TEAM: A Support Environment for Testing, Evaluation, and Analysis

Lori A. Clarke*, Debra J. Richardson†, and Steven J. Zeil*

COINS Technical Report 88-41

May 1988

*Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

†Information and Computer Science Department
University of California
Irvine, California 92717

Arcadia Document UM-88-03

This work was supported by grants DCR-8404217 from the National Science Foundation, CCR-8704478 from the National Science Foundation with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6104), 84M103 from Control Data Corporation, and RADC grant F30602-86-C-0006.

Abstract

Current research indicates that software reliability needs to be achieved through the careful integration of a number of diverse testing and analysis techniques. To address this need, the TEAM environment provides a framework for the extensible integration of and experimentation with software testing and analysis tools. To achieve this flexibility, we exploited three design principles: component technology so that common underlying functionality is recognized; generic realizations so that these common functions can be instantiated as diversely as possible; and language independence so that tools can work on multiple languages, even allowing some tools to be applicable to different phases of the software lifecycle. The result is an environment that contains building blocks for easily constructing and experimenting with new testing and analysis techniques. Although the first prototype has just recently been implemented, we feel it demonstrates how powerful modularity, genericity, and language independence can further extensibility and integration.

1 Introduction

As computer systems are being used increasingly in life critical applications, the quest for highly reliable software is becoming paramount. With this quest arises the need for a validation process that is capable of guaranteeing high reliability. Testing is the most common approach to assessing software reliability, but for the most part testing has hitherto been an extremely human intensive and ad hoc process with limited results. To achieve the high reliability required of today's software applications, a powerful environment that automates sophisticated testing and analysis techniques is needed. We are developing such an environment, called TEAM for Testing, Evaluation, and Analysis Medley. The TEAM environment strives to support:

- *integration* of diverse testing and analysis tools,
- *extensibility* of that tool set so that new tools can easily be added,
- *experimentation* with different approaches to software reliability, and
- *full lifecycle* testing and analysis.

It is clear that no single testing or analysis technique alone can provide the reliability assurance that is being sought for software. Instead, careful *integration* of a variety of techniques is required so that complementary techniques can exploit each other's strengths effectively. Exactly which techniques provide complementary capabilities is not totally known at this time. To address this issue, we have investigated several different techniques to understand their relative strengths and weaknesses. We have performed a graph-theoretic analysis of various path selection criteria [9], an algebraic analysis of the error detection capabilities of fault-based test data selection criteria [29], and a comparison of the selectivity of classes of testing approaches [40]. In addition there have been several studies into how to combine specific testing techniques, such as dynamic analysis with data flow [26] and concurrency analysis with symbolic evaluation [13,37]. Such work clearly indicates that, to be effective, TEAM needs to support a variety of testing techniques with diverse foci. Moreover, these techniques must be tightly integrated so that they can efficiently work together.

Our investigations have led to the selection of a preliminary set of testing tools to be included in the initial TEAM environment. We expect, however, that as testing and analysis research progresses, the set of desired capabilities and corresponding tool set will change. Hence, we have designed the TEAM environment to be *extensible* so as to facilitate additions and modifications to the testing tool set.

Most evaluation of testing techniques has thus far been of an analytical nature. These theoretical studies have provided considerable insight into the relationships among testing techniques, but have generated no empirical evidence of their relative value. Empirical studies are particularly important in testing research since often worse case analysis can lead to very different conclusions than experimental studies of typical operational performance (e.g., [3]). The TEAM environment allows us to *experiment* with different testing techniques and to modify the desired tool set according to the experimental results. Experimentation of this sort requires that it be relatively easy to add a new tool or to change the configuration of how certain tools interact. Therefore, TEAM has

been designed to facilitate the rapid prototyping of testing techniques and, most importantly, combinations of testing techniques.

It has been repeatedly shown that it is more effective to detect and correct an error as early in the development process as possible. Thus, we have had a long term interest in the development of testing and analysis techniques that are applicable throughout the software development and maintenance process. There have been several interesting analysis techniques proposed that attempt to address pre-implementation testing [16,18,28]. We are striving to support testing and analysis throughout the *full software lifecycle* within the TEAM environment.

This paper presents the design of our testing environment and describes how TEAM meets the stated goals. The next section provides an overview of our approach and describes the high-level architecture. Section 3 describes each of the major components of the TEAM environment and the ramifications of our design decisions. In the conclusion, we discuss the status of TEAM, what we have learned in the process of developing the environment, and our future plans.

2 The TEAM Architecture

The TEAM environment is designed to support the integration of and experimentation with an ever growing number of diverse and powerful testing and analysis tools applicable throughout the full software lifecycle. These goals have led to three important aspects in our design:

- identification of essential components and their interfaces;
- recognition of as many generic capabilities as possible;
- support for language independence.

We have identified a number of underlying analysis capabilities upon which many testing techniques rely. These ***basic analysis components*** provide the building blocks for the creation and integration of more sophisticated testing techniques within the TEAM environment. Often an advanced testing tool can be realized solely by compositions of interacting basic components. Creating more sophisticated tool capabilities out of less complex, more general components is a powerful and effective approach. A judicious choice of components provides the potential for reuse in a number of larger tools, thereby facilitating rapid prototyping and lower development costs for new tools. Moreover, we have observed that having first hand experience with such component technology encourages tool designers to develop good modular decomposition for their own tools, thereby extending the component tool library.

In addition to designing TEAM so that basic tool capabilities have been identified as system components, we have also designed the components as generic capabilities whenever possible. Generic components provide capabilities that can be instantiated to meet the needs of different testing tools in the environment. To accomplish this generality, a generic component's interface must be designed to capture the essence of its capabilities and to provide an appropriate means for other tools to tailor and fully exploit its features. The design of such generic components substantially

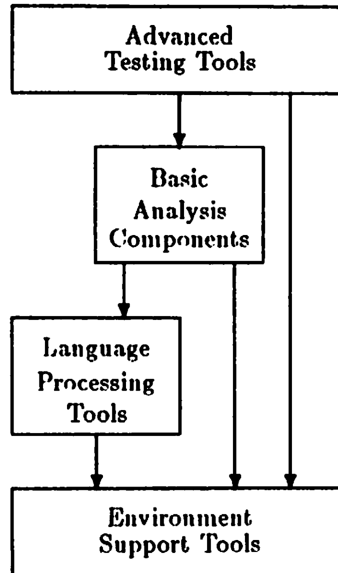

Figure 1: Layered Architecture of TEAM

increases the initial development overhead of a project; it is much harder to design generic tools than more specialized tools. We have found, however, that we gain substantial leverage from these generic components and, thus, believe the overhead is worth the effort.

To support and further new testing techniques, particularly those that address the pre-implementation phases of the lifecycle, it is important that the TEAM environment support an assortment of languages, including specification, design, and coding languages. To accomplish this, we have designed the system to be relatively language independent.

Of course, modularity, genericity, and even language-independence have been advocated for software development for some time. Developers often do not, however, see the extent to which they can exploit these ideas. In TEAM we have demonstrated how we have employed these ideas to develop a rather novel environment that we believe will greatly improve the state-of-the-art in software testing as well as provide a valuable platform for future research in software testing and analysis.

We have selected an environment architecture that provides layers of capabilities designed to support the implementation of advanced testing techniques. This architecture is shown in figure 1. The boxes represent the different abstract machines or layers of the system, while the arrows indicate which layers make use of which other layers. The architecture layers are briefly described below and each is elaborated on in the next section.

The highest layer in the architecture is the *advanced testing tools*. As noted, these tools are primarily composed of the underlying components. Although we currently only provide a few

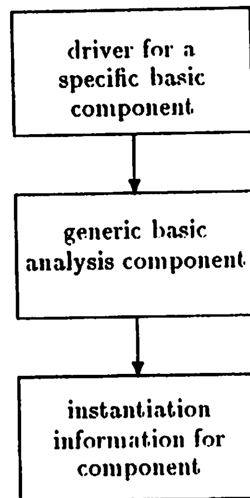

Figure 2: Decomposition of a Basic Analysis Component

instances of advanced tools, it is envisioned that this layer will eventually contain a large number of such tools. For example, we would like to see this layer support the RELAY model for integrating test data selection and path selection techniques [30] and the EQUATE model of test data evaluation [39]. We currently know of no proposed testing techniques that could not be naturally implemented in the TEAM framework.

At the next level in the architecture are those capabilities provided as ***basic analysis components***. To date we have recognized three basic analysis components: ***interpretation***, ***data flow analysis***, and ***reasoning***. We would like each component in this layer to be very general, providing capabilities required by many of the more advanced tools. Moreover, since the intended use of one of these components may vary from one advanced testing tool to another or from one model of computation to another, each should be designed to be extremely adaptable. We advocate an approach whereby the basic underlying functionality of the basic analysis component is recognized and separated from the more variable subparts, which must be instantiated depending on the desired use. Figure 2 shows this decomposition. While all future basic analysis components may not lend themselves to this particular decomposition, this seems to be an ideal design to be employed whenever possible. As is discussed in the next section, we have exploited this design to the fullest in the development of the interpreter and are extremely pleased with the adaptability we have achieved.

The ***language processing*** layer minimizes the impact of multiple source languages on the implementation of testing tools. This layer is used to translate all software into a common internal representation, which was chosen because it provides a substantial degree of language independence. We have carefully designed the higher level tools so that they employ this internal representation without regard to the source language. Thus, our environment supports analysis and testing for multiple languages, and some tools are even applicable to languages from different phases in the lifecycle. Moreover, the language processing layer of TEAM provides the typical lexical and syn-

tactic analysis tools so it is easy to extend the TEAM environment to recognize and analyze new languages.

To support the integration of components, the lowest level of the TEAM architecture consists of an *environment support* layer, which manages the sharing of objects between components. Each component manipulates new or existing objects, and the interaction between components is through these objects. The major requirement of this layer is to support the creation, storing, retrieving, and sharing of the kinds of complex objects required by the testing tools.

3 Environment Description

The TEAM environment architecture of Figure 1 has been motivated by our study of the requirements for the support of advanced testing and analysis tools, although we believe that this architecture is likely to be useful in other environments as well. In this Section, we explore these requirements and their effect on the environment design.

3.1 Advanced Testing and Analysis Tools

The authors have been actively involved in development and evaluation of testing and analysis techniques for some time [8,10,28,30,39,41]. In our research, we have seen an increasing need to implement a wide variety of testing and analysis techniques, so that the resulting tools could be used in experimental evaluation, as demonstrations of the techniques' feasibility and utility, and/or simply to gain insight into the practical ramifications of some of these techniques. Unfortunately, the implementation of testing and analysis tools tends to be a major undertaking, and the resulting tools are usually not flexible enough to be modified in the ways experimental results might indicate. Moreover, it is fairly clear that these tools must be integrated together. For example, data flow analysis tends to produce too many uninteresting anomalies unless it is integrated with a tool to evaluate path feasibility and subsequently remove unexecutable anomalies.

In examining several of the advanced testing and analysis techniques that we wished to study and implement, we noticed that they required only a few underlying capabilities, if we view these capabilities in a broader sense than has been done before. In particular, interpretation, data flow analysis, and reasoning facilities can be seen as basic fundamental operations required by a range of testing tools. For example, many advanced tools require some form of interpretation where the tool can monitor program state and/or exert control over the execution process, whether this interpretation entails conventional execution, symbolic execution, or monitoring data flow uses on paths. Work on a general data flow analyzer, which deals with arbitrary sequences of events instead of being restricted to data definitions and references is already being pursued [24] as are reasoning facilities that combine approaches and allow externally specified definitions of relationships [22,32].

Table 1 shows the reliance of many recently proposed testing systems upon these three basic analysis components. Clearly, this table does not present an exhaustive list of interesting testing techniques, and defining the techniques and demonstrating their use of the basic capabilities is outside the scope of this paper. It does serve to motivate how we arrived at our environment

Advanced Tools	Basic Components interpreter	data flow	reasoning
data flow coverage [9,20,23,27]	•		
data flow coverage with feasibility [15]	•		•
data flow analysis [14]		•	
data flow analysis with feasibility [26]	•	•	•
symbolic evaluation [10]	•		•
test data generation [10,35]	•	•	•
domain testing [8,35]	•		•
mutation analysis [6,12]	•		
partition analysis [28]	•		•
EQUATE [39]	•	•	•
RELAY [30]	•	•	•

Table 1: Advanced Tools and Their Required Basic Capabilities

architecture.

The basic components listed in table 1 are not readily available in most environments. In some instances they involve making visible information that is generally hidden (e.g., the structure of the internal code representation or of the execution state information). In other instances, they represent capabilities that are likely to be tailored to and enmeshed in the details of that particular advanced tool, instead of being isolated as an identifiable and callable unit. The effort required to reuse such code or, alternatively, to redevelop code for the same basic capabilities, makes the incremental process of developing new tools considerably more expensive. In TEAM, we provide these capabilities as part of the basic substrate upon which the advanced tools are based, thus facilitating the addition of more advanced tools to the environment.

3.2 Basic Analysis Components

As noted above, our basic analysis components provide capabilities for interpretation, data flow analysis, and reasoning. We will not further discuss here the data flow component, as we do not plan to implement a new data-flow analysis tool, but rather to import a system such as that described in [24], which is compatible with the basic principles of TEAM. In the remainder of this subsection, we explore our design for the interpretation and reasoning components.

3.2.1 The Interpretation Component

It may be far from apparent that interpretation is as common a requirement as is indicated in table 1. Nonetheless, careful study will show that each of the advanced tools listed there will be concerned either with monitoring program states that arise during execution or with exerting

control over the execution process. There are significant differences, however, in the underlying semantic models characterizing those executions.

Consider, for example, the following scenarios for the execution of the statement `A := B + C`, where `A`, `B`, and `C` are integer variables:

Actual Interpretation: One possibility is to request interpretation in a form that mimics, as closely as possible, the actions that would be taken by compiled code for that statement. This form of interpretation is the most familiar, and can serve as the basis for a wide variety of tools and activities that require more detailed monitoring/control of intermediate states of execution than are conventionally provided by compiled, machine-native code.

In this case, we might choose, for example, to represent integer variables as 32-bit binary numbers in 2's complement form. The semantics associated with '+' would be the familiar 2's complement addition, so that, if the initial values of `B` and `C` were the binary strings corresponding to 2 and 3, respectively, then following interpretation of this statement we would expect the value of `A` to be the binary representation of 5.

Symbolic Interpretation: Another option is to represent the values taken on by variables as algebraic expressions that denote the computational history of those variables. Symbolic interpretation has a variety of applications in software testing, verification, and analysis [10].

The semantics associated with '+' could then be to form a new algebraic expression with $+$ as its root operator and with the symbolic values of `B` and `C` as the operands of that $+$. Thus, if the initial values of `B` and `C` had been x and $2 * x + y$, respectively, then after interpretation the value of `A` might be $x + (2 * x + y)$. We can achieve different varieties of symbolic interpretation by altering these semantics somewhat. For example, we might choose to say that the semantics of '+' involve forming a new algebraic expression as above, but then simplifying that expression to yield, in this example, $3*x+y$ as the final value for `A`. Alteration of the semantics associated with conditional statements can yield the variants known as ***path-dependent symbolic evaluation*** [10] or ***global symbolic execution*** [7,10]. Symbolic and Actual interpretation can also be combined to yield ***dynamic symbolic evaluation*** [10].

Dynamic Data Flow Interpretation: An even less conventional form of interpretation can be achieved by letting the value of a variable or expression be a variable name or a null indicator. This form of interpretation can serve to monitor many of the data-flow based testing metrics surveyed in [9].

For this purpose, the semantics of '+' is to check each of its operands and, if that operand's value is a variable name, to mark that variable as having last been *referenced* at this statement. (Similarly, the semantics of '`:=`' would mark it's right-hand-side operand, if not null, as having been referenced, but to mark it's left-hand-side operand as having been *defined* at this statement.) Many variations of dynamic data flow interpretation can be realized by minor changes in these semantics, including monitoring of other testing metrics or, when combined with symbolic interpretation, the generation of program slices [34].

As diverse as the above examples of interpretation activities may seem, there are clearly common threads that suggest that an interpreter design to support all of them is possible. In each instance, the interpretation of `A := B + C` consisted of the following sequence of operations:

1. Fetch the current values of B and C.
2. Apply the + operator to those values.
3. Determine the location known as "A".
4. Apply the := operator to that location and to the value returned by the + operation.

What changed from one interpretation variant to another was:

1. the representation of *values* of variables and expressions, and
2. the semantics associated with the operators + and :=.

This suggests an interpreter design comprised of a common core interpretation algorithm that determines the order in which values are fetched and operators invoked, a variant part providing the representation of values, and variant parts defining the semantics of the language operators.

This separation of model-variant and model-invariant parts is a key idea in the development of the /Environment/ interpreter, called ARIES [42]. ARIES is designed to be

- Multi-Lingual: ARIES interprets programs encoded within the IRIS internal representation, which will be described later, thereby inheriting the substantial degree of language-independence afforded by that representation.
- Multi-Model: ARIES permits a tool-builder to specify *value kinds*, the data types that will be used to represent values, and *semantic functions*, the actions that will be bound to operator names such as '+', ':=', etc. Together, the choice of value kind and the binding of semantic functions to operators constitutes a description of a *computation model* [42].
- Generic: ARIES is not itself a standalone interpreter – rather it is a skeleton that is instantiated by the tool-builder to yield an interpreter tailored to the requirements of a particular tool. The computation model descriptions constitute the parameters controlling this instantiation.

3.2.2 The Reasoning Component

Testing and analysis tools frequently need to manipulate symbolic descriptions of various aspects of the code under test/analysis. A symbolic reasoning component is therefore a logical addition to the Basic Tools layer of the environment. Specifically, there appear to be three classes of symbolic problems that tools will wish to address:

Satisfiability: Given a system of constraints over some variables, does there exist an instance of those variables satisfying the constraints? Most approaches to automatic theorem proving address this question as their primary concern (universally quantified theorems are disproved by demonstrating the satisfiability of their negations).

Canonicalization: Given an expression, what is the desired canonical form of that expression? This is a more general problem than satisfiability, as the latter can be rephrased as asking whether a canonical form of an existentially qualified expression is "true". A common application of canonicalization is the reduction of a complicated symbolic expression to a "simplified" form. Another example of canonicalization is *loop analysis*, where a symbolic recurrence relation denoting the computation denoted by a loop is transformed into a closed form. For those programs where such transformations are possible, loop analysis can dramatically reduce the effort required for subsequent analysis and testing.

Constraint Solution: Given a system of constraints, what is an instance of the constrained variables satisfying the constraints? This "constructive" variant on satisfiability is a key problem in testing, since testing techniques frequently describe desired tests using constraint systems, in which case a constraint solution denotes an acceptable test.

We do not expect to provide a single technique to deal with all instances of these problems. Our basic component tool set will provide some special-case tools for symbolic reasoning (e.g., linear programming routines for determining satisfiability and constraint solutions of conjunctions of linear arithmetic relations) in addition to more general-purpose techniques. We also do not wish to discourage the later addition of new tools to this component. To achieve an appropriate level of support for the advanced testing tools, however, it is necessary that general-purpose techniques be provided as part of the basic analysis tools layer[1].

The selection of general-purpose techniques for our symbolic reasoning component poses some interesting problems. Given our emphasis upon facilitating the construction of new advanced tools, the design of these basic tools must meet some requirements not conventionally imposed upon similar symbolic reasoning systems in such contexts as program verification.

- Direct interaction of the reasoning tool with the user (e.g., prompting for suggested lemmas or other help) is usually unacceptable, as such interaction makes it impossible to hide the use of particular basic tools from the user of an advanced tool. Furthermore, such interaction may be of limited practical value, since in our environment the symbolic problems will usually be generated automatically in accordance with details of the internal processing of an advanced tool. It is unlikely that the user of the advanced tool would recognize or understand the problem statement and the consequences of solving or failing to solve that problem.

---

[1] An open question in the design of the symbolic reasoning component is the mechanism for selecting the correct tool. Our initial version leaves such selection to the judgment of the builder of advanced tools (e.g., someone building a domain testing tool would know that linear programming suffices for the constraint solution problems arising during the use of that testing technique). Automated aid for dynamically recognizing the opportunity for employing special-case tools will be investigated at a later time.

- Because the symbolic reasoning component will be employed as simply one operation in a more complex task, the computational resources devoted to it must be kept reasonable with respect to the rest of the advanced tool. In many applications of symbolic reasoning about software (e.g., program verification), we have a single theorem whose proof is of great interest to us. We are then willing to devote substantial effort to its solution. In TEAM, this is unlikely to be the case. The EQUATE testing technique, for example, may generate hundreds of theorems in the course of testing a single module. Most of these theorems will be trivially true or trivially false, and it is more important to quickly resolve the trivial 80 to 90% of the cases and return to the main tool than to struggle for long periods of time with the few truly difficult theorems. (This implies a design guideline for advanced tools – that they should provide a fall-back action in the event of failure to quickly resolve a symbolic problem. Such fall-backs would necessarily be less efficient than the actions taken upon successful resolution of the problem, but should allow the tool to continue to make progress.)
- Given the automatic generation of satisfiability problems, there is little reason to believe that the overwhelming majority of the generated constraint systems will be satisfiable. Nor is there reason to believe that the overwhelming majority of generated systems will be unsatisfiable. (Compare this to program verification, where the proposers of a theorem have a significant incentive to propose correct theorems.) Techniques that are *biased*, that tend to give quick confirmation of satisfiability but that perform exhaustive searches of some problem space before concluding unsatisfiability (e.g., resolution-based methods [31]), are therefore not acceptable as general-purpose components.

Note that we do not prohibit the use of reasoning techniques violating one or more of the above requirements in connection with *specific* advanced tools known to have special characteristics different from those presumed above. Our initial emphasis, however, has been to provide reasoning components that can be used with the widest possible variety of advanced tools, thereby gaining maximal leverage towards easing the burden of developing new advanced tools.

Currently we provide a term-reduction system to serve as the heart of our general-purpose symbolic reasoning component [18]. Term-reduction systems not only satisfy the requirements enumerated above, but appear to offer the most utility in the hands of novice specification writers. Furthermore, they meet our goal of genericity by serving as both satisfiability problem solvers and as canonicalization systems. In some instances they can also be employed for constraint solution (e.g., [5]).

3.3 Language Processing

This layer of TEAM is concerned with the conversion of program source code into an internal representation suitable for processing by the basic and advanced tools. Among the tools provided in this layer one would expect to find lexers, parsers, and translators. Given our concern for genericity, it should come as no surprise that we also include tools to aid in the production of new language processing tools, including analogues of the familiar LEX and YACC tools for generating lexers

and parsers. Various utilities and generics providing support for the generation of our internal representation are also provided.

The choice of internal representation for TEAM is one of the more interesting features of this layer, as it is the language-independence of our chosen representation that permits so many of the Basic and Advanced tools to be multi-lingual. We have chosen to employ IRIS [2,33], which stands for Internal Representation Including Semantics. IRIS is an abstract syntax graph that represents a program in terms of expressions. In essence, IRIS encodes everything as literals or as operators applied to a set of operand expressions. Thus the expression `2 + 3` is encoded as the application of an addition operator to the literals 2 and 3. A while-loop would be encoded as the application of a `while` operator applied to two operands, the first being an IRIS structure encoding a condition and the second an IRIS structure encoding a statement list.

A key feature of IRIS that separates it from other syntax graph representations (e.g., DIANA [17]) is that none of the operators in IRIS are predefined. Instead, the operator in each graph node is represented by a pointer to an IRIS structure representing the declaration of that operator, including such information as the operator's name, the number of operands it takes, the data types of those operands and of the returned value (if any), the in/out mode of the parameters, and whether each parameter is to be evaluated prior to invoking the operator's semantics (for example, '+' operators assume that their operands have been evaluated prior to performing the semantic action we call addition; the `while` operator does not expect its operands to have been previously evaluated but instead will, as part of its semantics, determine when and how often to evaluate them). There is essentially no difference between the declarations for the "predefined" operators for the programming language and for user-defined procedures and functions that have been compiled into IRIS. IRIS is therefore a general purpose structure for representing programs.

To represent a given language in IRIS, one must provide the IRIS-structured declarations of the syntactic operators for that language. Different sets of primitive operators would yield different languages. Basic and Advanced tools in our environment can often be designed to employ those operator declarations as encountered in the code being analyzed, rather than to presume a fixed set of primitive operators for some pre-chosen language. Tools designed in this manner will fully exploit much of IRIS's multi-lingual flexibility.

3.4 Environment Support

Tools in TEAM will create many complex objects, some of which will be interrelated in complex ways. Unfortunately, current file and data base systems do not provide adequate support for the creation, storing, retrieving, and sharing of such complex objects [4] and so we have had to design and implement a tool to meet our needs. This tool, called GRAPHITE, provides support for graph objects since graphs are the most common object defined by tools in the testing system and because many other common data structures can be treated as special cases of graphs. GRAPHITE also represents another example of a generic tool. Currently, GRAPHITE is the only tool in the Environment Support component although other tools may need to be added later.

GRAPHITE accepts specifications of classes of attributed graphs written in a graph description language, called GDL [11]. Given the GDL specification for a particular class of attributed graphs, GRAPHITE automatically produces the code for an abstract data type for that class of graphs.

Automatically creating the abstract data type from a graph description is not a new concept [19]. What is innovative about the GRAPHITE system is the design of the abstract data type that is actually generated and the ways in which it is used to meet our objectives. Central among those objectives is support for a prototyping approach to experimental tool development that permits a straightforward transition from prototype to production quality implementation. To this end, GRAPHITE produces two different kinds of abstract data type interfaces. One supports software development of experimental systems. It is designed so that when developers modify the definitions of graph classes there is a minimal effect on other tools in the system, even on those tools that use this modified class. The second interface is designed for efficient manipulation of graphs. When the definition of a graph class has been finalized, the second interface can be trivially substituted for the first so that a more efficient, but less flexible, version of the environment can be created. Thus, we have support for both development and production versions of an environment and a process for easily going from the development to the production version.

4 Conclusion

The TEAM environment has been under development for the past two years. It grew out of our interest to create an automated testing system for demonstrating the feasibility of many advanced testing techniques as well as to provide a platform to further our own research in software testing. Often empirical studies are required to evaluate a technique or to gain first hand experience with an approach. Unfortunately, many testing techniques are rather complex to implement, thereby thwarting such research, especially when investigating combinations of techniques and their effects. Moreover, we had previous experience with other testing systems and knew how hard it was to modify them in the numerous ways that inevitably emerged from our own research endeavors. Thus, we were and continue to be very motivated to develop a system that supports integration, extensibility, and experimentation.

The current version of the TEAM environment has been instantiated to accept a substantial subset of Ada as well as an Ada-like design language [36]. Although these languages are similar, they are different enough that we have successfully experimented with creating multi-language as well as multi-phase testing tools.

We chose Ada as the first language to analyze since it supports data abstraction and provides mechanisms for concurrency. Both abstraction and concurrency pose interesting problems for testing that we hope to address in our research [21,39]. Although it is relatively straightforward to instantiate a new language, it does require considerable effort and we have only limited resources. Thus, we feel we must initially choose our analysis languages carefully.

The environment is totally written in Ada to further portability and to allow us eventually to apply our own testing and analysis tools to themselves as well as to the other tools in the environment. TEAM currently runs on the SUN and DEC/Ultrix Verdix compilers as well as on

the DEC/VMS Ada compiler. We are currently in the process of porting the system to an ALSYS compiler. We have had considerable compiler difficulties; current Ada compilers are not known for their robust support for generics, a feature we use extensively.

Currently two advanced testing and analysis tools have been implemented, a symbolic evaluation tool and a simple debugger. In addition several other tools are under development, including a tool to do inter-module analysis [36] and tools to do concurrency analysis [1,21]. These efforts have demonstrated the relative ease with which tools can be composed from the basic testing and analysis tools and underlying components. The implemented tools use different computation models in the interpreter and make very different demands on the reasoning facility. They demonstrate the ease with which tools can exploit genericity to tailor the basic components to their particulr needs. These two tools are also designed to fully exploit the language independent representation provided by IRIS and, thus, can accept either of the languages currently supported by the language processing component. Most of our tools make extensive use of the GRAPHITE system, since graphs are a pervasive data structure in program analysis. To date we have only implemented the development version of this component. The prototyping support that it provides has been greatly appreciated by our programmers, who use it to avoid long compilations as they incrementally develop their software.

TEAM is only an initial prototype. There must be considerably more experience before stronger conclusions can be drawn about its effectiveness and weaknesses. In particular, more advanced and diverse testing and analysis components must be implemented to assess how well it meets our original goals. Many of the components are only first approximations to the eventual implementations that should be provided. In particular, much more powerful reasoning capabilities should be added to the system. This is an open ended area in that there will always be a desire for more and more powerful reasoning tools. The question is, how easily does our architecture allow us to incorporate and employ these new capabilities? We see this as a weakness in our current approach, since there is currently no mechanism for defining how tools interact and the objects they create and use other than through the code that implements the tools. Issues such as this are currently being addressed by the authors and their colleagues as part of the Arcadia project. In the Arcadia environment, process programs [25] will be used to explicitly define the relationship among the various tools and the objects they create and use, and support for defining and maintaining these relationships will be provided as a part of the environment infrastructure.

Another weakness in the TEAM environment is its support for object management. The GRAPHITE system supports graph objects but no other type of objects. Moreover, it only supports limited kinds of inter-graph relationships. Although it provides only limited functionality, this component has proven invaluable in our software development efforts as well as in helping to clarify some of the complex object management issues that need to be explored further.

Eventually, many of the components of the TEAM environment will be moved over to the Arcadia environment. The interaction among the testing and analysis tools will provide some of the first examples of process programs and their support for complex tool interaction. The environment support component of TEAM will be completely replaced by Arcadia's more powerful object management system. The language processing component described here was built as part

of the Arcadia environment and was developed jointly by consortium members. The Arcadia user interface [38] will provide a uniform way of interacting with the tool set. We believe the basic design of the advanced testing tools and the basic testing and analysis components, however, will remain relatively unchanged. Moreover, we believed that once they are supported by a more powerful environment infrastructure, we will be able to more widely experiment with how well they facilitate integration, extensibility and experimentation of testing and analysis tools.

REFERENCES

[1] G. S. Avrunin and J. C. Wileden. Toward automating analysis support for developers of distributed software. to appear in *Proceedings of the International Conference on Distributed Computing Systems*, June 1988.

[2] D. A. Baker, D. A. Fisher, and J. C. Shultis. *IRIS: An Internal Form for Use in Integrated Environments.* Technical Report, Incremental Systems Corporation, 1987.

[3] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278-1296, Dec. 1987.

[4] P. A. Bernstein. Database system support for software engineering. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 166-178, Monterey, CA, March 1987.

[5] L. Bouge, N. Choquet, L. Fribourg, and M. C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343-360, Nov. 1986.

[6] T. A. Budd. Mutation analysis: ideas, examples, problems and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129-148, North-Holland, 1981.

[7] T. E. Cheatham Jr., G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, SE-5(4):402-417, July 1979.

[8] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at Domain Testing. *IEEE Transactions on Software Engineering*, SE-8(4):380-390, July 1982.

[9] L. A. Clarke, A. Podgursky, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244-251, London, England, Aug. 1985.

[10] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, Jan. 1985.

[11] L. A. Clarke, J. C. Wileden, and A. L. Wolf. GRAPHITE: a meta-tool for Ada environment development. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments, IEEE Computer Society Press*, pages 81–90, Miami Beach, Florida, April 1986.

[12] R. A. DeMillo, F. G. Sayward, and R. J. Lipton. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34–41, Apr. 1978.

[13] L. K. Dillon. Symbolic execution of ada tasking. In *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, IEEE, May 1988.

[14] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8(3):305–330, Sep. 1976.

[15] P. G. Frankl and E. J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Proceedings of the ACM SIGSOFT/IEEE Workshop on Software Testing*, pages 4–13, IEEE, July 1986.

[16] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[17] G. Goos, W. A. Wulf, A. Evans Jr., and K. J. Butler. *Diana: An Intermediate Language for Ada*. Springer-Verlag, 1983.

[18] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, Dec. 1978.

[19] D. A. Lamb. *Sharing Intermediate Representations: The Interface Description Language*. Technical Report CMU-CS-83-129, Carnegie-Mellon University, Pittsburgh, PA, 1983.

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

[21] D. L. Long and L. A. Clarke. *Task Interaction Graphs for Concurrency Analysis*. in preparation, University of Massachusetts, 1988.

[22] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), Oct. 1979.

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

[24] K. M. Olender and L. J. Osterweil. *Specification and Static Evaluation of Sequencing Constraints in Software*. Technical Report CU-CS-335-86, University of Colorado, Boulder, CO, June 1986.

[25] L. Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, CA, March 1987.

[26] L. J. Osterweil. *The Detection of Unexecutable Program Paths Through Static Data Flow Analysis*. Technical Report CU-CS-110-87, University of Colorado, Boulder, CO, May 1987.

[27] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Apr. 1985.

[28] D. J. Richardson and L. A. Clarke. Partition analysis: a method combining testing and verification. *IEEE Transactions on Software Engineering*, SE-11(12):1477–1490, Dec. 1985.

[29] D. J. Richardson and M. C. Thompson. *An Analysis of Test Data Selection Criteria Using the RELAY Model of Error Detection*. Technical Report 86-65, Computer and Information Science, University of Massachusetts, Amherst, Dec. 1986.

[30] D. J. Richardson and M. C. Thompson. The relay model of error detection and its application. to appear in *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, July 1988.

[31] J. Robinson. A machine-oriented logic based on the resolution principle. *International Journal of Computer Mathematics*, 1:227–234, 1965.

[32] R. E. Shostak. Deciding combinations of theories. In D. W. Loveland, editor, *Proceedings of the Sixth Conference on Automated Deduction*, pages 209–222, Springer-Verlag, New York, 1982.

[33] S. D. Sykes. *IRIS Preliminary Design and Rationale*. Arcadia document, Aerospace Corporation, Sep. 1985.

[34] M. D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[35] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, May 1980.

[36] A. L. Wolf, L. A. Clarke, and J. Wileden. The AdaPIC toolset: supporting interface control and analysis throughout the software development process. to appear in *IEEE Transactions on Software Engineering*, 1988.

[37] M. Young and R. N. Taylor. Combining static concurrency analysis and symbolic execution. *to appear in IEEE Transactions on Software Engineering*, 1988.

[38] M. Young, R. N. Taylor, D. B. Troup, and C. D. Kelly. Design principles behind Chiron: A UIMS for software environments. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 367–376, IEEE, Singapore, Apr. 1988.

[39] S. J. Zeil. Complexity of the equate testing strategy. *Journal of Systems and Software*, 8(2):91–104, March 1988.

[40] S. J. Zeil. Selectivity of data-flow and control-flow criteria. to appear in *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, July 1988.

[41] S. J. Zeil. Testing for perturbations of program statements. *IEEE Transactions on Software Engineering*, SE-9(3):335–346, May 1983.

[42] S. J. Zeil and E. C. Epp. Interpretation in a tool-fragment environment. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 241–248, IEEE, Apr. 1988.