TAOS:
Testing with Analysis and Oracle Support

Debra J. Richardson
Information and Computer Science
University of California
Irvine, California 92717-3425
714-856-7353 djr@ics.uci.edu

Abstract

Few would question that software testing is a necessary activity for assuring software quality, yet the typical testing process is a human intensive activity and as such, it is unproductive, error-prone, and often inadequately done. Moreover, testing is seldom given a prominent place in software development or maintenance processes, nor is it an integral part of them. Major productivity and quality enhancements can be achieved by automating the testing process through tool development and use and effectively incorporating it with development and maintenance processes.

The TAOS toolkit, Testing with Analysis and Oracle Support, provides support for the testing process. It includes tools that automate many tasks in the testing process, including management and persistence of test artifacts and the relationships between those artifacts, test development, test execution, and test measurement. A unique aspect of TAOS is its support for test oracles and their use to verify behavioral correctness of test executions. TAOS also supports structural/dependence coverage, by measuring the adequacy of test criteria coverage, and regression testing, by identifying tests associated or dependent upon modified software artifacts. This is accomplished by integrating the ProDAG toolset, Program Dependence Analysis Graph, with TAOS, which supports the use of program dependence analysis in testing, debugging, and maintenance.

This paper describes the TAOS toolkit and its capabilities as well as testing, debugging and maintenance processes based on program dependence analysis. We also describe our experience with the toolkit and discuss our future plans.

1 Introduction

Most would agree that software testing is a necessary activity for evaluating high assurance properties of software-intensive systems, such as dependability, safety, performance, and correctness. Testing is seldom, however, given a prominent place in development and maintenance processes, nor is it an integral part of them. In particular, it is typically left as the last phase and skimped on when time or resources run slim. At the same time, there has been less emphasis placed on transitioning advanced testing technology than on introducing improved development technology in industrial organizations. Yet, it is well accepted that current software development technology is incapable of producing software satisfying the high assurance requirements of today's critical, software-intensive systems. In light of this and the fact that software development costs are exacerbated when low quality or unreliable software is released, testing processes used for critical software systems must be improved.

The typical testing process is a human intensive activity and as such, it is usually unproductive, error prone, and often inadequately done. Major productivity enhancements can be achieved by automating techniques through tool development and use. Errors made in testing activities can be reduced through formalizing the methods used. Defining testing processes secures more accurate, more complete and more consistent testing than do human-intensive, ad hoc testing processes. Moreover, accurately defining the testing process renders it repeatable, measurable, and hence improvable.

The TAOS, Testing with Analysis and Oracle Support, toolkit provides automated support for the testing process. This support includes test artifact management and development, test execution, and test process measurement.

One of TAOS' primary tasks is persistent object manage-

ment for the various and diverse types of artifacts that are created during the testing process. These test artifacts include test cases, which include test data and associated information such as actual output and execution status, test suites, which are collections of related test cases, test criteria, which are requirements to be met by the testing process, and test oracles, which are mechanisms for determining behavioral correctness during test execution. TAOS provides substantial support for developing these test artifacts. In addition, TAOS maintains the relationships between test artifacts and the system or system components to which they correspond and to analysis artifacts upon which they may depend as well as the many relationships between the test artifacts themselves.

One innovative aspect of TAOS is its provision for test oracles, which provide a mechanism for specifying the correct behavior of a system or system component for some or all test executions. Testing technology has long ignored the issue of test oracles and behavior verification, which is a critical step in the testing process. If behavior verification is left to humans, it is too often done carelessly. TAOS supports a variety of test oracles, from manual checking to specification-based oracles. Behavior is verified against test oracles, which are associated with individual test cases or entire test suites, during test execution.

TAOS provides a toolkit that automates many tasks in the testing process. The capabilities provided by the toolkit include random test data generation, source code instrumentation to support capturing execution traces, parallel test execution, and behavior verification against specified test oracles. The majority of these capabilities are independent of both language and component size; thus, TAOS provides support for unit to system testing of software components written in any language.

TAOS also supports structural/dependence testing by measuring coverage of test criteria that are based on the implementation. This is accomplished by integrating TAOS with ProDAG, Program Dependence Analysis Graph, which constructs control flow and dependence graphs.[1] TAOS and ProDAG also support regression testing by identifying test cases that are associated with or dependent upon modified software artifacts. The combination of ProDAG and TAOS thus supports the use of program dependence analysis in testing, debugging, and maintenance.

TAOS and ProDAG have been transitioned for use on industrial projects. The customer has been extremely pleased with the level of support provided, and feedback from project personnel has been instrumental in identifying toolkit enhancements and specific adaptions beneficial to their project. In this effort, we have found that basing the testing process on dependence coverage facilitates process measurement. In addition, TAOS provides effective test process management and automation, which enables process measurement and improvement. The combination of TAOS and ProDAG provided the customer with a more comprehensive toolkit than other available testing tools (including commercial options).

This paper describes the TAOS toolkit and testing, debugging and maintenance processes based on program dependence analysis that are supported. We begin by describing the capabilities of TAOS, which is the primary topic of this paper, with special emphasis on support for dependence coverage testing and the capabilities provided for test oracles and behavioral verification. Next, we outline the process support provided for testing, debugging and maintenance by TAOS and ProDAG. Finally, we describe our experience with using the toolkit, our future plans, and the contributions of this work.

2 TAOS Capabilities

On the TEAM project [CRZ88], we developed a toolkit infrastructure for prototyping, experimenting with, and integrating diverse testing, analysis and evaluation tools and processes. This infrastructure consisted of a layered architecture including an environment support layer that provided object management and language processing capabilities, a basic analysis components layer that provided generic analysis capabilities, an advanced tool layer with testing and analysis tools built upon the lower layers, and a process layer that supported integration of tools. During the TEAM effort, we identified several essential analysis components required to support development of advanced analysis and testing tools. 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. TEAM focused on identifying the most generic capabilities and abstract interfaces for these analysis components so that they could be instantiated to meet varying needs. Another emphasis was developing language independent components to support analysis and testing throughout the lifecycle of large-scale software systems, particularly for pre-implementation descriptions and systems implemented in multiple languages.

We have continued this design philosophy in the development of TAOS. We have developed capabilities to manage test and analysis artifacts above a general-purpose object management system. Based on our subsequent testing research, we recognized the need for more advanced static analysis techniques (than data flow analysis) and developed

[1] ProDAG currently analyzes Ada programs; thus TAOS' support for structural testing is only provided for Ada.

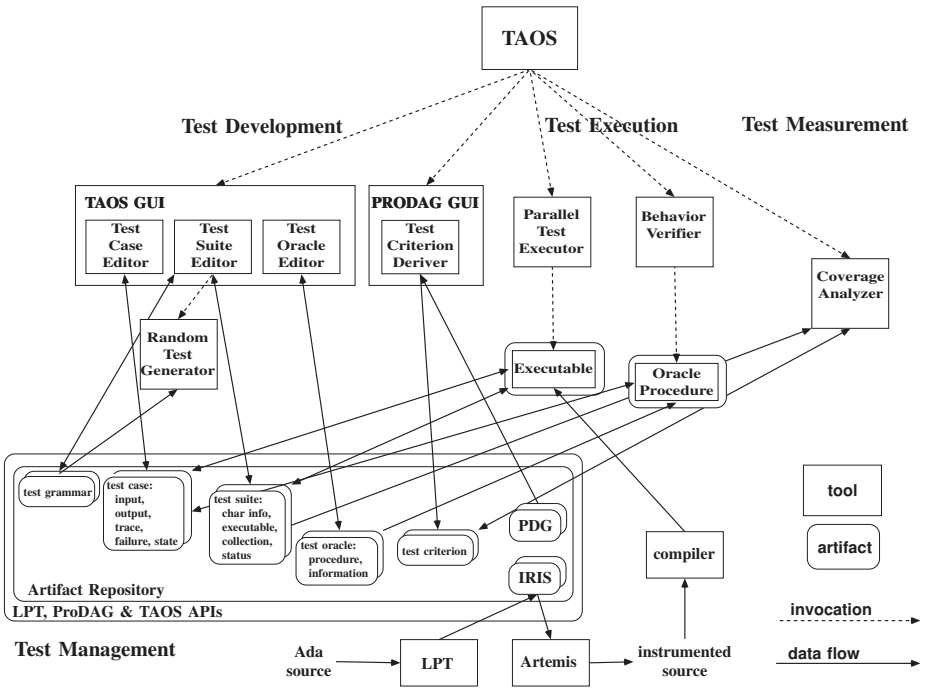

Figure 1: TAOS Toolkit

a program dependence analysis toolset. We also identified a number of other tools that can be integrated to provide sophisticated testing and analysis processes. We have continued the objective of language independence in the development of these analysis components and testing tools. Most capabilities are independent of the language in which the component under test is written. Others were developed so as to facilitate future support for multiple languages.

Here, we describe the TAOS toolkit and provided capabilities. Figure 1 shows a high-level view of the TAOS toolkit and the interaction of the various tools depicted along with the major test artifacts manipulated.[2] We also describe ProDAG, which is a syntactic analysis toolset integrated with TAOS and upon which some of the TAOS tools rely. These tools, artifacts, and the general activities of test management, test development, test execution, and test measurement are described in this section.

Throughout this paper, a running example appears that corresponds to a program implementing an alarm clock. The informal requirements of this alarm clock follow.

- Initially, the bell is off and the alarm is disabled.
- Whenever the current time is the same as the alarm time and the alarm is enabled, the bell shall start ringing. This is the only condition under which the bell starts to ring.
- If the alarm is disabled while the bell is ringing, the bell shall stop ringing.
- The time and the alarm time can be set at any time.
- Resetting the clock and enabling or disabling the alarm are considered to be done instantaneously.

[2] For clarity, not all tools and capabilities are shown.

Throughout the paper, the various artifacts developed for the alarm clock are shown. These include:

- a formal specification in Graphical Interval Logic [DKMS+92], which is used to develop a test oracle for verifying the behavior of the implementation during testing;
- an Ada implementation;
- a test grammar, which describes inputs to the implementation and is used for randomly generating test data;
- various analysis and test artifacts developed by the TAOS toolkit.

2.1 Test Management

Testing and validation is an extremely complicated process consisting of many activities and dealing with many artifacts created both during testing as well as during other development and validation activities. TAOS provides an application programmatic interface (API) that supports creating, manipulating, and accessing test artifacts plus the relations among these artifacts, which are maintained in a persistent artifact repository. These relationships are also maintained persistent in the repository. The repository is implemented atop PLEIADES, which provides flexible object management capabilities [TC93].

Through it's interaction with ProDAG, TAOS supports relationships between test artifacts and analysis artifacts. Through interaction with other software tools, TAOS supports interaction with development activities; test artifacts can be related to synthesis and/or analysis artifacts from which they are derived or for which they are to test.

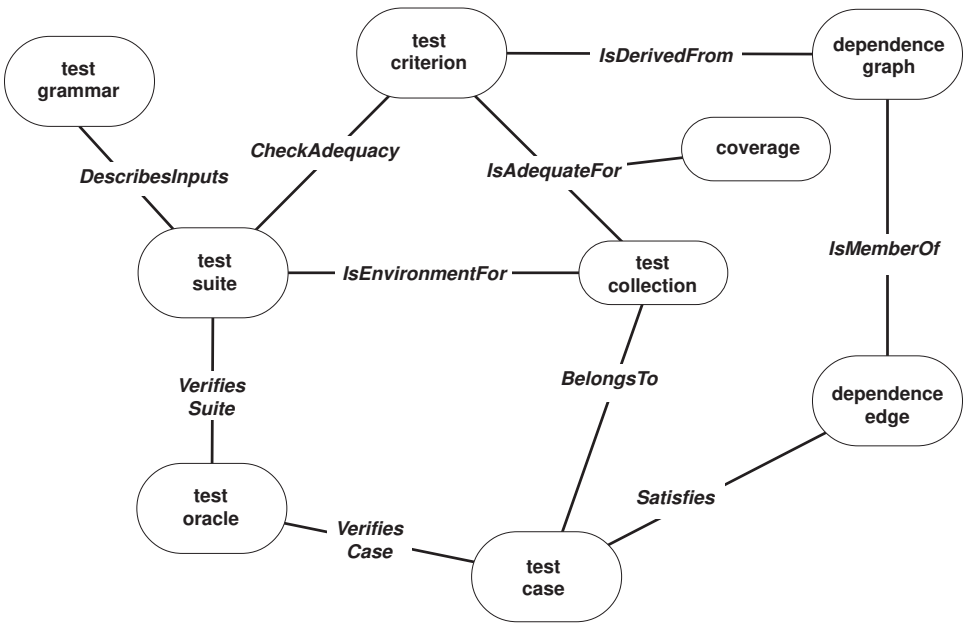

Figure 2: Test Artifacts

The TAOS user interface has several graphical editors that support editing and browsing of the test artifacts and also provides access to the TAOS toolkit, which automates various activities in the testing process and is integrated with analysis tools (the TAOS graphical user interface (GUI) is described in [RB93b]).

The primary artifacts managed by this process are summarized in Table 1. These artifacts are described more fully throughout this paper as they arise. The relations between several of these artifacts are shown in Figure 2. These relations are summarized in Table 2. More details on each relation are provided later in the paper as the related artifacts are discussed.

2.2 Program Dependence Analysis

During execution of a program, information flows between statements via variable values and conditional control of execution. The potential for information flow between statements is reflected syntactically by a program's control structure and its variable manipulation. These syntactic relationships between program statements define *program dependences*. Program dependences have been used in software development primarily as a basis for program optimization [KKL+81, FOW87], and program slicing for debugging [Wei82] and maintenance [HPR89]. Program dependences are also significant for program testing [Kor87, PC90]. During these activities, in particular, one is often interested in when a change in the semantics of one component affects the behavior of another component. These essential semantic relationships between components can be found by a syntactic analysis technique called program dependence analysis.

ProDAG (PROgram Dependence Analysis Graph) is a syntactic analysis toolset that supports the two basic syntactic dependence relations [FOW87, PC90]: [3] control dependence, which are features of a program's control structure, and data dependence, which are features of variable manipulations in a program. *Data dependence* holds, informally, between components s and s' if there is a sequence of assignments that potentially propagates data from a definition at s to a use at s'. *Control dependence* holds, informally, between components s and s' if s potentially decides whether s' is executed. ProDAG also supports chains of program dependences formed by the transitive closure of one or more dependence relations. *Syntactic dependence*, for instance, is the transitive closure of the data and control dependence relations, and holds, informally, between components s and s' if there is a chain of data dependences and/or control dependences from s to s'. ProDAG supports the syntactic program dependence relations identified in [PC90] and also other supporting graphs (see [ROM+93, ROMA92] for more detail).

The ProDAG Programmatic Interface. ProDAG is a basic analysis component designed using the TEAM design principles. Thus, we developed ProDAG not as a tool in its own right but as a general-purpose support capability for other analysis and testing tools and activities. ProDAG provides an application programmatic interface (API) to pro-

[3] Note that syntactic dependence is a necessary condition for semantic dependence; this implies that program dependences bound the components affected by the semantics of a given component, since absence of syntactic dependence implies absence of semantic dependence.

Artifact	Summary Description
test suite	test collection and associated characteristic information, such as rationale and executable, and test execution status
test case	test input data along with associated information, such as actual output, failure reports, behavior traces, and test execution state
test oracle	mechanism for verifying the correctness of execution behavior
test collection	collection of test cases
test grammar	grammar describing inputs to a component under test and used by the random test generator to create test cases
test criterion	requirements on the coverage of test execution
dependence graph (DG)	graph whose nodes are control flow graph nodes and whose edges represent dependence between two nodes
coverage	adequacy metrics for test criteria coverage by test execution
control flow graph (CFG)	graph whose nodes are statements and whose edges represent potential transfer of control
def-use graph (DUG)	CFG whose nodes are annotated with def-use information
IRIS graph (IRIS)	abstract syntax graph in Internal Representation including Semantics
instrumented executable	source code instrumented with monitoring probes
anomaly	potentially erroneous relationships between statements
failure	behavior of a test execution that does not meet the oracle
fault	syntactic component that causes a failure

Table 1: Test and Analysis Artifacts

Relation	Summary Description
IsEnvironmentFor	the test suite provides the environment of execution for a test collection (for each collection, there is a unique suite, but a suite may have multiple collections)
BelongsTo	the test case belongs to a test collection (and hence to the test suite that provides the collection's environment)
VerifiesSuite	the test oracle specifies the correct behavior for verifying execution of all test cases in the test suite
VerifiesCase	the test oracle specifies the correct behavior for verifying execution of the test case
DescribesInputs	the test grammar describes the test inputs for the test cases in the test suite
CheckAdequacy	the test suite should be checked for adequate coverage of the test criterion
IsAdequateFor	the test collection is adequate to cover the test criterion and coverage provides the metric information
IsDerivedFrom	the test criterion is derived from the dependence graph
IsMemberOf	the dependence edge is a member of the dependence graph
Satisfies	the test case satisfies (covers) the dependence egde

Table 2: Relations between Test and Analysis Artifacts

gram dependence relations. This abstract interface provides higher level tools and users with a uniform set of operations for creation of and access to program dependence relations as well as a standard mechanism for developing new dependence relations. ProDAG currently provides support for dependence analysis of Ada programs. ProDAG was designed, however, to be extremely language independent, and we are currently working on a version for C and C++. The design and implementation of this API is described in [ROMA92].

Conceptually, ProDAG represents a program dependence relation as a graph. In related work, the *program dependence graph (PDG)* [KKL+81, FOW87, HPR88] represents both data and strong control dependence in a single graph. The PDG has been used as an internal program representation to facilitate many software development activities [OO84, HR92]. ProDAG provides separate interfaces to each of the dependence relations above to support their independent use for testing, debugging and maintenance. ProDAG represents a program dependence relation as a *dependence graph (DG)* in terms of a *control flow graph (CFG)* and a *def-use graph (DUG)*, both of which use a language-independent abstract syntax graph representation, called IRIS.[4] A CFG describes the possible flow of control through a module; it consists of a set of nodes (typically representing simple statements) and a set of edges representing potential transfers of control. A DUG is an annotated CFG where *def-use attributes* associated with each node specify the definitions and uses of variables at the node. Control dependences are de-

[4] ProDAG currently analyzes Ada procedures. Before analysis, the Ada source is translated into IRIS and a CFG by the

fined entirely in terms of a CFG, while data and syntactic dependences are defined in terms of a DUG. A dependence relation is represented as a DG, where each relationship is represented as an edge between two CFG nodes.[5]

Figure 3 shows a CFG and the data dependence graph for the alarm clock program, where bold edges represent CFG edges and light edges represent data dependences. So, for instance, `Param` is defined at node 6 and used at node 10, thus the edge (6,10) represents a data dependence. Figure 4 shows a filtered weak syntactic dependence graph for the alarm clock program. Filtering capabilities are provided by the ProDAG user interface and allow the user to focus on particular subsets of dependences; in this case, the source filter is set on node 14 and hence only those syntactic dependences from node 14 are shown. Note, for instance, that (14,19) represents a data dependence, (19,21) represents a weak control dependence, and thus (14,21) represents a weak syntactic dependence.

The ProDAG API implements a graph-like interface, called the *dependence graph interface (DGI)*. In addition, ProDAG aids users in adding dependence graphs and defining DGIs specific to their own local needs. The ProDAG DGI provides functional capabilities to access graphs, build graphs, query graphs, and manipulate graphs and edges. Each DG is associated with the CFG from which it is derived. DGs may be stored in the artifact repository and hence made *persistent* across different invocations of ProDAG and thus may be accessed by multiple tools. The DGI is described in more detail in [ROM+93].

We are developing processes that perform testing, debugging, and maintenance based on program dependences. These processes invoke several tools that use the ProDAG API; the capabilities of many of these tools and their use of program dependences are described later in this paper.

ProDAG User Interface. We have developed graphical and textual user interfaces to ProDAG. The graphical user interface (GUI) provides graphical depictions of CFGs and DGs as well as a textual representation of the source code annotated with CFG node numbers. The depictions in Figures 3 and 4 are snapshots taken from the GUI we implemented with Chiron [KCTT91]; that GUI is described in [RB93a]. The GUI also allows the user to obtain various pieces of information associated with nodes and edges and provides the filtering capabilities mentioned above. The GUI is particularly useful for browsing dependences and software understanding.

Dependence Anomalies. For some dependence relations, certain syntactic patterns are indicative of an *anomaly*, which is a pattern that, although correct in terms of syntax and static semantics, may be evidence of a programming error. Data flow analysis finds data flow anomalies [OF76], such as a variable that is used before it is defined, while previous work in program dependence analysis has defined other types of anomalies [HT86]. ProDAG detects dependence anomalies while constructing the dependence graphs. Two anomalies detected in a data dependence graph are *undefined use*, which is a use for which there is no data dependence edge ending with that use node, and *unused definitions*, which is a definition for which there is no data dependence edge originating with that definition node. In addition, *useless statements* are detected in the syntactic dependence graph as a statement for which there is no output statement that is syntactically dependent on it [Kor87, PC90].

Path Representations. A major advantage of ProDAG is that it constructs the dependence graphs and supports their persistence rather than merely detecting anomalies, as is done by most static analysis tools. This enables the information to be used in a variety of ways, one of which is finding control flow paths to cover program dependences or to reach anomalies or measuring test coverage. For a dependence relationship, ProDAG develops a covering path representation, which consists of source and target node (the dependence edge) and nodes that must be avoided on a covering subpath from source to target, while constructing DGs.

Static and Dynamic Slices. A slice is a set of program statements that are relevant to the behavior of some selected statement [Wei82]. Program dependences form static program slices [OO84, HRB90]. The ProDAG GUI provides the ability to filter dependence graphs by selecting a particular source or target node, which provides forward and backward static slices, respectively. Forward slices depict the potential effect of a selected node; they are useful in identifying the potential ramifications of a change during maintenance. Backward slices depict the code that has potentially affected a selected node; they are useful in debugging faults that may have caused a failure revealed during testing. ProDAG is also capable of depicting either forward or backward dynamic slices (from a source or to a target node) based on trace information collected during test execution.

2.3 Test Development

TAOS provides a comprehensive set of capabilities for developing test artifacts. The TAOS GUI [RB93b], provides editors for creating test cases, test suites, and test oracles. TAOS also enables the automatic generation of many test inputs satisfying a specification. Furthermore, test criteria that specify requirements on test coverage can be derived

[5] By the terms relationship and relation, we mean a connection between entities and a set of such connections, respectively.

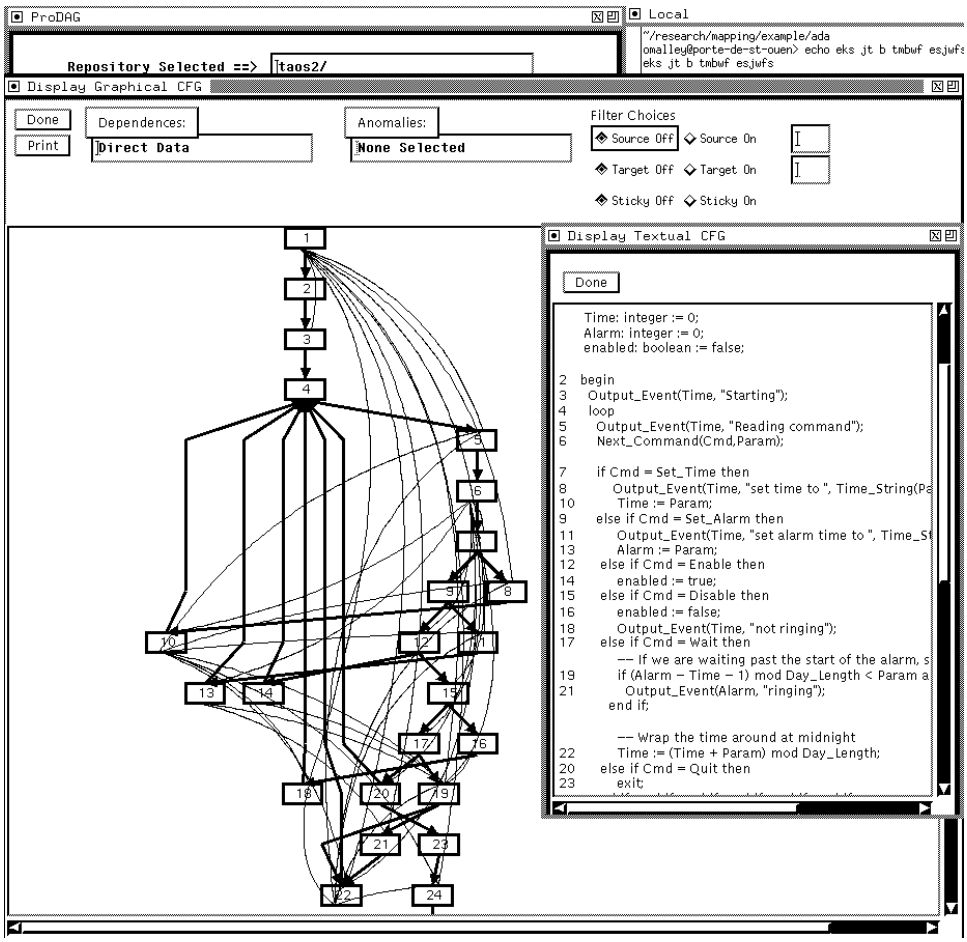

Figure 3: ProDAG Snapshot: Data Dependence of Alarm Clock

from associated tools. These artifacts are managed by the TAOS API and maintained in the artifact repository.

Test Suite and Test Case Creation. A test case is test input data along with associated information such as actual output, failure reports, execution traces, and the test execution state. A test suite is a grouping artifact for a collection of test cases that all have the same characteristic information. The characteristic information includes the test suite name, the rationale for the test suite, the component under test and the executable. The test suite provides the environment for execution of the test cases in the collection. There is a unique, primary test collection associated with each test suite, but there may be subcollections created for specific purposes, such as executing a specific subset of the test cases. A test suite also has status information associated with it, which incorporates the status of all the test cases in the collection.

Figure 5 shows TAOS' test suite and test case editors being used to browse test artifacts and test the alarm clock program. On the left is the test suite editor/browser, showing the characteristic information at the top and the status information at the bottom. In the middle is the test case editor/browser, with text canvases for test input, actual output, and status information below. On the right is the test oracle editor/browser, with selection of the oracle procedure (the GIL checker) and oracle information (a GIL specification of the alarm clock, whose textual translation is shown in the text canvas); GIL oracles will be explained further below. The pop-up window shows a failure report, which will be discussed later.

The primary test collection associated a test suite can either be created manually by use of the test case editor to create individual test cases or via the RandomTestGenerator.

To create a random test suite, the user specifies a test grammar to be used as well as parameters indicating the number of test cases to generate and a range for the length of the test event stream in each test case. The grammar must define `initialization`, the alternatives for a `test event`, and `cleanup`. A test input file is generated as a sequence of `initialization`, n repetitions of `test events`, and finally `cleanup`, where n is a random integer between the minimum and maximum length specified in the test suite

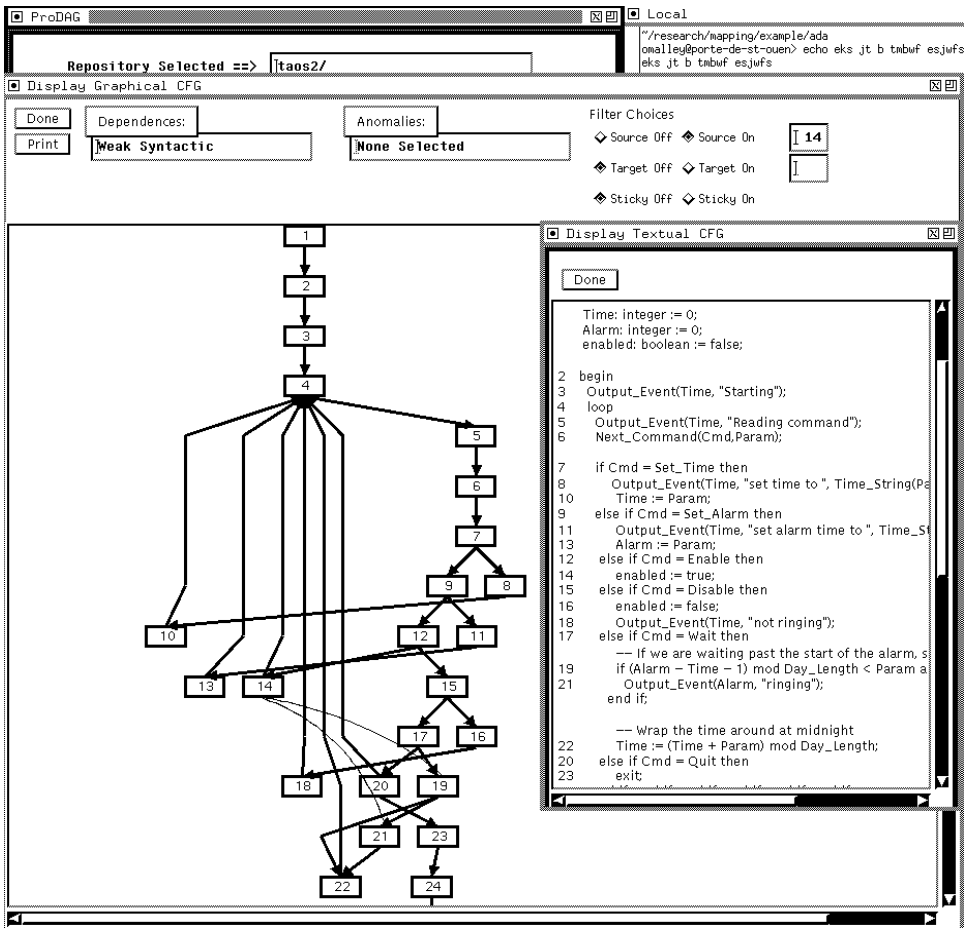

Figure 4: ProDAG Snapshot: Weak Syntactic Dependence of Alarm Clock

parameters. Figure 6 shows the test grammar for the alarm clock. Note that there are five events: `enable`, `disable`, `settime`, `setalarm`, and `wait`; `settime`, `setalarm`, and `wait` take `time` as a parameter.

Random test data generation has proven to be extremely useful, even though our simple test grammars are limited to context free grammars. We have recently integrated the DGL (Data Generation Language) system [Mau90], which enhances our test grammar capabilities to arbitrary grammars. We are investigating other classes of test grammars and test data generation for enhancing this capability.

Test Criteria Derivation. For testing, a test criterion consists of test data selection rules based on covering particular structures or elements of software. Several coverage criteria that are related to program dependences have been defined; e.g., data flow testing criteria [RW82, LK83, Nta84] require execution of paths that cover certain data dependences and may detect faults that cause incorrect variable definitions. Data dependence, however, is neither sufficient nor necessary for detection of such faults; rather, exercising complex chains of data and control dependence may be necessary to reveal incorrect variable definitions [TRC92]. Thus, ProDAG provides more general information than a data flow tool would provide for testing.

TAOS supports dependence coverage criteria by describing a test criterion in terms of a dependence graph. This capability enables the definition of several structural coverage criteria, including all-statements (nodes), all-branches (edges), and a variety of data flow criteria such as all-defs and all-uses [RW82]. Extensions also allow the definition of criteria for more sophisticated data flow criteria [RW82, LK83, Nta84]. In addition, very specific dependence coverage criteria can be defined by selecting a subset of a dependence graph (easily supported with the use of graph filters in the ProDAG GUI). TAOS' ability to measure coverage is described in Section 2.5.

We have designed the test criteria capacity within TAOS to be general enough to support future enhancement including test criteria based on requirements, specifications, and design. This supports our long-term interest in specification-based testing and test selection.

Test Oracle Creation. A test oracle is a mechanism for

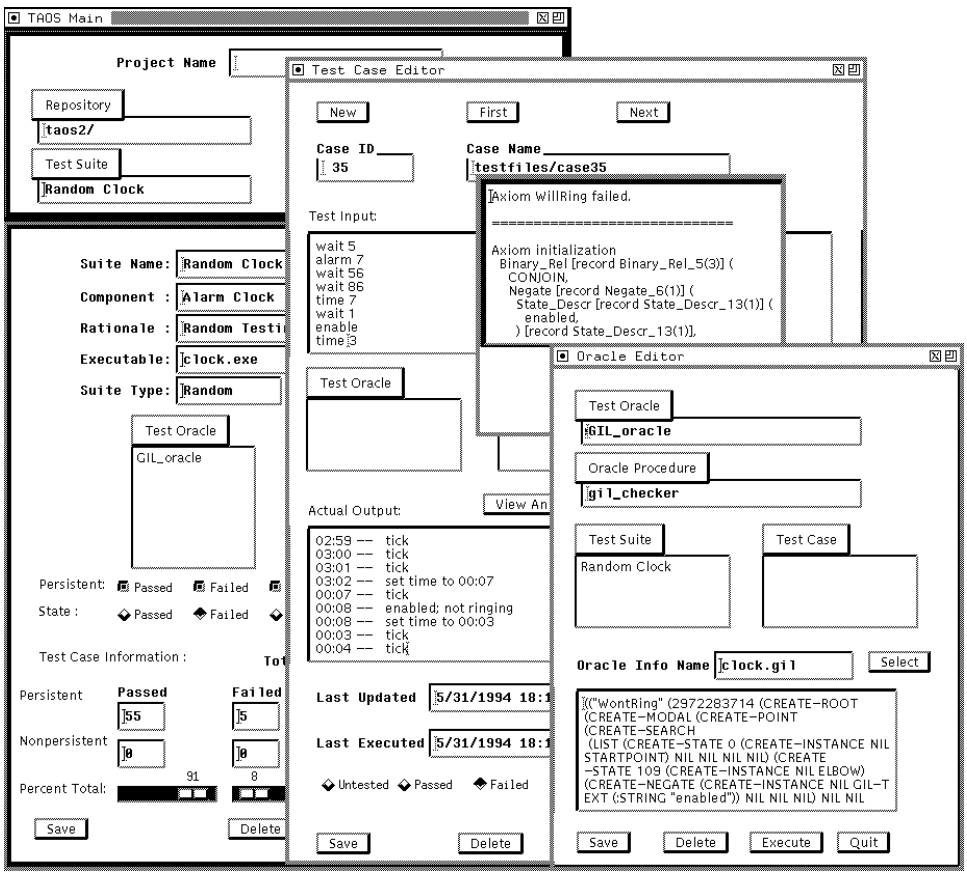

Figure 5: TAOS Snapshot: Testing of Alarm Clock

```
<digit>::=0|1|2|3|4|5|6|7|8|9
<time>::=<digit>|<digit><digit>
<enable>::=enable
<disable>::=disable
<settime>::=time <time>
<setalarm>::=alarm <time>
<wait>::=wait <time>
<others>::=<enable>|<enable>|<disable>|
          <settime>|<setalarm>
<command>::=<others>|<wait>
<initialization>::=||||wait 1400\n
<testcase>::=<command>\n
<cleanup>::=wait 100\nquit\n
```

Figure 6: Test Grammar for Alarm Clock

specifying correct and/or expected behavior and verifying that test executions meet that specification. Testing is of little use if behavioral correctness is not verified. Testing research has, for the most part, neglected the issue of oracles. Most testing methods do not check behavioral results, but focus only on defining what to test, thereby ignoring the test oracle and requiring manual checking of test results. In that most test criteria require an overwhelming number of test cases, manual checking can severely hamper the testing process — the test executions may be run, yet the goals of testing are not achieved since results may be checked only haphazardly.

TAOS supports creation and maintenance of test oracles and their use in determining whether execution behavior adheres to the behavior specified by the oracle. A test oracle consists of an oracle procedure and oracle information. An oracle procedure can be any executable. The oracle information is data that the oracle procedure uses (along with the test input, output, and execution trace) to verify the execution behavior. An oracle procedure/information pair can be associated with either an entire test suite (via the VerifiesSuite relation) or an individual test case (via the VerifiesCase relation); in addition, there may be more than one oracle associated with a test suite or test case.

TAOS supports a wide range of generic test oracle procedures, which may be specialized for the testing at hand by specifying the oracle information. In addition, more specific test oracles may be developed by the tester or developer using an executable specification or programming language. It is important to realize that behavioral verification is only as good as the test oracle(s) being used. There is a trade-

off between the effort involved in specifying an oracle and the accuracy of behavior verification. Some typical pairs of <oracle procedure/oracle information>, which vary widely in both effort and accuracy, are:

- Diff: <Diff_Checker / expected output> is the well-known input/outcome oracle (where Diff_Checker is similar to the unix diff program) is the most accurate but requires the tester to develop oracle information for each test case.
- Manual: <Manual_Checker / interactive decision> allows the user to interactively decide whether the results are correct; we have enhanced this oracle by providing the capability to save the output results as oracle information and change the oracle to Diff when the output results are determined to be correct (thus, the Diff oracle can be used any time this test case is retested – e.g., during regression testing); This oracle is potentially the most error-prone.
- Range: <Range_Checker / output range specification> is a fairly inaccurate oracle, supporting the verification of ranges on the outputs, but requires minimal effort to develop.
- GIL: <GIL_Checker / Graphical Interval Logic specification> allows the user to graphically specify safety, liveness, and temporal properties (usually for a test suite), which are then checked after test execution; this is a very powerful oracle, but requires knowledge of the GIL specification language.
- RTIL: <RTIL_Checker / Real-time Interval Logic specification> enables the user to express and verify complex safety, liveness, and temporal properties (usually for a test suite). This is the most powerful test oracle, but developing test oracles then requires expertise in RTIL.

Manual_Checker, Diff_Checker, and Range_Checker are standard oracles that are supplied with TAOS. We have recently prototyped GIL_Checker and expect to prototype RTIL_Checker soon.

Figure 7 provides a Graphical Interval Logic [DKMS+92] specification for the alarm clock. This illustrates one source of our specification-based oracles. The specifications consists of three axioms. Initialization states that initially the alarm is not enabled and the bell is not ringing. Will_Ring states that from the point at which Time equals the Alarm time and the alarm is enabled until the point at which the alarm is not enabled, the bell is ringing. Won't_Ring states that from the point at which the alarm is not enabled until the point at which the alarm is enabled and the Time equals the Alarm time, the bell is not ringing.

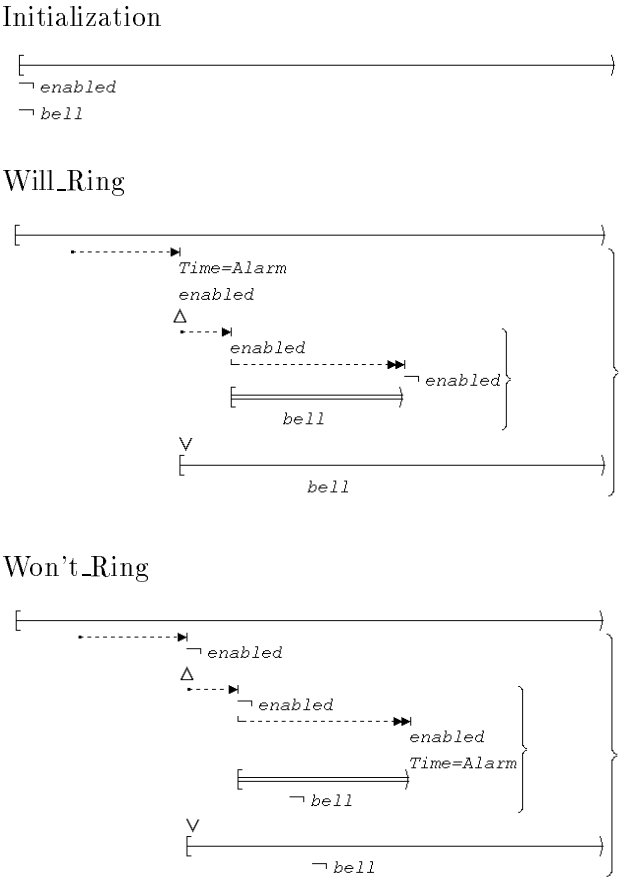

Figure 7: GIL Specification and Oracle for Alarm Clock

We are investing much of our research in specification-based oracles. We have defined a method for deriving test oracles from specifications [RAO92]. We have experimented with this approach in the context of TAOS by first encoding specification-based oracles in a very high-level language (ICON). Then we developed the checker for Graphical Interval Logic (GIL) oracles and are in the process of incorporating a trace checker for Real-Time Interval Logic (RTIL) oracles. We are developing further approaches for deriving specification-based oracles based on specification slices and expect to provide capabilities for oracles based on the Z notation in the near future.

2.4 Test Execution

TAOS automates the test execution process by following the preferences specified by the tester and retrieving the appropriate information in the test artifact repository. The behavior of each test execution is verified based on the oracles associated with the artifact being executed.

Parallel, Monitored Execution. TAOS executes test cases in separate processes so as to enable concurrent execution of multiple test cases. The TAOS GUI allows the user to initiate execution of an individual test case or a test suite (i.e., the test cases in a test suite) with various param-

eters choosing which test cases to execute and when to stop. When a test suite is selected for execution, the ParallelTestExecutor spawns several processes to execute multiple test cases parallel (the number of concurrent execution processes is selectable).

Monitored test execution is achieved by instrumenting the source code with the Artemis tool. Artemis allows the user to select the "level" of instrumentation and the instrumented executable produces a trace of the execution behavior. The choices currently include statement, branch, procedure call, task entry and exception; we are enhancing the Artemis capabilities to include tracing of specified events as well as variable manipulation and values. Artemis is built using the Language Processing Toolset (LPT) and thus only supports instrumentation of Ada at this time. The same instrumentation capabilities could, however, be provided for other languages. These traces are used for behavior verification and test coverage measurement.

Components to be tested under TAOS may be developed in any language; TAOS simply invokes the executable.[6] TAOS allows the user to test any executable, which may be a single procedure during unit testing, a set of integrated components during integration testing, or the entire system during software system testing. TAOS does not currently provide a test harness environment, thus the user must develop the driver and stubs for unit and integration testing as necessary. Using the LPT capabilities, we are developing a TestHarnessGenerator that will generate drivers and stubs for Ada programs.

Behavior Verification. Automated behavior verification is one of the innovative claims of TAOS. Most testing tools provide limited, if any, support for test oracles and behavior verification. In particular, some provide the ability to specify expected output for specific test inputs or they may provide capture/playback capabilities that are useful in retesting. Very few capabilities are provided for first time test execution or for general specification of expected behavior. This is a major shortcoming in testing tools, as leaving behavior verification totally to the user can be extremely error prone.

After execution of each test case, TAOS applies all oracle procedures associated with the test case and/or the test suite (providing the environment for the test case) using the affiliated oracle information. The oracle procedure basically compares the execution trace, input, and output with the oracle information and determines if this test case execution has passed or failed. If the test case fails, a failure report generated by the test oracle is associated with the test case. The test status is associated with the test case, and a cumulative summary of the test case statuses are maintained with the test suite.

Referring back to Figure 5, the pop-up window shows a failure report, which indicates that the square root procedure failed for this particular test case, as indicated also by the test status. In this particular case, the Will_Ring axiom has been violated; note in the input that the alarm enabled and then set to 7, after which the time is set to 7, yet in the output the alarm does not start to ring. This is a fault in the implementation, which does not check to see if the alarm is equal to the time at which the clock is being set. Looking at the test suite editor/browser, we see that 91% of the test cases in the selected test suite passed and 8% failed.

2.5 Test Measurement

Accurate process and product measurement is required to support continuous improvement of products and processes. Thus, TAOS supports both static and dynamic measurements of both software processes and the products they build. Automation of the testing process supports automatic metric collection. TAOS currently performs coverage measurement and limited quality assessment. Yet, we have developed TAOS with the hooks in place to collect more metrics and do more sophisticated measurement.

Coverage Measurement. A test criterion does not explicitly define test cases by actual inputs but rather describes the requirements on test inputs or test execution. TAOS uses these descriptions to measure whether and how much of a test criterion is adequately satisfied. A test criterion may be paired with one or more test suites by the CheckAdequacy relation. When a test suite is executed, the CoverageAnalyzer determines whether the test suite is to be checked against any test criterion. If it is related, then the execution traces produced by executing each test case in the suite are compared with each element of the test criterion (e.g., a dependence edge) to determine which, if any, criterion elements were covered by the test execution. Each test case is associated with any criterion element that it satisfies (covers) and also becomes a member of the test collection that IsAdequateFor the test criterion. It is important to realize that the instrumentation must be sufficient to determine whether the test criterion is satisfied.

Our primary capability for coverage measurement at this time is based on program dependences. The path representations created by ProDAG provide the information required to check execution traces for test case coverage. For these structural test coverage criteria developed in conjunction with ProDAG, test coverage can be viewed while browsing the dependence graph from which the criterion was derived.

[6] As mentioned, some of the sophisticated capabilities, such as program dependence analysis and coverage measurement, are limited to Ada components due to our current language processing capabilities.

Quality Assessment. TAOS collects basic metrics during the testing process to help assess the quality of the software under test. At this point, TAOS measures failures, dependences, execution counts, dependences covered, and ratios of these basic metrics (such as failures/execution and failures/dependence covered (or criterion element), and dependence covered/execution). TAOS provides simple plotting capabilities for these metrics.

TAOS can use such metric collection to empirically guide testing, debugging, and maintenance processes and focus further efforts on high-payoff areas. For instance, as ProDAG is both computationally feasible and performs fine-grained analysis, it can be used both as an analysis component that is focused by static metric-based techniques (such as source lines and cyclomatic complexity) and as a metric that focuses more sophisticated analysis and testing on modules prone to dependence faults.

3 Testing, Debugging, and Maintenance

A process model of testing, debugging and maintenance based on dependence analysis and supported by TAOS appears in Figure 8. The picture shows a IDEF-like diagram, where the boxes represent activities, arrows entering boxes from the left represent activity inputs, and arrows exiting boxes to the right represent activity outputs (the controls, which would normally come in from the top, and the resources/mechanisms, which would normally come in from the bottom, have been omitted for clarity). The artifacts referenced as inputs and outputs are those summarized in Table 1.

The process described is based on program dependences, because program dependences represent the essential semantic relationships between program components and should be taken into account when testing, debugging, or maintaining software. Research has shown that program dependences have implications for software testing, debugging and maintenance [Kor87, PC90, TRC92]. ProDAG and the TAOS toolkit can be used independently or in an integrated fashion as modeled in the process described here.

Dependence Analysis. The process begins by performing dependence analysis of the component to be validated and tested. The Ada source code is translated into an internal representation and control flow graph. Through the ProDAG GUI, the user selects the type of dependence analysis, and the dependence graph is constructed. A graphical depiction of the dependence graph can be browsed and filtered to analyze the component's structure. Dependence analysis may discover dependence anomalies, which can also be viewed and should be corrected.

Based upon the dependence graph, the user can create a test criterion, which provides structural testing requirements for dependence coverage. The test criterion does not contain actual test inputs, but rather requirements for test case execution to cover each dependence. This test criterion will be compared to traces from test execution of other test suites.

[Dependence Coverage] Testing. The next activity in the process is testing. Test suites are created and modified with the TAOS test suite editor. The test cases in a test suite contain actual input data and can be created either using the test case editors or with the random test data generator. The random test data generator uses a test grammar describing the component's inputs and generates a specified number of test cases. A CheckAdequacy relation may then be created linking a test suite to a test criterion. During execution of the test suite, execution profiles will be compared against the test case requirements of the test criterion to determine the extent to which required dependences have been covered. Test coverage can be viewed with the ProDAG GUI by browsing the dependence graph from which the criterion was created. Based on the coverage reports, the user may choose to develop another test suite or another test criterion or that testing of this component is complete.

Testing must also include behavior verification. As described, TAOS supports automated behavior verification through the use of test oracles. For each test oracle associated with a test case (or the test suite providing the environment for the enclosing test collection), the test inputs, outputs and execution trace are compared by the oracle procedure with the oracle information. When test execution behavior does not satisfy an associated oracle, a failure report is generated.

Debugging. When a failure is detected during testing, dependence graphs can be used to aid in locating the fault that caused the failure. Using the filtering capabilities provided by the ProDAG GUI, the user can select the failure node as the target node and then see only the dependences to the failure. Filtering the syntactic dependence graph in this way is tantamount to using static slices to locate faults in debugging [Wei82], since once a failure is revealed only those statements that the failure is syntactically dependent on could have caused the failure [PC90]. The components upon which the failure is dependent can be analyzed to determine if they are faulty. With ProDAG, various types of dependences may be used to focus the search. In addition, the execution traces can also be filtered to determine dynamic slices, the dependences that have actually been executed and which may have led to a revealed failure.

Maintenance and Regression Analysis. Dependence graphs are also useful in maintenance and regression anal-

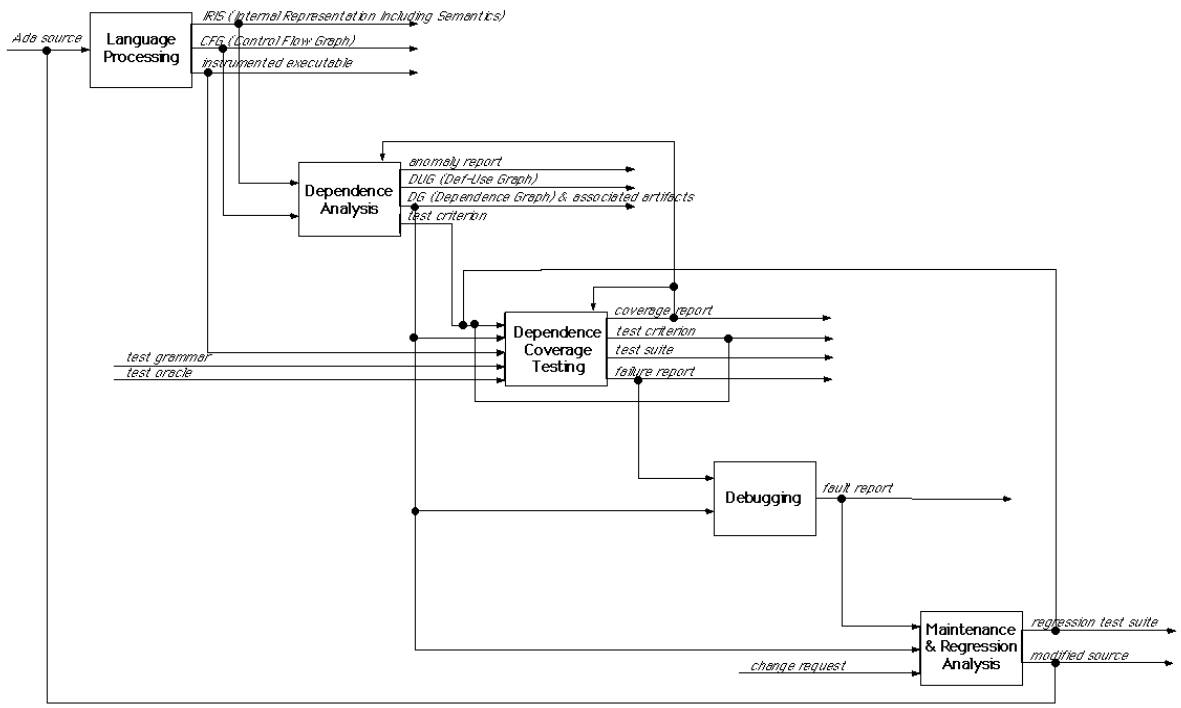

Figure 8: Testing, Debugging and Maintenance

ysis. Static slices (program dependences) identify the components that may be affected by a change and thus must be analyzed and tested to ensure that the change does not have adverse affects. Only the statements that are syntactic dependent on modified statements could be affected by a modification. Likewise, only those statements upon which the modified statement is syntactic dependent could affect the modified statement. Thus, syntactic dependence (in both directions) identifies the statements that must be regression tested after a modification. When a change is requested, the ProDAG GUI filtering capability can be used to select the node to be changed as the source node and see only the dependence from the change. The effect of the change can then be analyzed to determine what ramifications it might have or what other statements must be modified.

These filtered dependences can also be used to create a test suite for regression testing. The purpose of regression testing is to ensure that no other part of the program regresses (to a point where it is does not behave correctly) when a change was made. Thus, only the nodes dependent on the changed node could be affected by the change. These dependences should be retested after the modification to ensure that the change did not have an adverse affect on the rest of the program. In addition, if test suites have been previously executed for the component being modified, the previous test results, if correct, can be used as oracle information for those test cases that we expect to behave as they did before the modification.

4 Conclusion

4.1 Experience with TAOS

We have been participating in a pilot project that is using TAOS, ProDAG, and the process defined in Section 3 in an industrial setting. We transitioned TAOS and ProDAG into use in a software organization at Hughes Aircraft Company about six months ago. The objectives of the pilot project are to noticeably improve the testing process within the organization and to provide feedback on how the technology and tools could be made more useful for testing organizations throughout the company. The long-term goal is to provide a discipline for effective testing of large-scale, critical, software-intensive systems for software development organizations. This eventual goal will require continued evaluation of available testing technology as well as the transitioned toolkits.

The pilot project involves the development of a library of reusable components for a domain-specific software architecture for new radar signal processing applications. The architectural design is object-oriented and there are currently 37 reusable Ada packages that are being analyzed with ProDAG and thoroughly tested using TAOS. The test artifacts are being maintained for further empirical evaluation as the project continues. There have been four people working with the project. Although this is a small project, it is real software

to be transitioned to multiple product line software systems.

The project personnel working with TAOS found the toolkit easy to learn and gradually work into their daily routine and gradually change their traditional way of testing. They found the capacity for program understanding to be greatly enhanced due to the ability to graphically analyze software and graphically represent the testing process. The organization found the most improvement to their process arose from the automated generation of test data and the automatic verification of test execution behavior. These aspects helped to ensure that the software was more thoroughly tested.

Based on the project's experience with TAOS and ProDAG, the lead technical person determined that they achieved a potential savings around 25%, spread across the activities of test development, test execution, and fault isolation. In addition to these dramatic savings, several enhancements, which we believe will afford even greater benefit, have been identified. These are capabilities that we intend to add to future versions of the TAOS toolkit and are discussed in the next section.

4.2 Future Work

Our plans for future work include additional experimentation with TAOS, ProDAG, and the processes we have built upon these tools. In addition, we have identified a number of adaptations and enhancements to the tools that would be beneficial. Finally, we are working toward the specification of production-quality support for analysis, testing, debugging, and maintenance; we are trying to determine what basic capabilities are missing from our current toolkit.

Experimentation. We have yet to perform significant, empirical evaluation of the effectiveness of TAOS, ProDAG, or processes based upon them. We hope to be able to do so in the next year in the context of our current project and potentially on other projects. In particular, we would like to collect various data and evaluate the effectiveness of dependence coverage and automated oracles by comparing the effectiveness of the newly developed process to the organization s traditional process on previously tested code. This would be an extension of the limited evaluation we have done on the current project. We hope to answer such questions as: How much more test data is required for comprehensive, dependence coverage? How much less costly is automated test execution and behavior verification? How many more failures are detected? How does effectiveness vary with type of dependence? How much time is saved in generating random test data over manual test cases? How effective is random test data versus manual test cases? How difficult is it to write useful test oracles? How many more failures are detected with automated oracles than manual behavior verification?

Enhancements to ProDAG. We have several planned enhancements to the ProDAG toolset. First, we intend to add intercomponent dependence analysis. This will then support the identification of program dependences between procedures and tasks and allow the user to focus their analysis at different levels. We would like to add incremental capabilities for dependence analysis, which would make analysis of evolving software more efficient. However, we do not anticipate making ProDAG incremental until such time as the language processing capabilities upon which it depends are incremental.

We developed ProDAG atop an internal representation that is extensible to multiple, diverse languages. This promotes the language independence of ProDAG. We are working on extending ProDAG to other programming languages, particularly C and C++. More importantly, we believe that we can extend ProDAG to formal specification languages. Our long term interest in specification-based testing would be furthered by such capabilities. We are developing techniques for specification slicing whose implementation would require such a capability [CR94].

Enhancements to TAOS. One basic improvement to the TAOS toolkit that we are currently working on is a TestHarnessGenerator. This tool would take the specification of a component under test and generate a driver to set up and invoke the component, stubs for any procedures called by the component that are not to be tested simultaneously, a test grammar describing the inputs to the component and from which a random test suite could be generated, and a template oracle that could be specialized and used for behavior verification.

With the implementation of intercomponent dependence analysis, we intend to implement dependence-based integration testing criteria. Moreover, we are currently developing a proactive regression testing process that uses program dependences to determine what must be retested, what test cases and suites are available for reuse, and automatically initiates regression testing triggered by modification.

Finally, we are continuing our long term research interest in specification-based testing. As mentioned previously, we intend to develop program dependence analysis capabilities for one or more specification languages. This will support not only the specification slicing discussed above, but also specification testing and specification-based oracles based on slices. Moreover, we intend to further integrate TAOS with formal methods so that there are additional capabilities to develop and use specification-based oracles with TAOS.

Concluding Remarks. The critical nature of current software applications indicates the need for more powerful testing technology. Through our development and experience with TAOS and ProDAG, we have shown that advanced testing technology can be transitioned and used effectively within industrial software development organizations. This process achieves a less human-intensive activity, more comprehensive test coverage, higher failure detection rates, and more accurate behavior verification, resulting in increased testing productivity, higher quality software, and lower maintenance costs. In short, the process will improve development and maintenance cost and schedule, as well as software quality. The process was developed in conjunction with a particular project, but we believe that with slight tailoring, it can be transitioned and can bring with it major improvement for most traditional software development organizations.

References

[CR94] Juei Chang and Debra J. Richardson. Static and Dynamic Specification Slicing. In *Proceedings of the Fourth Annual Irvine Software Symposium*, pages 25–37, April 1994.

[CRZ88] Lori A. Clarke, Debra J. Richardson, and Steven 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).

[DKMS+92] Laura K. Dillon, George Kutty, P. Michael Melliar-Smith, Louise E. Moser, and Y.S. Ramakrishna. Graphical specifications for concurrent software systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 214–224, Melbourne, Australia, May 1992.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 146–157, January 1988.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411. ACM Press, May 1992.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HT86] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[KCTT91] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[KKL+81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 207–218. ACM Press, 1981.

[Kor87] B. Korel. The program dependence graph in static program testing. *Information Processing Letters*, 24:103–108, January 1987.

[LK83] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[Mau90] Peter M. Maurer. Generating test data with enhanced context free grammars. *IEEE Software*, 7(4):50–56, July 1990.

[Nta84] Simeon C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[OF76] Leon J. Osterweil and Lloyd D. Fosdick. DAVE – a validation, error detection, and documentation system for FORTRAN programs. *Software — Practice & Experience*, 6:473–486, 1976.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN*, 19(5):177–184, May 1984.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.

[RB93a] Debra J. Richardson and Bach Bui. Prodag graphical user interface manual. UCI–ICS Technical Report TR-93-12, Department of Information and Computer Science, University of California, Irvine, August 1993.

[RB93b] Debra J. Richardson and Bach Bui. Taos graphical user interface manual. UCI–ICS Technical Report TR-93-11, Department of Information and Computer Science, University of California, Irvine, August 1993.

[ROM+93] Debra J. Richardson, T. Owen O'Malley, Cynthia Tittle Moore, Stephanie H. Leif Aha, and Debra A. Brodbeck. ProDAG: An application programmatic interface for program dependence analysis graphs. Technical Report UCI-93-10, Department of Information and Computer Science, University of California, 1993.

[ROMA92] Debra J. Richardson, T. Owen O'Malley, Cindy Tittle Moore, and Stephanie Leif Aha. Developing and Integrating ProDAG in the Arcadia Environment. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 109–119, Washington, D.C., December 1992.

[RW82] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 272–278, Tokyo, Japan, September 1982.

[TC93] Peri Tarr and Lori A. Clarke. Pleiades: 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.

[TRC92] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. Information flow transfer in the Relay model. Technical Report TR-92-39, Department of Information and Computer Science, University of California, May 1992.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.