AUTOMATING INSTRUMENTATION: IDENTIFYING INSTRUMENTATION POINTS FOR MONITORING CONSTRAINTS AT RUNTIME

Mary F. Payne

Computer Science Department

APPROVED:

Ann Q. Gates, Ph.D., Chair

Steven M. Roach, Ph.D.

Scott Starks, Ph.D.

Charles H. Ambler, Ph.D.
Dean of the Graduate School

To Dr. Gates

AUTOMATING INSTRUMENTATION: IDENTIFYING INSTRUMENTATION POINTS FOR MONITORING CONSTRAINTS AT RUNTIME

by

Mary F. Payne, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science Department

THE UNIVERSITY OF TEXAS AT EL PASO

December 2003

ProQuest Information and Learning Company
300 North Zeeb Road
PO Box 1346
Ann Arbor, MI 48106-1346

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor and committee members Dr. Ann Gates, Dr. Steve Roach, and Dr. Scott Starks for their time, guidance, and also patience. I would especially like to thank Dr. Gates for giving me the opportunity to work as a research assistant and for encouraging me to pursue a graduate degree. I acknowledge my family for their unwavering support and for laying the foundation for my accomplishments. In addition, I thank my friends for their encouragement and for helping me to maintain my sanity.

Finally, I would like to thank the National Aeronautic Space Agency (NASA) and the National Science Foundation (NSF) for their financial support under project numbers NCC5-205 and 26-1005-2, respectively, during my tenure as a research assistant.

ABSTRACT

Code instrumentation is the insertion of code, at the source code level or a lower code level, into specific locations of a program in order to verify, measure, or collect certain aspects of program behavior. The inserted code must not alter the behavior of the original program. Code instrumentation is a tool with various applications, such as debugging and performance analysis. Code instrumentation is also used by runtime monitors to ensure that programs are executing correctly with respect to specified properties. There exist programming languages that support runtime assertion checking, such as Eiffel and Anna. However, they require that the constraint checking code be inserted manually by the programmer. Many existing runtime monitors also require that the instrumentation be performed manually. Manual instrumentation is not only tedious, but can introduce many problems, such as human error and difficulty maintaining the software system. These drawbacks can be rectified by automating the instrumentation process. This work explores some of the issues automatic instrumentation faces and, using the DynaMICs runtime monitor framework, provides algorithms to statically locate the points in a program where behavior checking code should be inserted.

Table of Contents

LIST OF FIGURES

Chapter 1
INTRODUCTION

Code instrumentation is the insertion of code, at the source code level or a lower code level, into specific locations of a program in order to verify, measure, or collect certain aspects of program behavior. The inserted code must not alter the behavior of the original program. Code instrumentation is a tool with various applications. For instance, instrumentation is frequently used for debugging programs. It is common practice among programmers to insert code into their programs for debugging purposes. More sophisticated code instrumentation is used by profilers to analyze the performance of software and architecture implementations. Code instrumentation is also used for the runtime monitoring of programs. The focus of this thesis is on instrumentation to support runtime software-fault monitoring (henceforth referred to as runtime monitoring).

Runtime monitoring is "a dynamic technique that provides assurance that specified properties hold during program execution" [6]. Because traditional verification approaches such as testing and inspections are not sufficient to ensure the correct behavior of software, especially in the presence of subtle faults, runtime monitoring has become a valuable, if not necessary, tool for checking that safety critical systems behave in a predictable manner. Runtime monitoring approaches address reliability, safe failure, and avoidance of hazardous states with respect to software systems. Examples of runtime monitoring systems that provide some support for instrumentation are Anna and the Annotation PreProcessor (APP). These methods have drawbacks that have discouraged

their wide-spread use. For instance, the use of these methods focuses on assertions defined at the design and implementation phases [9]. As a result, the justification for the assertions is not always traceable to the specification documents. The most significant drawback, however, is that not all types of assertions can be inserted automatically requiring programmers to insert checking code manually in a system's code. The programmer, therefore, must have knowledge of the design and implementation of the system [2]. The programmer may introduce new errors by manually inserting code. Another drawback is that some of the assertions may become unnecessary and new assertions may become necessary during the system's maintenance. The original assertions would have to be located and removed from the system.

Other monitoring methods, such as DynaMICs and MaC, strive to overcome the drawbacks given above by automatically instrumenting programs with assertion-checking code. The *purpose of the thesis* is to present a methodology and algorithms to statically and automatically locate the necessary instrumentation points in a system where monitoring code should be inserted. This work could then be used in runtime monitors to automate the instrumentation process. The *significance of the thesis* is that automatic instrumentation reduces the programmer's role in inserting code, thus reducing human-introduced errors. Also, the constraint definitions and the points of instrumentation can be kept separately; simplifying the maintenance of software systems. This work is presented in the following manner. Chapter 2 discusses the issues related to instrumentation and methods of instrumentation, including MaC and Java PathExplorer. Chapter 3 presents the DynaMICs framework that is used for the algorithms. Chapter 4

introduces the algorithms and examples of their use. Chapter 5 discusses the results, unresolved issues, and future work.

Chapter 2
BACKGROUND

This chapter discusses instrumentation in general and automation issues. Special attention is given to aliases and alias analysis algorithms.

2.1 Instrumentation Approaches

Runtime monitors are especially important to safety-critical software systems and to systems whose failures can cause financial loss. The extra security provided by monitors makes up for the incompleteness of testing and current formal verification methods. It is impossible to conduct exhaustive testing to ensure the correctness of a system's implementation. Formal verification methods are not yet able to verify large systems. Also, formal verification methods are used for verifying specifications of a system and not the actual implementation.

The following sections provide brief overviews runtime monitoring tools and approaches that use manual and automated instrumentation.

2.1.1 Manual Instrumentation

Most instrumentation is performed manually. That is, the programmer must know what aspects of a system need to be checked for correctness, locate the points in the system where those aspects are affected, and insert code fragments to check that the system is behaving correctly with respect to those aspects. Assertion checkers are tools that support manual instrumentation. Assertions are a subclass of constraints in that they do not capture implementation independent knowledge regarding the system [8]. Some

tools and languages that support assertion checking are Anna, Eiffel, the Annotation PreProcessor for C (APP), and AspectC++. Anna is used for assertion checking in Ada programs. This approach uses Boolean expressions to express assertions on objects, statements, and modules. Anna also has the ability to compute values that are not actually computed by the system but are needed by assertions. APP is used for monitoring C programs through pre- and post-conditions. Eiffel, an object-oriented language, supports in-lined assertions. AspectC++ is an extension of the C/C++ programming languages. These assertions are inserted directly into the system [9]. A full discussion of monitoring systems can be found in Delgado's survey [6].

As previously mentioned, the aforementioned approaches do not provide full support for automated instrumentation; thus, the programmer must locate the points where instrumentation should take place in order to use those tools. As a result, errors may be introduced inadvertently. There is a risk that the programmer may not insert code to check assertions at every location where it is required, especially if the system is complex and the programmer does not have sufficient knowledge of the system's design and specification. Also, assertion checks inserted by a programmer may not be traceable to design and specification documents to justify their applicability. These monitoring systems show their vulnerability during the maintenance phase, as well, when the requirements and program are modified and the constraints may need modification. Much time and effort must be spent in order to find and change these constraints.

2.1.2 Automated Instrumentation

This section briefly discusses the approaches used by runtime monitors that automate the instrumentation process. The two runtime monitors discussed are MaC and the Java PathExplorer. A third runtime monitor, DynaMICs, is discussed in detail in Chapter 3. Also briefly discussed are dynamic instrumentation approaches.

The first tool's framework that is examined is the Monitoring and Checking runtime monitoring framework (MaC) [19]. MaC has two main phases of operation. The first phase concerns actions taken before the system's execution. The second phase concerns the runtime actions performed by MaC. Of major concern to this work is the first phase. First, system requirements are expressed in the Meta Event Definition Language (MEDL). A *monitoring script* is created using the Primitive Event Definition Language (PEDL). This script maps the events described by MEDL to the low-level information extracted during runtime. A *filter* and *event recognizer* is generated from the script. The filter generates code fragments that are inserted into the system to check specified objects. Instrumentation is performed statically. The event recognizer recognizes events from the information received by the filter. Events are sent to a runtime checker where their values are checked with respect to the requirements.

The Java PathExplorer (JPaX) framework also creates a script for specifying instrumentation. JPaX uses two languages for formalizing high-level specifications [6]. One is JPaX's own linear temporal logic. The other language used is the algebraic specification language Maude. This specification script includes an instrumentation script and a verification script. The instrumentation script describes how and where the

system is instrumented. The verification script defines what properties need to be checked.

The focus of this work is on performing instrumentation statically. However, research is being done on dynamic instrumentation. The Wyong system [27] and the Metric Description Language (MDL) compiler [16] are two instances of this research. MDL specifies what information needs to be collected and how. It also specifies code to be inserted into a system. This system is mainly designed to collect performance data. Because of complexity concerns, information can only gathered at the entrance and exit of procedure calls. The Wyong system generates dynamic analysis tools from specifications formalized by using attribute grammars. These attribute grammars are based on machine-level concepts and are not dependent on any particular source code language. Wyong is built upon Eli and ATOM.

2.2 Automation Issues

The ultimate goal for the users of code instrumentation is to automate the instrumentation process. In order to do so, it is necessary for whatever tool performing the instrumentation to have some basic "understanding" of the system being instrumented. The instrumentation tool requires information regarding the flow of execution of the system (control-flow), as well as information regarding how data is used and modified in the system, in order to locate those points in a program where a system should be instrumented. This knowledge is acquired through control-flow analysis and data-flow analysis. Traditionally, control-flow and data-flow analyses have been used by compilers to perform effective and correct optimizations on a program's code. However,

the information required by an instrumentation tool is very similar to an optimizing compiler's required information. Control-flow analysis is discussed first because "any static, global analysis of the expression and data relationships in a program requires a knowledge of the control flow of the program" [1]. Data-flow analysis is then discussed. Interprocedural analysis is also introduced. Finally, alias analysis, a major issue facing automated instrumentation, is described.

2.2.1 Control Flow Analysis

Control-flow analysis of a routine is characterized by control-flow graphs, or flowgraphs. These are rooted, directed graphs that describe the flow of execution for the routine. The flowgraph of a routine is composed of a set of nodes N and a set of edges $E \subseteq N \times N$, often referred to as $G = (N, E)$ [24].

There are two main approaches to performing control-flow analysis on a single routine. Each approach begins by identifying basic blocks and creating a control-flow graph, but differ in the program constructs that they identify. The first and simplest approach to control-flow analysis is to use dominators to identify loop constructs. The second approach is known as interval analysis. Interval analysis "includes a series of methods that analyze the overall structure of the routine and that decompose it into nested regions called intervals" [24]. Both approaches to control-flow analysis are discussed in this section.

To construct the control-flow graphs, each approach must first identify the basic blocks comprising a routine. A *basic block* is a maximal sequence of instructions such that control can only be transferred to its first instruction and the block can only be exited