URC97081

A Categorization of Dynamic Analyzers

Michelle R. Lujan*
Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968
email mlujan@cs.utep.edu

1 Introduction

Program analysis techniques and tools are essential to the development process because of the support they provide in detecting errors and deficiencies at different phases of development. The types of information rendered through analysis includes the following: statistical measurements of code, type checks, dataflow analysis, consistency checks, test data, verification of code, and debugging information. Analyzers can be broken into two major categories: dynamic and static. Static analyzers examine programs with respect to syntax errors and structural properties [17], This includes gathering statistical information on program content, such as the number of lines of executable code, source lines. and cyclomatic complexity. In addition, static analyzers provide the ability to check for the consistency of programs with respect to variables. Dynamic analyzers in contrast are dependent on input and the execution of a program providing the ability to find errors that cannot be detected through the use of static analysis alone. Dynamic analysis provides information on the behavior of' a program rather than on the syntax [1 7]. Both types of analysis detect errors in a program, but dynamic analyzers accomplish this through run-time behavior.

This paper focuses on the following broad classification of dynamic analyzers:

- Metrics
- Models
- Monitors

Metrics are those anal yzers that provide measurement. The next category, models, captures those analyzers that present the state of the program to the user at specified points in time. The last category, monitors, checks specified code based on some criteria. The paper discusses each classification and the techniques that are included under them. in addition, the role of each technique in the software life cycle is discussed. Familiarization with the tools that measure, model and monitor programs provides a framework for understanding the program's dynamic behavior from different, perspectives through analysis of the input/output data.

2 Metrics

The term *metrics* is defined as the measure of properties of systems [14]. Metrics are used to measure the quality of a program, control productivity of software projects, and attempt, to predict the effort in developing software [2]. In addition, errors can be detected in programs through the use of statistical information. According to [14], metrics should have the following properties to be useful:

1. Automatable: the metrics must be processed automatically.
2. Feasible: the cost of gathering metrics must be feasible in cost and time in order to be useful.
3. Understandable: the metrics must have theoretical and empirical foundations and clear meaning.

* This work was sponsored by NASA under contract NAG-1012 and NCCW-0089

4. Sensitive: the metrics should be sensitive to all the factors that affect the quality to be estimated and not those that, are unrelated.

5. Applicable: the metrics should be applicable to any programming language and to a large class of systems, and not to any particular stage of the lifecycle.

6. Useful: metrics should not measure values but should provide feedback concerning software activities.

7. Flexible: a variety of metrics is needed to provide users with a broad picture of the properties of the system.

W bile static metrics generate information about variable anomalies and program content, dynamic metrics produce data on program interconnections and variable behaviors [9]. one category of dynamic metrics, coverage analyzers, records information concerning logic, statements, cent rol paths, decision, conditions, dataflow and iteration. A few are discussed next. ***Control path coverage analyzers*** record the control paths covered by each test case [5]. *Statement coverage* ***analyzers*** check that every statement in the program is executed by the test data set at least once. *Decision* ***coverage analyzers*** verifies that each predicate decision assumes a ***true*** and a ***false*** outcome at least once during testing on a test suite. Clearly, coverage measures are categorized as dynamic analyzers because they require execution of the program in order to collect the metrics. Even though the general notion of coverage provides a simple methodology for developing test suites, many errors may still escape detection [17].

Reliability measures, another category of dynamic metrics, generate a probability measure that a software fault does not occur during a specified time interval [1 4]. These measures provide information concerning the number of failures during a specified amount of runs and the totality of failures within a specified interval.

Performance measures produce a statistic on the overall performance of a program. Measures of this type include estimating the stability and reliability of a program [9] [14]. An example of a performance measure that approximates the stability of a program is Yau and Collofello's Logical Stability Metric [9]. This calculation is used to determine the expected impact of a modification of a variable in a module. The stability metric reflects the result of changing a single variable in a module and the impact of the change on the behavior of the program. The authors' basic argument is that maintenance breaks down to changes in variables regardless of how complex the task.

3 Models

Observing the state of a program throughout different points of execution time provides valuable information to the person analyzing a program. *Models* provide a systematic method to accomplish this end. By supplying the users with the ability to test, hypothesis and draw conclusions about program behavior, models ***support*** the evolution of programs over time [3].

Interpreters, a category of models, execute and translate programs at the same time. This is an advantage because many interpreters show the likely place and cause of errors [13]. Interpreters are made up of four basic components [16]:

1. an engine to interpret the program,

2. memory that contains the pseudocode to be interpreted,

3. representation of the control state of the interpretation engine, and

4. a representation of the current state of the program being simulated.

As a result, the state of the machine can be observed through the execution of the program with the provided input. Because interpreters depend on execution and input, they are categorized as dynamic analyzers.

Prototypes, the second category of models, provide a mechanism to learn more about the problem and the problem solution through a partial implementation of the system [4]. Prototypes enable potential users to experiment with the system relatively early in the development process. Essentially, this allows users to provide feedback to designers on whether the behavior of the system is as expected and to determine user needs [4]. Prototyping is often viewed as a way of progressively developing an application and at the same time understanding the requirements [8]. The cost, of the system is reduced because prototypes can be produced early in the development process without implementing the entire system. Prototypes are viewed as the first version of the system. Two types of prototypes are throwaway and evolutionary. Throwaway

prototypes are those that are discarded after it is used. Evolutionary prototypes are those that continually changed over time until it behaves as expected (becoming the final producet)[4]. The evolutionary prototype is more cost effective because it is not destroyed after it is used [4]. A prototyping environment is provided through Computer-Aided Prototyping (CAPS) [i 1], being developed at the U.S. Naval Postgraduate School. CAPS is comprised of the following tools that aid the development of a prototype: a graph data model, change merging facility, automatic generators for schedule and control code and automated retrievals for reusable components.

Debuggers aid in locating, analyzing, and correcting errors by providing the user with the ability to examine a program by executing code one line at a time. These tools allow the users to inspect the execution of a program in detail, to control the time that each instruction takes to execute, and to control the progress of the computation [3]. The history of execution can be generated along with the state of variables and the machine. This trace facilitates the collection and manipulation of information [3].

A few examples of debuggers are YODA [12], TSL [12], and EBBA Event Based Behavioral Abstraction) [1]. The YODA system stores Ada event histories as Prolog facts. Predicates in Prolog define the common temporal relationships. This system has the capability to specify when variables can be updated, what, values variables can take on when updated and the communication between variables. The TSL system automatically checks specifications against the events produced by an Ada tasking program. In addition, it uses Ada semantics to ensure that pairs of events appear in the correct order in the event history. These debuggers provide a method to detect errors in concurrent Ada programs [1 2].

The debugging tool EBBA is also based on event histories. A distinguishing feature of this approach is its ability to model system behavior through clustering and filtering. Clustering expresses behavior as composite events whereas filtering removes from consideration those events that are not needed for the behaviors being investigated [1].

4 Monitors

Programs often need to meet certain criteria in order to provide the desired functionality. *Monitors* provide the ability to *examine code* against criteria imposed by the user or the designer to check for satisfiability. The aim is to monitor the reliability and quality of software systems.

The first, category of monitors, assertion checkers, are those tools that support automatic runtime detecion of software faults during debugging, testing and maintenance [15]. Through the use of assertion checkers, developers are provided with the capability to incorporate assertions in programs in order to ensure that they are not violated throughout execution. Assertions are defined as formal specifications that describe the properties of programs using mathematical notation. In other words, assertions specify what a system is supposed to do instead of how to implement it. Assertion checkers verify that assertions are maintained throughout, runtime. A few tools that fall under the category of assertion checkers follow.

The annotation language ANNA (A NNotated Ada) is used to embed assertions into Ada programs and performs consistency tests to determine if the computation satisfies the specified properties, ANNA has the ability to ensure that assertions are maintained throughout the execution of a program. Features of A NNA include the following [10]:

- generates consistency checks from annotations on types, variables, subprograms, and exceptions,
- . uses incremental theorem proving to check algebraic specifications at runtime. and
- constructs large software systems based on algebraic specification of system models.

Based on ANNA, the Annotation Preprocessor (APP) for C programs is a replacement for the standard preprocessing pass of the C compiler. In addition, APP provides a mechanism to define how assertion violations will be handled during execution and the level of checking that is to be done [15]. An assertion in APP specifies a constraint that is related to some state of computation. Constraints are specified using C's expression language. APP converts each assertion into a runtime check in order to test for violations of constraints. In this way, APP provides a convenient method to specify and maintain assertions.

FORMAN (FORmal Annotation), an assertion language, has the capability to express assertions on events and sequences of operations and events [6]. Included in FORMAN, is the ability to describe universal assertions on the program. Assertions can be collected into ii braries to increase the level of automation to encounter errors. FORMAN includes a flexible language for trace specification based on event patterns

and regular expression [1]. In addition, FORMAN has the capability to express both general operational assertions and declarative assertions.

Another language, Behavioral Expressions (BE), provides the capability to write assertions about sequences of process interactions. It also has the functionality to describe allowed sequences of events as well as some predicates [1]. Events are used to describe process communication, termination, connection, and detachment of process to channels. BE performs evaluations of assertions at runtime.

Context monitoring [7] is an approach that provides the developer with tools to manage, and communicate across personnel, application domain knowledge about the properties on and relationships between objects being modeled by a software system. Knowledge about the data, the intended context in which programs will run and other knowledge about the program is captured through integrity constraints. The constraints are elicited from domain experts, customers, analysts, designers and programmers using established methods. The constraint satisfiability mechanism dynamical ly monitors a program to ensure that the constraints are being enforced by the program. If a violation occurs, the user is notified and, because links exist between the constraints and the documents that support the constraint, the user can identify the source of the constraint,. This approach is distinguished from the others because t he constraints are not embedded in the program code, but are maintained in a repository.

5 Summary

Classification	*Techniques*	*Principal Life Cycle Support*
Metrics	Coverage measures	Testing
	Reliability measures	Implementation
		Maintenance
	Performance measures	Implementation
		Maintenance
Models	Interpreters	Implementation
		Testing
	Prototypes	Requirements
		Design
	Debuggers	Implementation
		Maintenance
Monitors	Assertion checkers	Requirements
		Implementation/Maintenance
		Testing
	Context monitoring	Requirements
		Design
		Implementation/Maintenance
		Testing

Figure 1: A Classification of Dynamic Analyzers.

Examining runtime behavior is an important step in error detection and analysis. Data gathered from runtime behavior can provide insight into errors which may not be detected through static analysis. Metrics, models and monitors all produce different types of dynamic information about programs and, depending cm the technique, support different aspects of the software life cycle (see Fig. 1). Metrics generate statistical information about variables and program interconnections. Models monitor the state of the machine at specified times during program execution. Monitors oversee that criteria specified by designers or users are not violated, Even though dynamic analyzers alone do not supply enough data about programs to localize all errors, they do furnish information that static analyzers do not.

References

[1] Auguston, M., " A Language for Debugging Automation", in *Proceedings of SEKE*, U.S.A.: Knowledge Systems Institute, 1994, pp.108-115.

[2] Basili, V. R., Selby, R. W., Yun, T., " Metric Analysis and Data Validation Across Fortran Projects", *IEEE Transactions Software Eng.* **SE-9 (6),** *6.52-663 (*1983).

[3] Brindle, A. F., Taylor, R. N., Martin, D. F., "A Debugger for Ada Tasking". *IEEE Transactions Software Eng.* **SE-15 (3), 293-304 (1989).**

[4] Davis, A. M., "Soft ware Prototyping", in Yovits, M. C., Zelkowitz, M. V. (eds.), *Advances in Computers Vol. 40. San* Diego: Academic Press, 1995, pp. 39-63.

[5] Fairley, R., ***Software Engineering Concepts. New*** York: McGraw-Hill Publishing Company) 1985

[6] Fritzon, P., Auguston, M., Shahmehri, N., "Using Assertions in Declarative and Operational Models for Automated Debugging", *J. Systems Software 25, 223-239 (1994).*

[7] Gates, A. Q., F. G. Fernandez and L. Romo, "Building Systems with integrity Constraints," to appear in ***The Proceedings of the Second World Conference on Integrated Design and Process Technology,*** December 1-4, 1996, Austin, Texas.

[8] Ghezzi, C., J azayeri, M., Mandrioli, D., *Fundamentals of Software Engineering.* New Jersey: Prentice Hall, 1991.

[9] Kafura, D., Reddy, G. R., " The Use of Software Complexity Metrics in Software Maintenance", ***IEEE Transactions Software Eng*** **SE-13** *(3), 335-343 (1987).*

[10] Luckham, D. and Von Henke, I?. W., "An Overview of Anna: A Specification Language for Ada," *IEEE Software,* **20**(2):9-23, 1985.

[11] Luqi, Goguen, J. and V. Berzins, "Formal Support for Software Evolution," *1994 Monterey Workshop Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Soft ware Evolution.* Monterey, CA: U.S. Naval Postgraduate School, Sept. *7-9,* 1994, pp. 10-21.

[12] McDowell, C. E., Helmbold, D. P., " Debugging Concurrent, Programs", *ACM Computing Surveys* **21** *(4), 593-622* (1989).

[13] Pfaffenberger, B. *Que's Computer User's Dictionary.* 4th ed. U. S. A.: Que, 1993

[14] Ramamoorthy, C. V., Prakash, A., Garg, V.. Yamura, T., Bhide, A., " Issues in the Development, of Large, Distributed, and Reliable Software", in Yovits, M. C. (ed. *), Advances in Computer Vol.* .26. San Diego: Academic Press, 1987, pp. 393-443.

[15] Rosenblum, D. S., " A Practical Approach to Programming With Assertions", *IEEE Transactions Software Eng.* **21** (l), 19-31 (1995).

[16] Shaw, M., Garlan, D., ***Software Architecture: Perspectives on an Emerging Discipline. New*** Jersey: Prentice Hall, 1996.

[17] White, L. J., "Software Testing and Verification", in Yovits, M. C. (cd.), ***Advances in Computer Vol.*** 26. San Diego: Academic Press, 1987, pp. 335-391.