air Force Material Command, Rome Laboratories, and the Advanced Research Projects Agency under Contract # F30602-94-C-0218 and the University of California MICRO program and Hughes Aircraft Company under Grant #94-105. The content does not necessarily reflect the position or policy of the U.S. Government, the University of California, or Hughes Aircraft Company, and no official endorsement should be inferred.

References

[1] J. M. Beiman. "Deriving Measures of Software Reuse in Object Oriented Systems." In Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, pages 63-83, South Bank University, London, May 5, 1991.

[2] F. Belli and O. Jack. "Implementation-based analysis and testing of prolog programs." In the Proceedings of the 1993 International Symposium on Software Testing and Analysis, pages 70-80. Cambridge, Massachusetts, June 1993.

[3] J.P. Cavano and J.A. McCall. "A framework for the measurement of software quality." In the proceedings of the Software Quality and Assurance Workshop, pages 133-139. November 1978.

[4] L. A. Clarke, D.J. Richardson, and S. J. Zeil. "Team: A support environment for testing, evaluation, and analysis." In Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments, pages 153-162. November 1988. Appeared as SIGPLAN Notices 24(2) and Software Engineering Notes 13(5).

[5] Daybreak Technologies Inc., Data sources software products guide.. Cherry Hill, New Jersey, Data Sources Inc., 1990.

[6] R. A. Fairley. Software testing tools. In Computer Program Testing, New York: Elsevier North Holland, 1981.

[7] D. Garlan, G. Kaiser, and D. Notkin. "Using tool abstraction to compose systems." IEEE Computer, vol. 25, June 1992.

[8] D. Garlan and M. Shaw. "An introduction to software architecture.". Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Co., 1993.

[9] D. Gelperin and B. Hetzel. "The growth of software testing." Communications of the ACM, 31(6):687-695, June 1988.

[10] R. Kazman, L. Bass, G. Abowd, and M. Webb. "SAAM: A method for analyzing the properties of software architectures." In Proceedings of the Sixteenth International Conference on Software Engineering, 81-90, Sorrento, Italy, May 21, 1994.

[11] G. J. Myers. The art of software testing. New York, John Wiley and Sons, 1978.

[12] E. Miller. Mechanizing software testing. TOCG Meeting, Westlake Village, California, April 15, 1986.

[13] Guideline for Lifecycle Validation, Verification, and Testing of Computer Software. National Bureau of Standards Report NBS FIPS 101. Washington, D.C., 1983.

[14] T. Nomura. "Use of software engineering tools in Japan.". In Proceedings of the Ninth International Conference on Software Engineering, 449, Monterey, California, March 1987.

[15] D.L. Parnas. "On the criteria to be used in decomposing systems into modules."Communications of the ACM, 15(12):1053-1058, December 1972.

[16] D.J. Richardson, S.L. Aha, and T.O. O'Malley. "Specification-based test oracles for reactive systems." In Proceedings of the Fourteenth International Conference on Software Engineering, 105-118, Melbourne, Australia, May 1992.

[17] D. J. Richardson. "TAOS: Testing with Oracles and Analysis Support." In Proceedings of the 1994 International Symposium on Software Testing and Analysis, pages 138-153. Seattle, Washington, August 1994.

[18] Software Quality Engineering. Survey of Software Test Practices. Jacksonville, Florida, Software Quality Engineering 1988.

[19] P. Tarr and L.A. Clarke. "An Object Management System for Software Engineering Environments." In ACM SIGSOFT '93: Proceedings of the Symposium on the Foundations of Software Engineering, Los Angeles, California, December 1993.

[20] P. A. Vogel. "An integrated general purpose automated test environment." In the Proceedings of the 1993 International Symposium on Software Testing and Analysis, pages 61-69. Cambridge, Massachusetts, June 1993.