The *MOTHRA* Software Testing Environment*

Richard A. DeMillo†
Eugene H. Spafford

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
+1 404 894-3180

ABSTRACT

The value of software testing in the development of large software systems is well-documented. Unfortunately, the development and employment of an integrated test plan is often avoided due to the costs associated with testing. These costs include more than just capital expenses associated with obtaining test systems and software. They also include the time and effort involved in educating personnel in the use of the testing system, the time taken to run the tests, and the costs of rerunning the tests after errors are found and corrected. Furthermore, some forms of testing are difficult or impossible to run incrementally, and they produce results which may be difficult to use in correcting or enhancing the tested software.

The *MOTHRA* Environment is an integrated set of tools and interfaces that support the planning, definition, preparation, execution, analysis and evaluation of tests of software systems. The support provided by *MOTHRA* is applicable from the earliest stages of software design and development through the progressively later stages of system integration, acceptance testing, operation and maintenance. *MOTHRA* has been designed to address some of the cost concerns mentioned above. Two primary design criteria, in particular, are significant in this regard. First, the *MOTHRA* interfaces—particularly user interfaces—are high-bandwidth. This allows us to present more information during testing and retesting. Coupled with proper design and integration with familiar displays, it should obviate the need for extensive training to use *MOTHRA*.

Secondly, the overall *MOTHRA* architecture imposes no *a priori* constraints on the size of the software systems that can be tested in the environment. The practical meaning of this criterion is that the same architecture is able to service programs varying in size from individual modules of less than 10^2 source lines to fully integrated systems of more than 10^8 lines. The human user—*the tester*—is able to apply comparable functions across a familiar interface as the software being tested evolves in size and complexity by several orders of magnitude. In fact, the only indicators of size or complexity that have ties to the *MOTHRA* architecture are the operating system cost penalties and performance delays inherent in manipulating massive objects. All other costs and resource demands are under the direct control of the tester. In most cases, the tester will choose to allow critical resources such as *time* or *memory* to grow linearly with program size and complexity. The tester may, however, choose to conserve these resources by sacrificing other resources (*e.g.*, dollars) or even by reducing the fidelity of the test. These are ulti-

mately economic decisions determined by the relative costs of tests and failures—*MOTHRA* does not *legislate* or even *favor* one kind of decision in preference to another.

An important mechanism for meeting these criteria is that *MOTHRA* is reconfigurable, allowing the integration of user and system tools with which the tester may already be familiar, and allowing the system to make use of different underlying hardware architectures of differing capabilities. We address this in *MOTHRA* by the use of *thematic tools* for software testing. It has been our experience that software testing is most effective when the test procedures can be reduced to a set of well-understood and natural activities. Since *MOTHRA* supports tests of both very small and very large programs, the details of the tools that are actually invoked vary in power and scope. However, even very different tools can implement basic *themes* that are carried along throughout the several phases of testing. For example, programmers in modern development environments interact increasingly with an array of very powerful source language debuggers. Even though formal testing methodologies and debugging are very different activities, the *debugging theme* can be used as a metaphor to carry the tester from tool to tool as the software being tested evolves.

One *MOTHRA* system has been constructed using the AT&T Bell Labs *Blit* interactive bitmap display terminal running under the control of a UNIX® window manager called *Layers*. The host environment is a modestly configured VAX 11/780 running UNIX 4.3 BSD. Another version has been implemented on VAXstations® running Ultrix 1.2 and the X Window System. However, the architecture of *MOTHRA* encourages re-hosting. Furthermore, explicit operations allow *MOTHRA* processes to spawn parallel and vectorized processes for execution by a Cyber 205 (or any other powerful parallel machine).

January 23, 1987

* The work presented in this paper was funded, in part, by RADC contract F30602-85-C-0255.

† The authors may be reached by e-mail addressed to:

Internet: rad@gatech.EDU spaf@gatech.EDU
uucp: ...!{akgua,decvax,hplabs,seismo}!gatech!{rad,spaf}

® UNIX is a registered trademark of AT&T Technologies.

The *MOTHRA* Software Testing Environment*

Richard A. DeMillo†
Eugene H. Spafford

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
+1 404 894-3180

1. Introduction

The *MOTHRA* Environment is an integrated set of tools and interfaces that support the planning, definition, preparation, execution, analysis and evaluation of tests of software systems. *MOTHRA* is designed to be used starting at the earliest stages of software development and continuing through the progressively later stages of system integration, acceptance testing, operation and maintenance.

The *MOTHRA* system satisfies three primary criteria. First, its interfaces—particularly user interfaces—are high-bandwidth. Second, the overall architecture imposes no *a priori* constraints on the size of the software systems that can be tested in the environment. While these seem to be unrelated criteria that address issues at differing levels of detail, they are, in fact, closely linked.

Since the ability to process very large integrated software is an explicit design goal, increasing the effective *feedback bit rate*[1] along key interfaces is an obvious way to design for acceptable functional performance. The bandwidth of the interface is simply the feedback bit rate that it supports. Bitmap displays and windowing are the usual means of increasing the bandwidth of user displays, for instance. Less obvious are techniques which increase the effective bit rate by graphical compression, statistical sampling, and analog representations. In *MOTHRA* information is highly compressed for presentation to the tester. This provides a high-bandwidth user interface in which structural and dynamic information is summarized graphically and exact representations of algorithm and program behavior are replaced by inexact *animations* of behavior, higher-order descriptions of process execution, and non-procedural specifications of program function.

The practical meaning of the second requirement is that the same architecture should be able to service programs varying in size from individual modules of less than 10^2 source lines to fully integrated systems of more than 10^8 lines. That is, the human user—*the tester* should be able to apply comparable functions across a familiar interface as the software being tested evolves in size and complexity by several orders of magnitude.

In fact, virtually the only indicators of size or complexity that have ties to the *MOTHRA* architecture are the operating system cost penalties and performance delays inherent in manipulating massive objects. All other costs and resource demands are under the direct control of the tester. In most cases, the tester will choose to allow critical resources such as *time* or *memory* to grow linearly with program size and complexity. The tester may, however, choose to conserve these resources by sacrificing other resources (*e.g.*, dollars) or even by reducing the fidelity of the test. These are ultimately economic decisions determined by the

[1] This use of the term *feedback bit rate* is apparently due to S. C. Johnson and refers to the natural measures of work and efficiency in software development environments. Roughly speaking, the feedback bit rate is the number of bits transferred across an interface (from host to user) per atomic user interface operation.

relative costs of tests and failures. *MOTHRA* does not *legislate* or even *favor* one kind of decision in preference to another.

The key to this approach is to design an environment in which most primitive operations are implemented as *local transformations* of data objects. Global operations, on the other hand, are never applied to these objects but rather are defined in terms of primitive transformations of more complex *atomic* objects.[2]

MOTHRA satisfies these requirements by first organizing the user interface around a high-resolution bit map display with adequate graphics and windowing capabilities and, second, by using the display as a tester's *view* into a larger (virtual) test context. A view is defined by a consistent set of object instances that comprise a meaningful state for the *MOTHRA* system. Such a state contains sufficient information for applying a set of primitive operations and generating test-related data and results in the form of new object instances. The tester need have only a dim idea about the representation or physical location of aspects of the test which are not in view. As a matter of fact, the total context of a sufficiently complex test may not be meaningful to a software tester at all; in this instance, a large team of testers will each have differing views of the test, the total context of which is really only understood by systems engineers.

One of our major concerns has been to make *MOTHRA* reconfigurable. For the most part, *MOTHRA* does not attempt to re-create capabilities provided by the environment in which it is hosted. The guiding principle has been to structure *MOTHRA* as a *subenvironment*[DeMi86] of an overall software development or support environment. This implies both a certain closure and a robust interface. The *MOTHRA* architecture supports as a function any meaningful composition of basic functions. This is accomplished through an object-oriented architecture and user interface. There are several motivations for not viewing *MOTHRA* as highly integrated into a more global host environment. Foremost among these are the need for isolation and protection of test-related processes.

This same goal is also addressed in *MOTHRA* by the use of *thematic tools* for software testing. It has been our experience that software testing is most effective when the test procedures can be reduced to a set of well-understood and natural activities. Since *MOTHRA* supports tests of both very small and very large programs, the details of the tools that are actually invoked vary in power and scope. However, even very different tools can implement basic *themes* that are carried along throughout the several phases of testing. For example, programmers in modern development environments interact increasingly with an array of very powerful source language debuggers. Even though formal testing methodologies and debugging are distinct activities, the *debugging theme* can be used as a metaphor to carry the tester from tool to tool as the software being tested evolves. For example, program mutation[DeMi78, Budd81, Howd82] requires testers to construct sets of tests to demonstrate that certain basic design and programming errors are not present.[3] A fundamental activity in program mutation is *revealing bugs* in the mutant programs. Powerful debuggers are therefore useful tools during the tests and can be carried along as thematic tools. Many other test methodologies can, in turn, be reduced to mutation testing.[Acre79, Budd81] Thus, these methodologies can also be supported by the thematic tools.

---

[2] We use the term *object* to mean a collection of data and operations on that data. An *atomic object* is one which allows only atomic operations, in the sense of *view atomicity*.[Allc83] We do not address concepts like reliability or fault tolerance with the design of *MOTHRA*. Further, the exact structure of these objects (*active* or *passive*, etc.) does not matter. The object paradigm is intended as simply a design approach to the construction of *MOTHRA*.

[3] In this sense, program mutation is a kind of fault detection experiment, as might be carried out to detect faults in digital circuits. Here, the experiments are applied to software and the fault model is the space of likely errors that programmers make. The "local transformations" mentioned previously are simply the fault insertion operations. This technique is general enough to simulate common coverage-based tests such as statement, branch, and path coverage as well as many other systematic software tests.

There are subsidiary issues that are addressed in the design of *MOTHRA*. Foremost among these is our belief in *capitalizing* the software development effort at an appropriate level. The notion of capital-intensive software engineering and production is not a new one. For the *MOTHRA* development group, this point of view has led us to a fairly cavalier attitude toward trading machine cycles for human effort in conducting a test. Provided only that it can be justified economically, *MOTHRA* will spawn machine-intensive tasks and organize them for execution by a computer resource of appropriate power. This function is called *resource-shifting* and, although it is under the control of the tester, *MOTHRA* organizes and partitions all test views to accommodate such remote processing.

2. User Views

Testers interact with *MOTHRA* through a *view* of the test. The tester's view presents images representing global test status as well as local objects, attributes and processes. There may be several views to which the tester has access at any one time, but these views must be accessed serially and the user cannot have two simultaneous and distinct views of tests.

Some of the objects in view are entirely local and *private* to the user. For example, the user may create a temporary file as an aid in deriving appropriate test cases. These objects are under the complete and total control of the current view, and the user who "owns" the view can create copy, share, and destroy these objects at will. At the other extreme are those objects that are *shared* by all views. These objects are typically under the control of agents or processes external to *MOTHRA*. An example of such a shared object is the source listing of the software being tested. Such objects might be the property of configuration management and library tools residing in a host environment. These tools enforce a specified set of rights to access or modify the shared objects. *MOTHRA* operations on any shared objects in view respect the rights inherited from the external owners or managers of these objects. Intermediate to these private and shared objects are the *public* objects. Objects that are public represent the visible activity of the test. These objects are generated by testers and by *MOTHRA* tools. Public objects may include test cases and results, traceability mappings between test events and specifications, and error/fault statistics. Some of these public objects are transient while others are persistent. Occasionally, a transient object (*e.g.*, test case number 6) affects a persistent object (e.g, the error count for path number 26) and is incorporated into the *MOTHRA* object base according to predefined dependencies, relationships, and operations in much the same fashion as source code files dependencies are treated by the UNIX **make** utility.[Feld79] The exact nature of these dependencies define a *policy* that is unique to the test and its organization. *MOTHRA* does not define these policies—it only enforces them.

In physical appearance, a view is bounded by the edges of a high-resolution bitmap display. Each window in the view gives the tester access to certain objects and operations that are currently meaningful. The tester *selects* windows, objects, and operations with a mouse that can be used to point to windows and their contents and to pull down menu selections that are displayed under user control.

MOTHRA interfaces have been implemented for the Bell Labs *Blit* interactive bitmap display terminal[4] running under the control of a UNIX window management executive called *Layers*, and on Digital Equipment's VAXstation II color and black-and-white display terminals running under the X Window System.[Sche86] These particular instances of the user interface are, however, not the only ones possible. The underlying architecture effectively disassociates the physical properties of the display from the tools which the display accesses. In essence, the display is treated as just another tool in the environment. Other display tools can be substituted provided that the environment's interface conventions are satisfied.

[4] The AT&T 5620 Dot-Mapped Display. See[Pike84], for example.

2.1. Functions and Operations

We will begin by briefly describing a typical set of functions that the tester invokes. These functions are generally invoked in a sequence of views, called a *run*. Runs may be suspended (saving the complete view at the time of suspension) and resumed at any time. However, atomic operations are non-interruptible. Therefore, the view that is actually associated with a suspended run may contain objects resulting from values returned at a later time by on-going atomic operations. These are managed by a data- and event-driven harness. The same mechanism is used to manage multiple views of a test. A single display, for instance, may be used to invoke a series of functions applied to two different source modules. Since only one view at a time can be available, the tester can invoke a set of atomic actions and suspend the run to begin a run for the second module.

2.1.1. Run Initiation

The key shared objects are the source files.[5] A run is initiated by identifying a set of source files and associating the name of the run with those files. *MOTHRA* handles the parsing of the source files to a convenient internal form and also manages the naming conventions for modules and other syntactic units contained in those files.

2.1.2. Test Level Selection

A test plan may specify any of several *levels* of testing to be performed.[Budd81] Examples of these levels are *statement analysis, predicate and domain analysis,*[Whit78] and *coincidental correctness* analysis. Statement analysis is used for determining that every statement in the program has been executed and has some effect on the functional behavior of the program. Predicate and domain analysis are used to determine that all branches and specified paths are properly selected and that domains associated with these predicates are properly defined. Coincidental correctness analysis is used to test for the presence of a wide variety of computational errors, including various arithmetic, data flow, and interface errors.[Good79]

Within each level, the user may also choose a *strength* of test, represented by a percentage. The exact meaning of a strength value depends on the specific level of testing and certain subsets of the levels that may be selected. For example, if the user selects the statement analysis level at 100%, the test can only be passed by constructing tests that fully exercise every statement in the program. Within the predicate and domain analysis level at 90% strength, the tester will be required to construct tests that with 90% certainty determine the boundaries of predicate domains.

The levels of test are defined in terms of certain *mutant* operators.[Budd78] That is, source code transformations that implement the desired level of testing. For example, in the statement analysis level, mutant operators called **san** and **sdl** are used to determine whether each statement has been executed and to what effect. The **san** operator replaces each source statement by a special statement called **trap** that raises an exception. Unless test cases are provided that raise all possible exceptions, all statements cannot have been exercised. On the other hand, the operator **sdl** replaces each statement by a **no-op**. Unless the transformed programs behave differently than the program being tested, the test data does not demonstrate that the given statements have any functional effect on program behavior.

Within the levels, classes of these mutant operators may be selected by the tester. In these cases, the tester will use the selected operators to implement specialized testing strategies.[Acre79] These selections may be made on the basis of known or suspected weaknesses, or perhaps upon economic considerations (*e.g.*,the tester may only have the resources available to test 25% of the mutants in a specified time span).

---

[5] *MOTHRA* is a multi-lingual environment. In the current version, *MOTHRA* is limited to processing Fortran 77 (the complete language) and Ada (a large subset). Later versions are planned for C, Modula 2, Lisp, and possibly others.

Selection of levels, mutant types, and strengths may also be associated with source code components. For example, during a unit test, the user may select only a certain subroutine for a particular level and strength of testing. During software integration testing, the tester may choose an incremental (*i.e.*, bottom-up) strategy in which a given level and strength are successively applied to units, then to integrating software that calls these subroutines, and so on.

2.1.3. Test Data Selection and Execution

An important test function is the construction of tests and the execution of the program on the test data. The creation of a set of test cases is essentially an *editing* function. The editing may be under the control of the human tester, who is trying to meet some specified level of testing (*e.g.*, testing for the presence of all coincidental correctness errors of a given type), an automated test data *generator*, a simulator, or even some data capture device that records digitalized inputs from sensors, operators and communications channels. Creation of appropriate tests is a key function. We will return to it again after some other supporting functions have been described.

The actual *testing* is carried out by executing programs on the test data. The results are observed by an *oracle* that decides whether or not the program has behaved properly. The notion of *proper behavior* can be quite complex. In unit and module testing, the concept is usually identified with functional correctness—that is, consistency with a written formal or informal specification. In later views of a more highly integrated software system or subsystem, correctness is less important than meeting functional or user requirements. The oracle mediates all of these authorities. If a formal specification is available, the oracle consults it. If a human user is the authority, the oracle takes advice from this source. If the behavior cannot be assessed without additional instrumentation, the oracle receives instrumented output and reacts accordingly.

If unacceptable behavior is observed, the policies in force for the test determine the next course of action. In some cases, the test proceeds after the nature and location of the error is recorded in a public record. In other cases, the cause of the failure is located and fixed immediately, resulting in a new view of the test.

2.1.4. Test Status Evaluation

During the testing process, the tester eventually wants to know whether or not testing has been completed. This determination may be subjectively made or it may be specified quite precisely and unambiguously. The latter case is obviously the more interesting one in *MOTHRA*.

Test status is instrumented and reported as dynamic progress toward meeting test goals specified during run initiation. The user may be interested in overall progress toward completing a test specified for a given level and strength. By the same token, the user may be interested in whether or not a test has been carried out to reveal a specific error or type of error. In all of these cases, test status can be defined in terms of a single primitive function: execution of a *mutant program* on the test data. If the test data—in the judgement of an oracle—does not distinguish the program being tested from the mutant program, then the mutant is said to be *live* and is reported as such. If, on the other hand, the oracle determines that the mutant behavior varies significantly from the behavior of the original program, then the mutant is marked *dead*.

Dynamic information on test progress can be displayed in graphical and tabular format and is archived in public and shared objects according to test policies enforced by *MOTHRA*.

2.1.5. Test Data Creation Revisited

Test status evaluation is used to guide the test creation process. The tester may elect to stop testing at this point or to strengthen the test data by attempting to kill some live mutants.

If all currently enabled mutants have been killed, the tester may wish to create new mutant types or begin testing a different subroutine.

In this process, the user is aided by the evaluation displays as well as by tools that may be imported. Suppose, for example, that the tester is attempting to kill all mutants that replace integer constants n with n+1 and n-1 (as might be required for domain analysis). In addition to reporting that these mutants remain alive, *MOTHRA* allows the user to examine the effects of these mutants in the context of the original program or even to browse through related source lines or live mutants. More powerful *test case* editing capabilities are available to create new tests, modify previous tests or to capture the results of other test data generators. If the user has an especially difficult time in constructing a test that kills these mutants, he may import a **debugger** to attempt to exhibit that the mutants are in fact "buggy" versions of the program.

2.2. The Display

The technology used in the display and the material presented in that display are critical to the design of *MOTHRA*. The *MOTHRA* window layout presents the user with a view of all the objects that were described above. Based on our classification of objects we have defined the following subwindows (displays) within the *MOTHRA* display:

- Mutant Status Manipulation: The icons that define and reference specified mutant types, aggregations of these types, and the levels and strengths of tests that can be defined from them.
- View Status: The graphic symbols or textual displays that represent the progress of the current view toward test objectives, or other measures of completion.
- Test Cases: Any object—whether constructed by the tester or captured from an external source such as a simulator—that is used to stimulate the software being tested.
- Source Language Representations: Each view of the test defines a fragment of the software being tested, and a source language representation of such a fragment is a high-level description of the fragment. By definition, the most primitive constructs in any source language representation are the source lines of code; all other representations associate text or graphical information with sets of source lines.
- Command Line: Terse communications, prompts and system status reports are directed to a degenerate (one line) window called the command line.

Testers may query and modify attributes displayed in any of these subwindows. Transient information and data are displayed by whatever means is most appropriate for the display tool. In our implementation, such transient data are displayed in windows that overlay (and may sometimes obscure) the fixed windows just described. An example of a transient object might be one of the thematic tools mentioned in Section 1. The tester must make any explicit interfaces and functional dependencies between transient objects and *MOTHRA* objects since none are implicit in our design of *MOTHRA*.

The *MOTHRA* Display handles "global" information in two distinct ways. First, it gives the tester access to objects not in the current view. For example, to initiate the testing session, the tester provided file names that were meaningful to the host's file system, even though *MOTHRA* does not contain file management capabilities. Second, simply touching and changing the attributes of objects in the Display can have affects on the other windows in the view—thus the Display encapsulates a set of "global" relationships for the rest of the view. For example, selecting a random sampling of substitution mutants results in a propagation of mutant status information to the other subwindows, such as the View Status subwindow.

Attributes of objects displayed in each window can be modified dynamically, so that, for instance, the display format of the source language text can be changed to bring the live mutants into view. More complex interactions between view and source windows are

possible. For example, the tester can point to a histogram "bar" in the view window and cause the corresponding live mutants to appear in the source window.

3. Subenvironment Architecture

Supporting the user display is a collection of tools bound together by an *information interface* and hosted on another environment. Specified access pathways and ports allow information, commands, and signals to flow between *MOTHRA* and the host environment. While most of these have operating system dependencies, they have been hidden in higher level constructs that appear to be primitives to *MOTHRA*. Although the overall design is robust, implementing these primitives is easier in some environments than in others.

For example, one of the reasons for conceiving *MOTHRA* as a subenvironment of a host is the need to control and manipulate faulty processes. Unlike most programming environments, the *intent* of *MOTHRA* is to execute faulty processes. While most software developers would like to consider failure to be an abnormal condition, the *MOTHRA* user deliberately seeks it out through the process of killing mutant programs. Many of the failures induced in this way are benign (the mutant program runs to completion but delivers incorrect results). Approximately one fourth of the mutants generated,[6] however, are not benign. They generate processes that run seriously amok and must be tightly controlled. The modes of failure in these processes run from simple errors such as division by zero to storage allocation and concurrency errors that could harm unrelated processes if allowed to proceed unconstrained.

An important aspect of these definitions is that the system defines a *process* at each time *n*, rather than just a state. This is a key idea for several reasons. First, the atomicity of actions may result in several intermediate states before any other *MOTHRA* function can be applied. Second, the display architecture and logical driver together constitute a data and event driven network of autonomous processes and unique definitions of sequences of states may not be possible in certain circumstances, whereas definitions of sequences of processes can be defined in terms of the external actions needed to invoke them. Third, error recovery and roll-back procedures as well as look-ahead optimization are easier to define and implement. Fourth, we anticipate the use of *MOTHRA* in conjunction with nondeterministic system testing procedures; recording and replaying test scenarios and associating internal test events with software inputs is relatively easy to implement if each major time step of the environment corresponds to a history of states.

The information interface is the *MOTHRA backplane*. In many respects, *MOTHRA* combines the features of both open and closed programming systems. *MOTHRA* is closed in that the fixed windows of a view and the objects, attributes and operations associated with them define an Entity-Relationship (E-R) model[Chen80] that cannot be modified. Thus the process monitors, test data generators, instrumentation and other tools associated with the fixed windows can always count on certain dependencies and relationships among essential objects in view—ensuring, for instance, reproducible behaviors.

On the other hand, *MOTHRA* is open to the extent that any E-R model-respecting tool whatsoever can be attached to the backplane. Editing is a simple example of a transient activity that can be imported in this way. Any file can be edited by any editor provided only:

- the file is editable by the editor in question;
- the point in time at which the editor is invoked does not preempt or interrupt an action defined to be atomic in the E-R model;
- no attributes or properties are introduced by the editor's actions or side effects that contradict attributes or properties of the E-R model.

In other words, any tool can be imported to the user's view, provided that the user is able to plug (or wire) that tool into the backplane. This is a particularly valuable design for a

[6] In our testing so far.

testing environment, since many testing tools share common tool fragments. It also permits some novel interactions between the host and *MOTHRA* environments. A software developer, for example, can attach a mutant generation and execution capability as a background activity during coding and debugging. This is a generalization of Weinberger's dynamic instruction counting tool.[Wein84] The underlying E-R model allows the processes of mutant generation and execution to be decoupled from the integrating framework provided by the display architecture (recall that the display technology is simply another tool that plugs into the backplane). One application of this capability is the inexpensive maintenance of test status throughout the development process by keeping killed mutant status information for object code.

4. Resource Shifting

The process of creating and executing mutant programs on the test cases $\tau_1, \tau_2, \ldots, \tau_k$ can be done serially in one of two logical orderings. The first ordering would be to apply the test cases, one at a time, to each live mutant and observe the results. The second ordering is where all test cases are applied to each live mutant and the results observed. All such serial processes consist of on the order of $\mu \times k$ independent transactions, where μ is the number of enabled mutants and k is the number of tests to be executed.[7] In either case, we are presented with a series of independent tasks.

Simply spawning these independent tasks to m independent parallel processors reduces the elapsed time for processing the test cases against the mutants to:

$$\frac{\mu \times k}{m} + OVERHEAD.$$

Since the *OVERHEAD* can be compressed to one of the serial protocols mentioned above, this amounts to a linear speed up on independent parallel processors. However, large blocks of these tasks have an internal structure that can be exploited to achieve more impressive speed gains.

For example, the **substitution** mutants of a simple assignment (using C-like notation) can be written in one of the following forms:

$$\textbf{*lhs} = \textbf{operand}_1 \times \textbf{operand}_2 \Rightarrow \textbf{*lhs}' = \textbf{operand}_1 \times \textbf{operand}_2 \quad (1)$$

$$\textbf{*lhs} = \textbf{operand}_1 \times \textbf{operand}_2 \Rightarrow \textbf{*lhs} = \textbf{operand}'_1 \times \textbf{operand}_2 \quad (2)$$

$$\textbf{*lhs} = \textbf{operand}_1 \times \textbf{operand}_2 \Rightarrow \textbf{*lhs} = \textbf{operand}_1 \times \textbf{operand}'_2 \quad (3)$$

Furthermore, the order in which these mutants appear is fixed once the program is known. At the time **mutgen** returns a value, the mutant statements (1)-(3) are equivalent to a *vector operation*

$$\textbf{LHS} = \textbf{OPERAND}_1 \otimes \textbf{OPERAND}_2,$$

where $\otimes$ is the vectorized binary operation and the vectors **LHS**[i], **OPERAND**$_1$[i], and **OPERAND**$_2$[i] are defined respectively to be ***lhs**, **operand**$_1$, and **operand**$_2$[i] if $i = 0$. For $i \geq 1$, the vector positions are defined by the mutant definitions (1)-(3). Thus, the **substitution** mutant executions are equivalent to a series of vector operations (followed by inner product operations to determine which mutants have been killed).

Interleaving the generation of vectorized expressions with parallel tasks can result in a multiplicative speed-up. This is especially attractive for the case of **substitution** mutants since for a typical n line program, the (worst-case) number of substitution mutants grows

[7] Some simplification is possible by "short-circuiting" an iteration once a mutant has been killed (there is no need to apply further test cases to a dead mutant), but we will ignore that and other optimizations in the following presentation so as to make it more accessible.

proportional to

$$\binom{n}{2}$$

which is the dominant term in the expression denoting the worst-case complexity of mutant generation and execution. For moderately sized software systems (*e.g.*, systems for which $10^4 \leq n \leq 10^6$) complete tests have required several days of dedicated computer time. With interleaved parallel tasking and vectorization on processors with MIPS rates in the 50-100 range, a thousand-fold speed-up is possible, bringing these tasks to within the reach of real-time responses.

This has led us to consider seriously the possibility of *shifting resources* to accommodate such processor intensive tasks. *MOTHRA* is designed to be hosted on hardware configured with multiple machines of varying capabilities.

For example, one host might consist of the bitmap displays, object definitions, and file services required for tester interaction. We assume also that whatever programming environment serves as the *host environment* for *MOTHRA* can be accessed through this host. In particular, editing and other transient functions do not make any demands on subsequent layers.

A second host consists of large-to-medium granularity parallel processors. Each of these processors operates on a common memory with appropriate programmer control of parallelism. The tester may—when local resource thresholds are exceeded— *shift gears*. The result is the spawning of blocks of independent parallel tasks for each of the processors. Coordination of destination processors and the collection and collation of the results of process execution is the responsibility of a process that resides on the first host. It is intended that the tester have complete control over the allocation of parallel resources. At present, however, this control is restricted to partitioning the serial tasks mentioned above in some appropriate manner.

In the same manner, vectorization is carried out as described above and the vectorized code and test cases are sent to a third host. Since the result of the vector operation is itself a vector, only this result is returned from this host. The precise format of vector operations is a machine-dependency that cannot be easily removed, although we anticipate that UNIX systems capable of 100-500 scalar MIPS with powerful vector extensions to C will become widely available. For the current version of *MOTHRA*, however, we are adopting a conservative approach. For example, long chains of data dependencies within loops are being partitioned to avoid vectorization difficulties.

The experimental performance studies of resource-shifting will be reported in detail elsewhere.

5. Conclusion

The *MOTHRA* environment described in this paper is currently implemented and running in a multi-host environment consisting of Digital Equipment VAX 11/780 and 11/750 minicomputers, VAXstation II workstations, AT&T Blit bitmap display terminals and a Control Data Cyber 205 supercomputer. Version 1.0 of *MOTHRA* contains at least primitive implementations for the functions described above, although many of the most desirable integrating features (*e.g.*, automating the transmission of vectorized processes from the VAX host to the Cyber 205) are not fully functional. Thus far, *MOTHRA* has been used to test Fortran 77 programs in the 20-500 line range. With current memory and other constraints (there are no *MOTHRA* design constraints) complete testing of 1,000-10,000 line Fortran programs seems well within the capabilities of Version 1.0.

A second version that exploits optimization opportunities and will be tailored to extremely large-scale applications is under design.

Although user experience with *MOTHRA* is currently confined to our development group, we expect Version 1.0 to be available on a limited scale to a community of 30-50 software

testers. In spite of the care we have taken to ensure that fundamental design concepts really match the needs of realistic software testing, we anticipate that many hitherto unidentified issues will surface. These experiences will be analyzed and reported at a later date. We are optimistic, however, that a software testing environment architected as described above will deliver acceptable levels of computing resources to the important problem of how to test and evaluate the quality and reliability of large software systems. Furthermore, we anticipate that the system will be easily learned and easily used, thus leading to improvements in testing and software production.

References

Acre79.

Acree, A. T., R. A. DeMillo, T. A. Budd, R. J. Lipton, and F. G. Sayward, "Mutation Analysis," TECHNICAL REPORT GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1979.

Allc83.

Allchin, J. E., "An Architecture for Reliable Decentralized Systems," PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1983. Also released as technical report GIT-ICS-83/23

Budd78.

Budd, T. A., R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The Design of a Prototype Mutation System For Program Testing," *PROCEEDINGS NCC, AFIPS CONFERENCE RECORD*, pp. 623-627, 1978.

Budd81.

Budd, T. A., "Mutation Analysis: Ideas, Examples, Problems, and Prospects," in *Computer Program Testing*, ed. B. Chandrasekaran and S. Radicchi, pp. 129-148, North-Holland, 1981.

Chen80.

Chen, P. P., *Entity-Relationship Approach to Systems Analysis and Design*, North-Holland, 1980.

DeMi78.

DeMillo, R. A., R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *COMPUTER*, vol. 11, no. 4, pp. 34-43, IEEE, April 1978.

DeMi86.

DeMillo, R. A., "Functional Capabilities of a Test and Evaluation Subenvironment in an Advanced Software Engineering Environment," TECHNICAL REPORT GIT-SERC-86/07, Software Engineering Research Center, Georgia Institute of Technology, Atlanta, GA, 1986.

Feld79.

Feldman, S. I., "Make—A Program for Maintaining Computer Programs," *SOFTWARE PRACTICE AND EXPERIENCE*, vol. 9, pp. 255-265, 1979.

Good79.

Goodenough, J. B., and S. L. Gerhart, "Towards A Theory of Test Data Selection," *TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-1, no. 2, pp. 156-173, IEEE, June 1979.

Howd82.

Howden, W. E., "Weak Mutation Testing," *TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-8, no. 4, pp. 371-379, IEEE, July 1982.

Pike84.
Pike, R., "The Blit: A Multiplexed Graphics Terminal," *BELL LABORATORIES TECHNICAL JOURNAL*, vol. 63, no. 8, pp. 1607-1630, AT&T, October 1984.

Sche86.
Scheifler, R. W., and J. Gettys, "The X Window System," *TRANSACTIONS ON GRAPHICS*, no. 63, ACM, 1986.

Wein84.
Weinberger, P. J., "Cheap Dynamic Instruction Counting," *BELL LABORATORIES TECHNICAL JOURNAL*, vol. 63, no. 8, pp. 1815-1826, AT&T, October 1984.

Whit78.
White, L. J., E. I. Cohen, and B. Chandrasekaran, "A Domain Strategy for Computer Program Testing," TECHINCAL REPORT OSU-CISRC-TR-78-4, Ohio State University, 1978.

THE VIEWGRAPH MATERIALS

FOR THE

E. SPAFFORD PRESENTATION FOLLOW

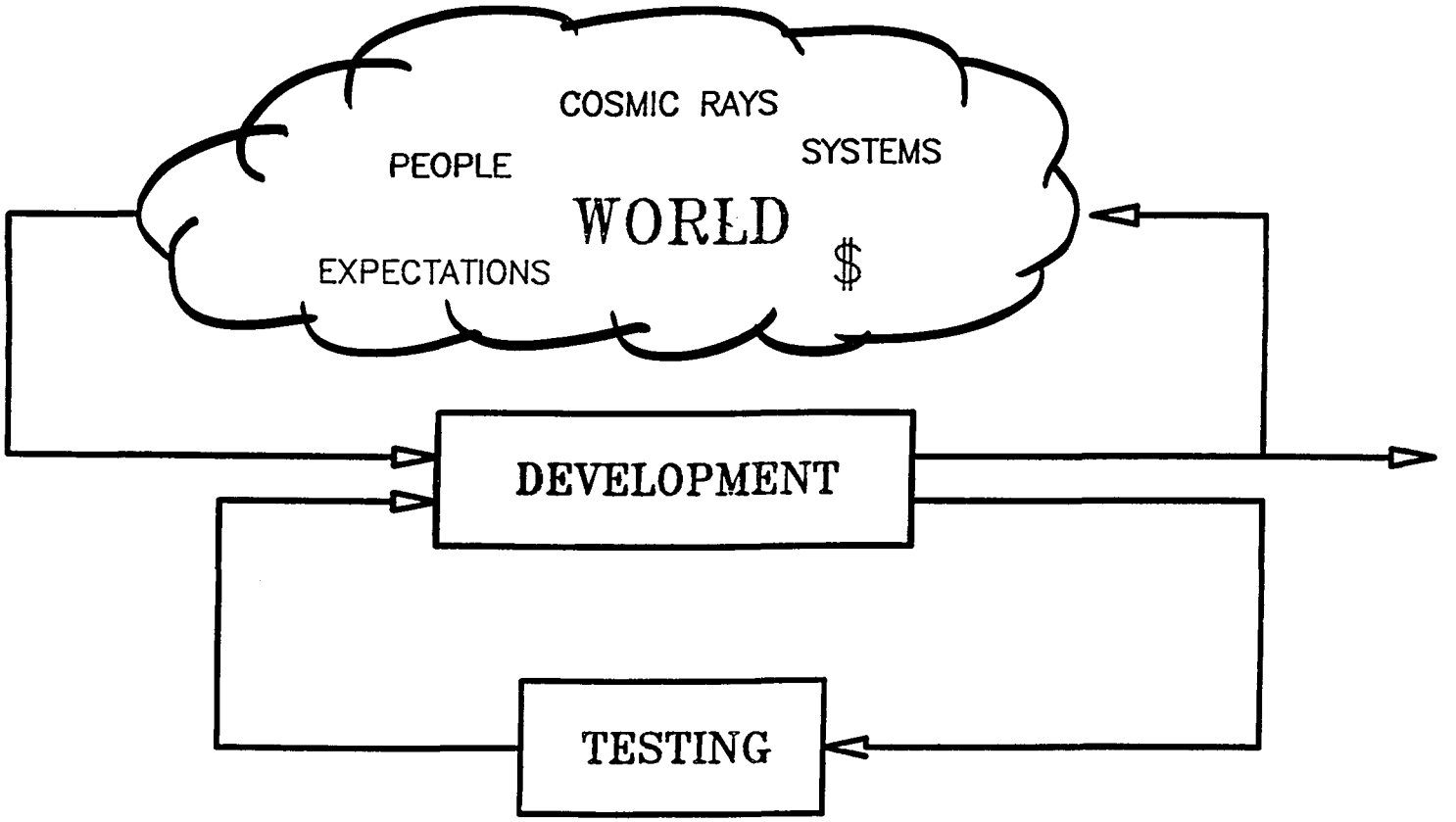
THE **MOTHRA** SOFTWARE TESTING ENVIRONMENT

Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332

TESTING IS A RISK–REDUCING ACTIVITY

* INCREASE CONFIDENCE THAT SYSTEM BEHAVES AS INTENDED
* DEMONSTRATE MODES OF FAILURE
* FIND AND REMOVE INTRINSIC CAUSES OF FAILURE

STRICTLY SPEAKING,
TESTING IS NOT PART OF THE DEVELOPMENT PROCESS



TESTING IS A CONTROL ON THE DEVELOPMENT PROCESS

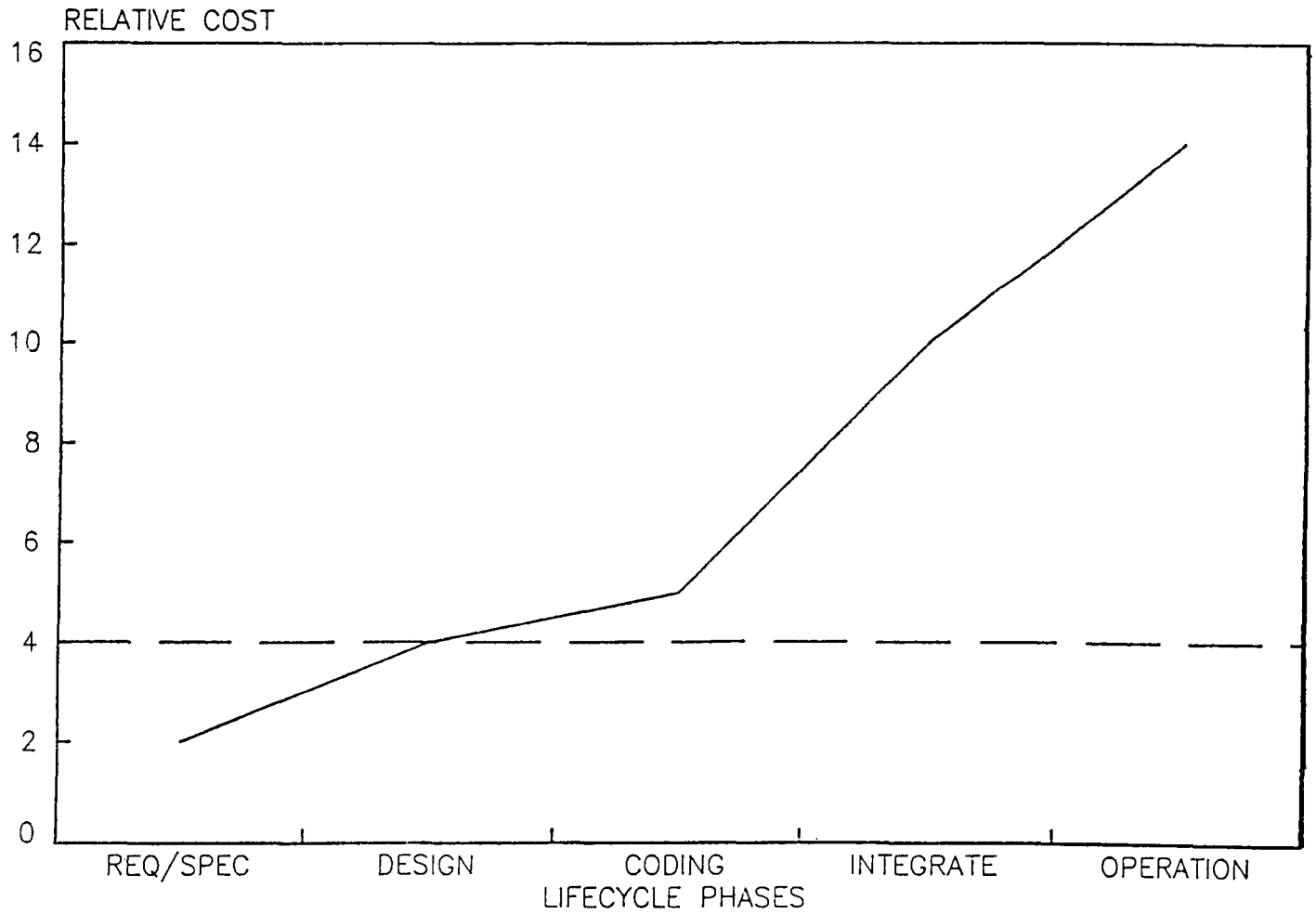
COSTS

COSTS DUE TO TESTING

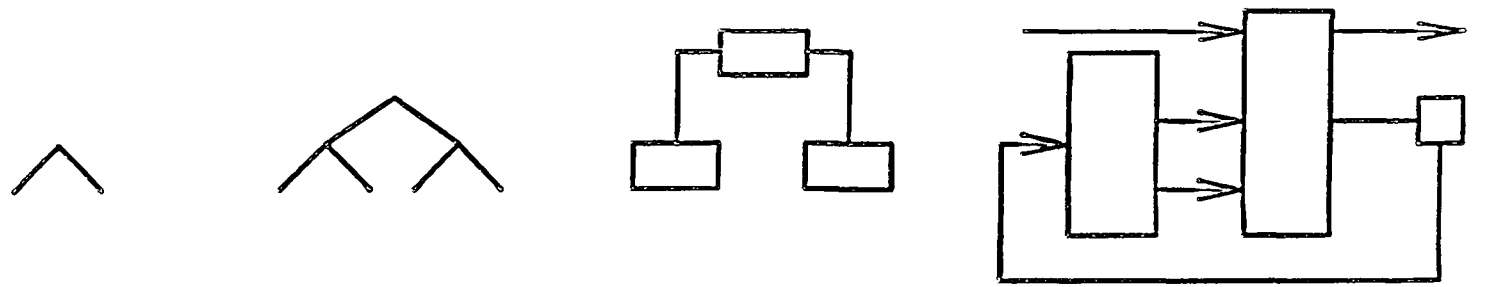
COST OF ERRORS

RELATIVE COST OF ERROR CORRECTION

REL. COST — DESIGN COST - -



THE MOTHRA SOFTWARE TESTING ENVIRONMENT



UNIT *SYSTEM*

100 LOC → *1M LOC*

COMMON INTERFACES, TOOLS, METHODS, THEMES

HIGH BANDWIDTH INTERFACES

GRAPHICS, WINDOWING, ANIMATION, DATA COMPRESSION, OBJECT-ORIENTED

NO A PRIORI SIZE CONSTRAINTS

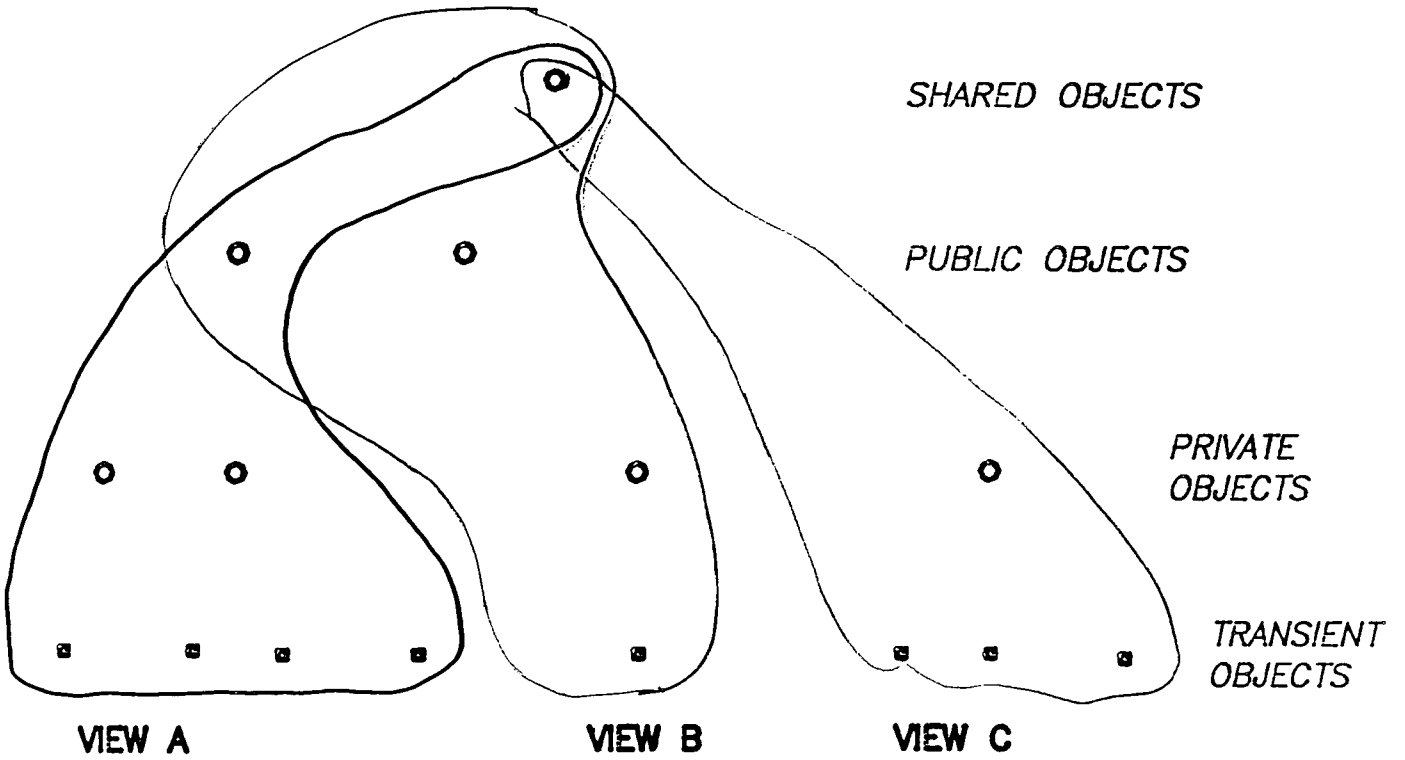
ATOMIC OPERATIONS ARE LOCAL TRANSFORMATIONS OF EXECUTABLE OBJECTS

THREE ASPECTS OF MOTHRA

* *VIEWS -- A WAY OF MANAGING LARGE TESTS*
* *THEMATIC TOOLS -- A METHODOLOGY FOR USER INTERACTIONS*
* *SHIFTING GEARS -- CAPITALIZING THE TEST AT AN APPROPRIATE LEVEL*

VIEWS OF LARGE TESTS

BOUNDARIES OF BITMAP DIPLAY DEFINE A WINDOW INTO A MORE GLOBAL TEST CONTEXT



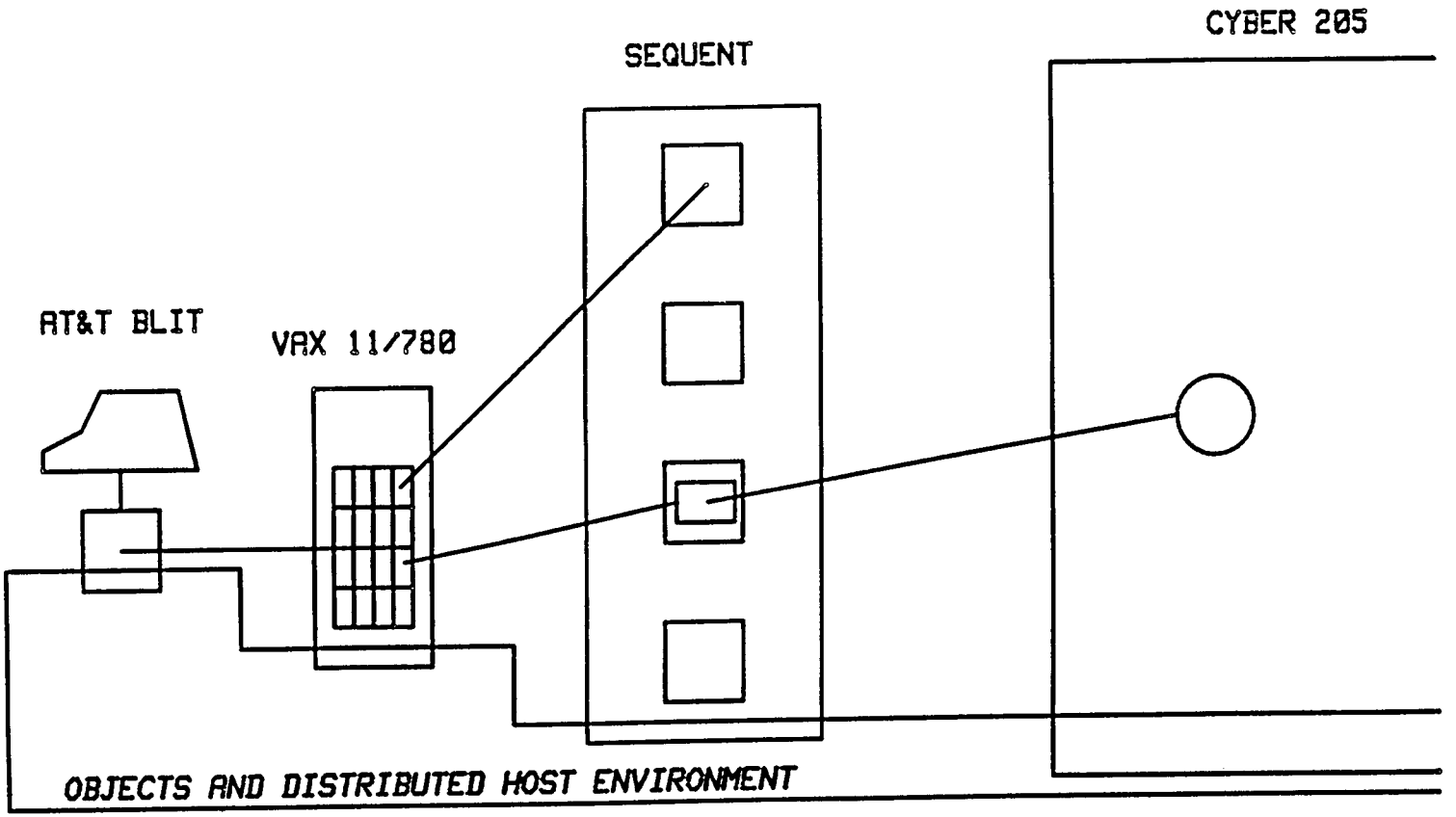
VIEW DEFINE CONSISTENT, MEANINGFUL STATES FOR MOTHRA

COMMON TOOL THEMES

* *INSTRUMENTATION*
  * *PI–LIKE SYMBOLIC DEBUGGERS*
  * *ZERO–IMPEDANCE PROBES*

* *EDITING*
  * *TEST ADVISOR*
  * *THE ORACLE*

SHIFTING GEARS IS EASY
ONCE THE ENVIRONMENT IS CAPITALIZED



1 MIP ——————————————————————→ 200 MIPS

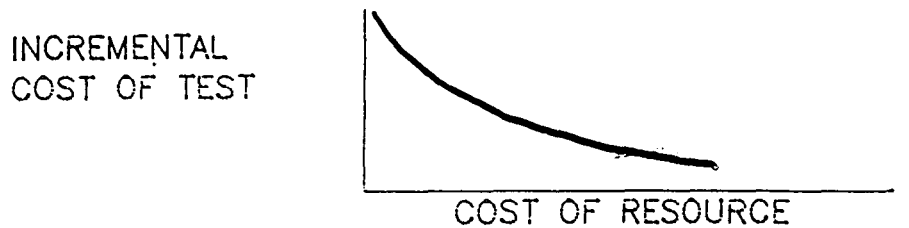
INCREASED COST
INCREASED FIDELITY
INCREASED SIZE

INCREMENTAL COST OF TEST

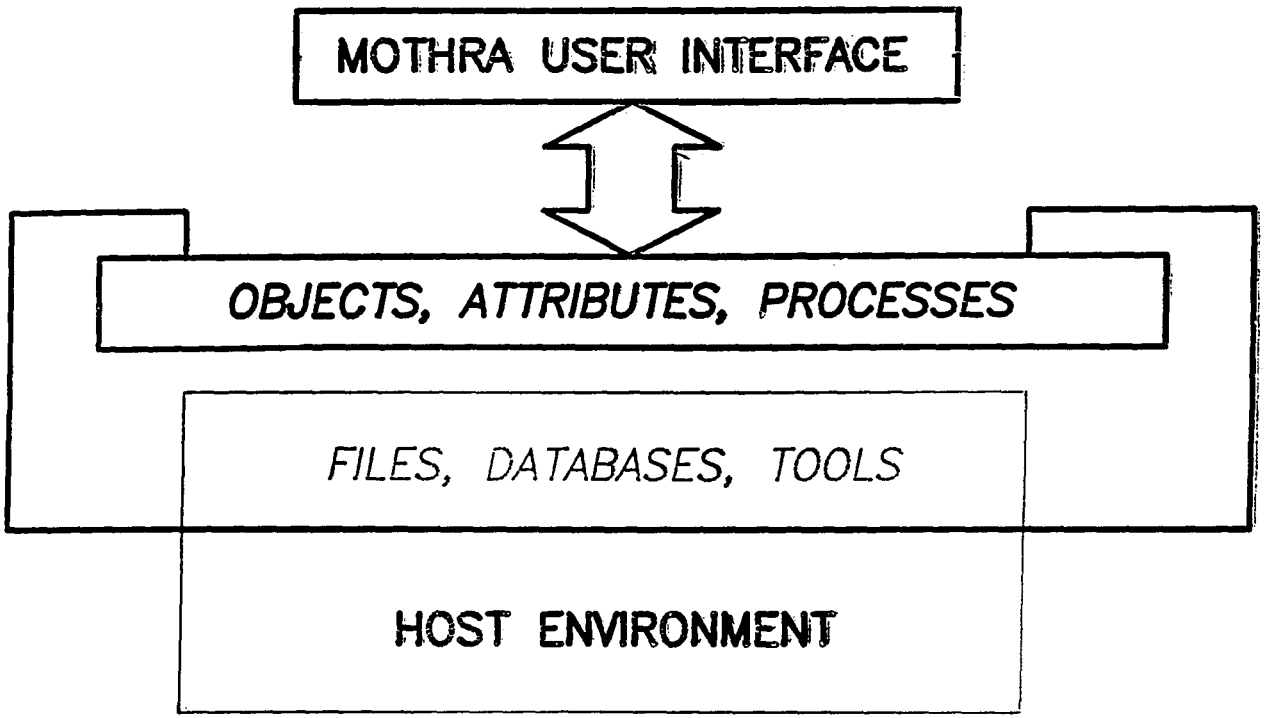
$$C(R,P,L,S) = \frac{\text{COST OF TESTING P AT LEVEL L AND STRENGTH S ON R}}{\text{COST OF RUNNING P ON R}}$$

FIRST SET OF EXPERIMENTAL RESULTS
(NOT TO BE GENERALIZED)

C({VAX 780, CYBER 205}, P, {sal, pdl, ccl}, 1.00) = 1.02
C({VAX 780}, P, {sal, pdl, ccl}, 1.00) > 500



MOTHRA IS DESIGNED TO BE A SUBENVIRONMENT



* *4.2 BSD ON VAX 11/780 (Layers on Blit)*
* *ULTRIX ON VAXStation II (X-Windows)*

TECHNIQUE

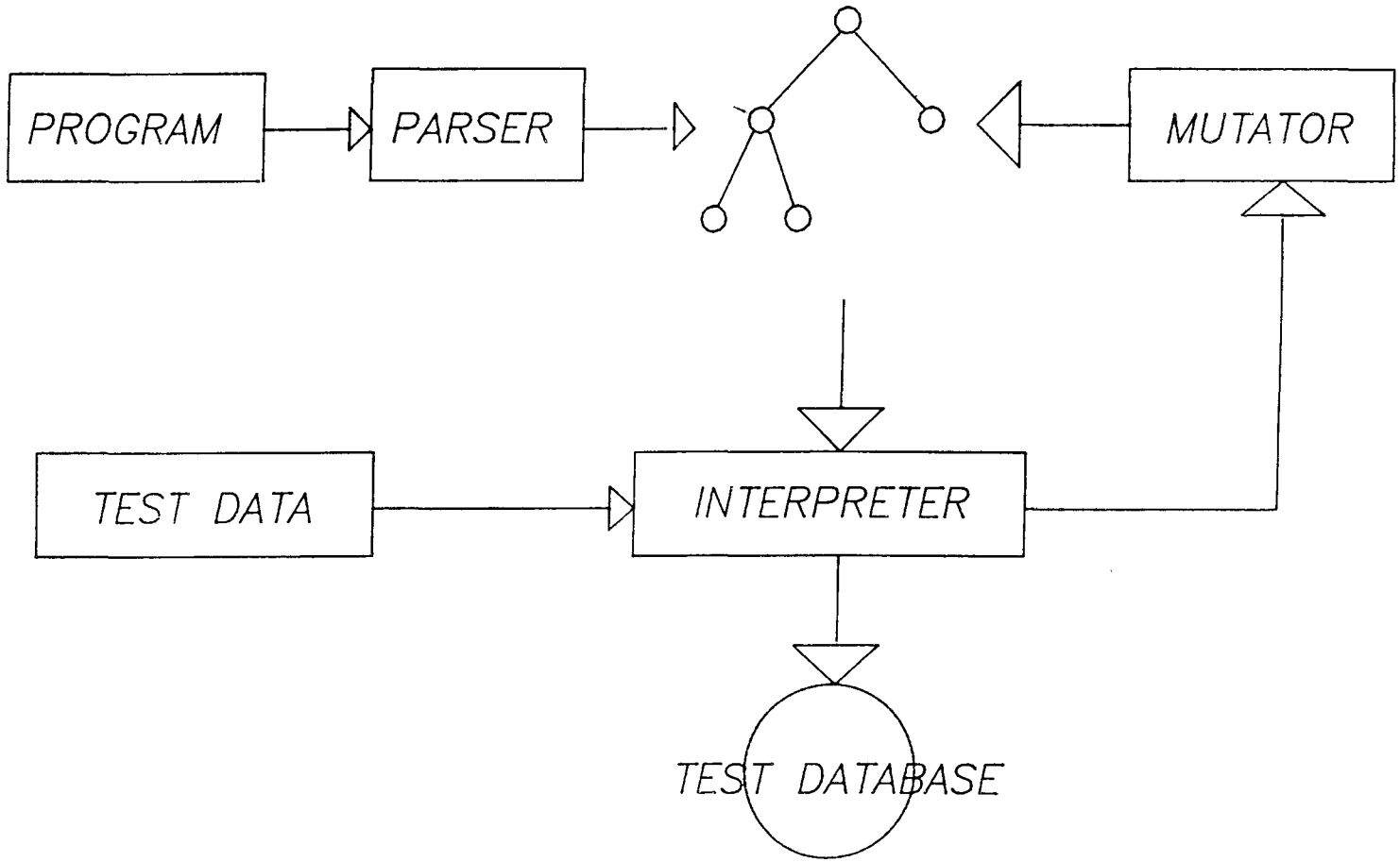
Given: program P
test data T

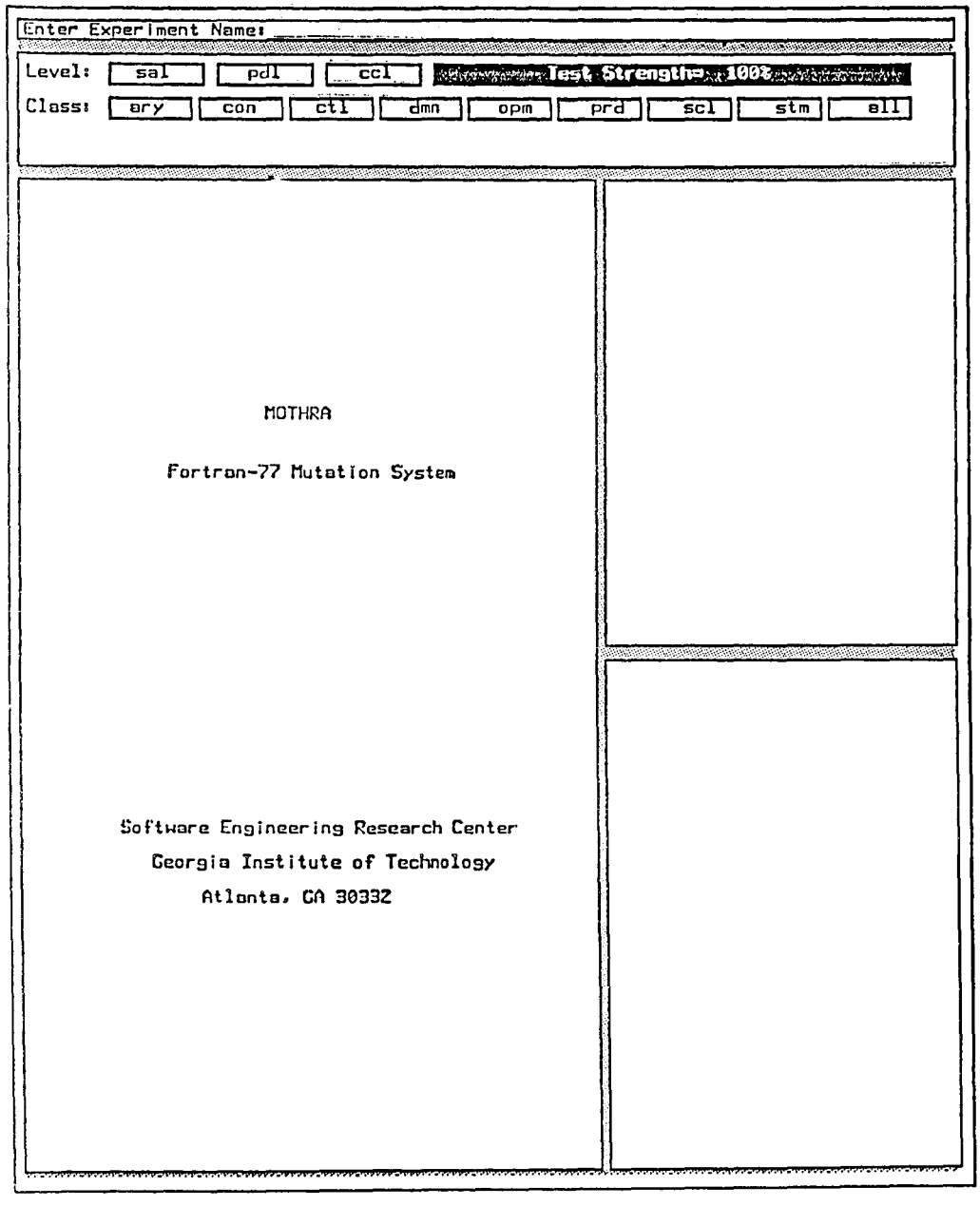
Construct: a set of MUTANT programs M(P)

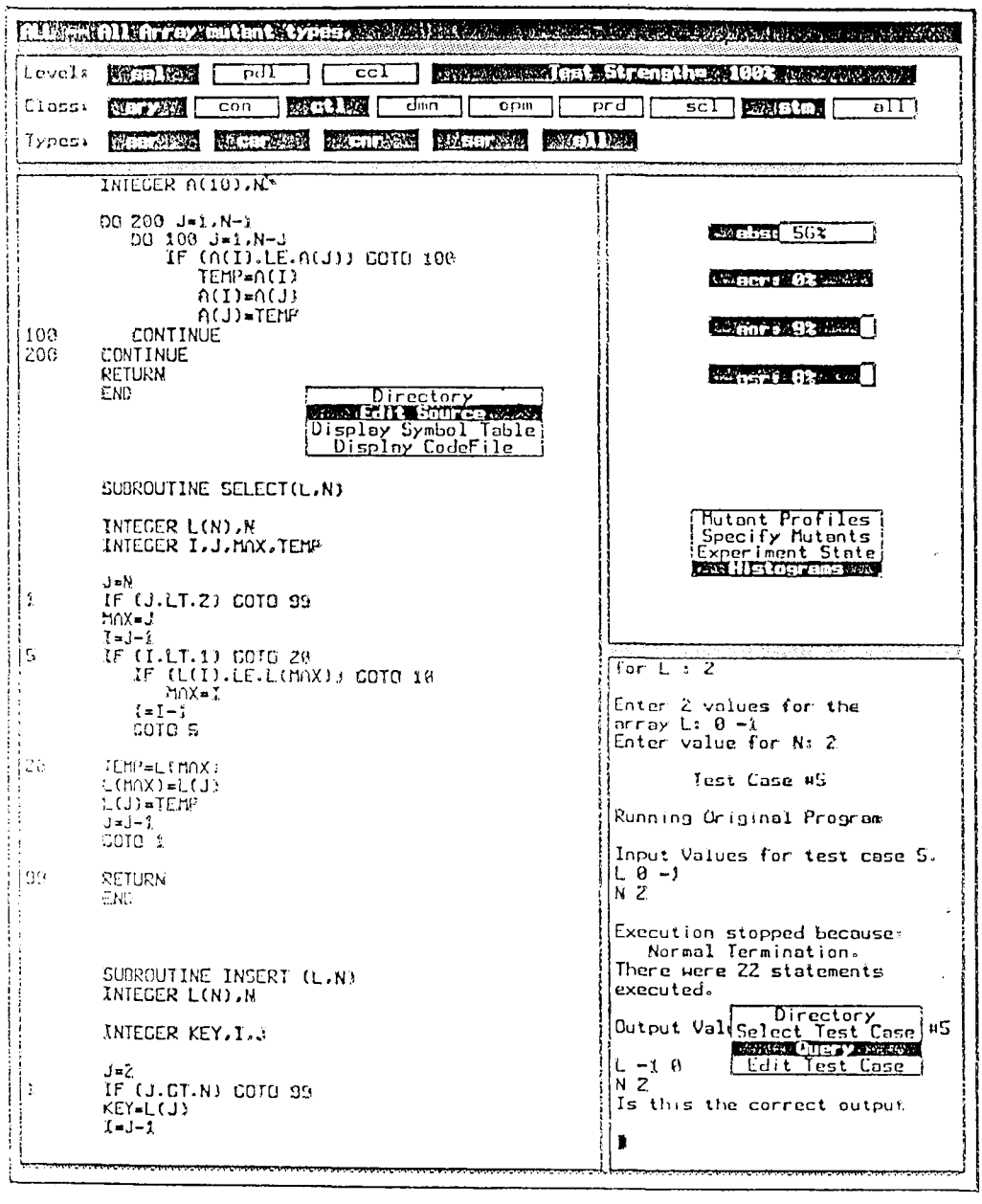
M(P,T) = % of programs in M(P)

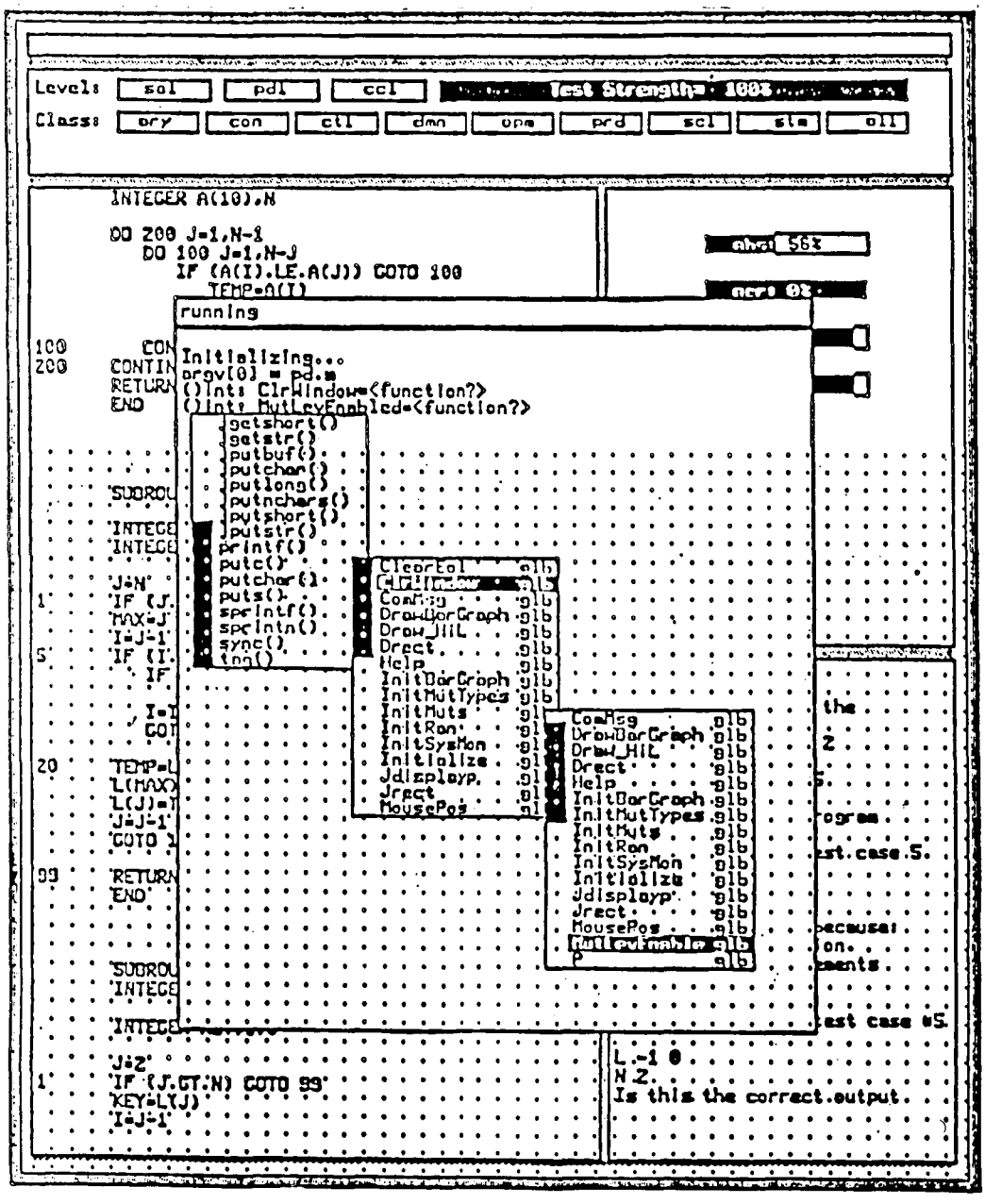
(1) not equivalent to P
(2) give different results on T

AUTOMATIC MUTATION SYSTEM




Enter Experiment Name:
Level:
sal
pdl
ccl
Test Strength= 100%
Class:
ary
con
ctl
dmn
opm
prd
scl
stm
all
MOTHRA
Fortran-77 Mutation System
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, GA 30332


Level: pdl ccl Test Strength: 100%
Class: con dmn opm prd scl all
Types:
INTEGER A(10),N
DO 200 J=1,N-1
DO 100 J=1,N-J
IF (A(I).LE.A(J)) GOTO 100
TEMP=A(I)
A(I)=A(J)
A(J)=TEMP
100 CONTINUE
200 CONTINUE
RETURN
END
Directory
Edit Source
Display Symbol Table
Display CodeFile
SUBROUTINE SELECT(L,N)
INTEGER L(N),N
INTEGER I,J,MAX,TEMP
J=N
1 IF (J.LT.2) GOTO 99
MAX=J
I=J-1
5 IF (I.LT.1) GOTO 20
IF (L(I).LE.L(MAX)) GOTO 10
MAX=I
10 I=I-1
GOTO 5
20 TEMP=L(MAX)
L(MAX)=L(J)
L(J)=TEMP
J=J-1
GOTO 1
99 RETURN
END
SUBROUTINE INSERT (L,N)
INTEGER L(N),N
INTEGER KEY,I,J
J=2
1 IF (J.GT.N) GOTO 99
KEY=L(J)
I=J-1
abs: 56%
acr: 0%
mnr: 9%
Mutant Profiles
Specify Mutants
Experiment State
Histograms
for L : 2
Enter 2 values for the
array L: 0 -1
Enter value for N: 2
Test Case #5
Running Original Program
Input Values for test case 5.
L 0 -1
N 2
Execution stopped because:
Normal Termination.
There were 22 statements
executed.
Output Val
Directory
Select Test Case
Query
Edit Test Case
#5
L -1 0
N 2
Is this the correct output


Levels: sol pdl ccl Test Strength: 100%
Class: ary con ctl dmn ope prd scl sim all
INTEGER A(10),N
DO 200 J=1,N-1
DO 100 J=1,N-J
IF (A(I).LE.A(J)) GOTO 100
TEMP=A(I)
running
Initializing...
argv[0] = pd.m
()int: ClrWindow=<function?>
()int: MutLevEnabled=<function?>
getshort()
getstr()
putbuf()
putchar()
putlong()
putnchars()
putshort()
putstr()
printf()
putc()
putchar()
puts()
sprintf()
sprintn()
sync()
ClearEol glb
ClrWindow glb
ComMsg glb
DrawBarGraph glb
Draw_Hit glb
Drect glb
Help glb
InitBarGraph glb
InitMutTypes glb
InitMuts glb
InitRan glb
InitSysMon glb
Initialize glb
Jdisplayp glb
Jrect glb
MousePos glb
MutLevEnable glb
P glb
L-1 0
N 2
Is this the correct output

TECHNICAL APPROACH

* *BUILD PROTOTYPE BASED ON PROGRAM MUTATION APPROACH TO TESTING*
  - ***SYSTEMATIC AND QUANTITATIVE***
  - ***WELL–DEVELOPED THEORETICAL BASIS***
  - ***EXTENSIVE EXPERIMENTAL VALIDATIONS***
  - ***SCALES UP***
  - ***IDEAL FOR SUPERCOMPUTER IMPLEMENTATION***

* *ADAPT PROTOTYPE TO ADA*

* *CONDUCT FEASIBILITY DEMONSTRATIONS*

PROJECT STATUS

* *TWO IMPLEMENTATIONS OF VERSION 1 INSTALLED AND BEING TESTED*
  - PROCESSES COMPLETE FORTRAN 77 LANGUAGE
  - LAYERS (VAX 11/780) AND X-WINDOWS (VAXStations)
* *ADA CAPABILITY DESIGNED*
  - ERROR OPERATORS DEFINED
  - ARCHITECTURE SPECIFIED
* *PERFORMANCE STUDIES INITIATED*
  - VERSION 1 OPTIMIZATION
  - SUPER COMPUTER IMPLEMENTATIONS