Winter 2000

Extending Traditional Static Analysis Techniques to Support Development, Testing and Maintenance of Component-Based Solutions

Robert David Cherinka
Old Dominion University

EXTENDING TRADITIONAL STATIC ANALYSIS TECHNIQUES TO SUPPORT DEVELOPMENT, TESTING AND MAINTENANCE OF COMPONENT-BASED SOLUTIONS

by

Robert David Cherinka
B.S. May 1987, University of Pittsburgh
M.S. December 1991, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2000

Approved by:

C. M. Overstreet (Director)

C. J. Wild (Member)

K. J. Maly (Member)

S. Zeil (Member)

M. A. Malloy (Member)

ABSTRACT

EXTENDING TRADITIONAL STATIC ANALYSIS TECHNIQUES TO SUPPORT DEVELOPMENT, TESTING AND MAINTENANCE OF COMPONENT-BASED SOLUTIONS.

Robert D. Cherinka
Old Dominion University, 2000
Director: Dr. C. M. Overstreet

Traditional static code analysis encompasses a mature set of techniques for helping understand and optimize programs, such as dead code elimination, program slicing, and partial evaluation (code specialization). It is well understood that compared to other program analysis techniques (e.g., dynamic analysis), static analysis techniques do a reasonable job for the cost associated with implementing them. Industry and government are moving away from more 'traditional' development approaches towards component-based approaches as 'the norm.' Component-based applications most often comprise a collection of distributed object-oriented components such as forms, code snippets, reports, modules, databases, objects, containers, and the like. These components are glued together by code typically written in a visual language. Some industrial experience shows that component-based development and the subsequent use of visual development environments, while reducing an application's total development time, actually increase certain maintenance problems. This provides a motivation for using automated analysis techniques on such systems. The results of this research show that traditional static analysis techniques may not be sufficient for analyzing component-based systems. We examine closely the characteristics of a component-based system and document many of the issues that we feel make the development, analysis, testing and maintenance of such systems more difficult. By analyzing additional summary information for the components as well as any available source code for an application, we show ways in which traditional static analysis techniques may be augmented, thereby increasing the accuracy of static analysis results and ultimately making the maintenance of component-based systems a manageable task. We develop a technique to use semantic

information about component properties obtained from type library and interface definition language files, and demonstrate this technique by extending a traditional unreachable code algorithm. To support more complex analysis, we then develop a technique for component developers to provide summary information about a component. This information can be integrated with several traditional static analysis techniques to analyze component-based systems more precisely. We then demonstrate the effectiveness of these techniques on several real Department of Defense (DoD) COTS component-based systems.

'Keep your nose clean, stay out of trouble and work hard!'
These are the words my father told me over and over when I was growing up, and again when I started my work towards this degree. These are probably the best words a father could give to his children to help them succeed in whatever they do. He passed away while I was still working hard for this degree, and I miss and love him dearly. I am sure that he is with me looking down from a better place. In fact, many of his traits can be seen in my two daughters. This is my motivation. This dissertation is dedicated to his eternal memory, and to my loving wife, daughters and family who help keep this motivation alive.

ACKNOWLEDGEMENTS

There are many people who have contributed to the successful completion of this dissertation. First and foremost, I want to thank GOD for all of his blessings and for giving me the strength and wisdom to do my best. Next, I feel that an honorary degree should go to my wife, Chris, for she has sacrificed so much in order for me to be successful. She is a very loving, intelligent and caring person who dedicates herself to her family and provides inspiration for us all. I also want to thank my daughters, Jessica and Julianna, who made sacrifices as well to see their daddy succeed. There were many nights where I had to see the sad look in their eyes when I had to tell them I could not lie down with them as they went to sleep, or I could not play with them because daddy had to finish his schoolwork. I also want to thank my entire family for their love and support. My mom made numerous novenas and other prayer services in her efforts to help. I especially want to thank all of the folks at MITRE for their help, understanding and support. Dr. Bob Miller and Mark Heller are two leaders who are role models for success. They understand the importance of this degree and what it means to me, and they allowed me to focus on it as much as possible. I owe special thanks to my good friend and colleague, John Ricci, who provided me daily motivation to keep me going, as well as his excellent skills at editing documents. John also helped to collect much of the data used in the experiments for this research. I want to thank Tuomas Salste, Randy Sparks, Charlie Altizer and James Garriss for their help in customizing and using many of the tools during this research. I want to thank Dr. Chris Wild for taking on the role of me to keep work progressing at MITRE while I took sabbatical leave to finish this dissertation. I also need to thank Sheila Allsbrook at MITRE and Phylis Woods at ODU for their unending office management support. They keep us all moving in the right direction. I extend many thanks to my committee members for their patience and hours of guidance on my research and the editing of this dissertation. Finally, my warmest thanks go to my advisor and friend, Dr. Mike Overstreet, whose untiring efforts to keep me focused have finally paid off. He has been asking me to go sailing with him for several years, so maybe it is time I do that.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

In an attempt to support the long-term goal of decreasing software development time and costs without unduly complicating future software maintenance, analytic techniques and tools, such as static code analysis, are used to assist with the understanding, development and maintenance of software.

Traditional static code analysis encompasses a mature set of techniques for helping understand and optimize programs. Most of these techniques use information resulting from the solution of one or more data flow problems, such as reaching definitions, available expressions, live-variable analysis, and definition-use chains [9, 59, 90]. Traditional approaches using data flow information may include dead code elimination [20], program slicing [63, 110, 116], and partial evaluation (code specialization) [37]. It is well understood that compared to other program analysis techniques (e.g., dynamic analysis [93]), static analysis techniques do a reasonable job for the costs associated with implementing them. This represents a trade-off, which is generally accepted, between the accuracy and usefulness of the information resulting from such algorithms, and the overhead to implement and conduct the analysis. One area of current research within the static analysis community is examining ways to improve the accuracy of information that these static techniques collect, store and use.

In an effort to decrease software costs and to shorten time-to-market, industry and government alike are moving away from more 'traditional' development approaches and towards integration of commercial off-the-shelf (COTS) components [21, 36, 48]. An interesting aspect of component-based development is that automated solutions are comprised of a variety of 'non-traditional' constructs. A program is a collection of distributed object-oriented components including, for example, forms, code snippets, reports, modules, databases, objects and containers. Components are glued to other components and controls using code snippets written in a visual language, such as one

---

The journal model used is *IEEE Transactions on Software Engineering*.

might do when establishing a command button on a form. As an example, consider a COTS solution that is typical in many organizations. Microsoft (MS) Visual Basic may be used as a fundamental integration mechanism to tie together a variety of MS Office products (e.g., Word, Excel, PowerPoint, Outlook), MS Back Office products (e.g., SQL Server) and web-based services/applications [27].

It is well documented that throughout an application's lifecycle the cost of software maintenance is typically much higher than the original cost of development (i.e., 60-80% of total cost [16, 32, 77]). Software maintenance also is the most overlooked and least structured phase of the lifecycle. The fact that component-based solutions represent a new methodology and a new set of challenges does not change this [52, 124, 127, 129]. The nature of the component-based environment is such that rapid change becomes the norm. On the one hand, the ability to effect rapid changes in the functionality of their software enables maintainers to respond quickly to changing requirements, thus shortening the maintenance lifecycle. On the other hand, component-based solutions require that maintainers be able to adapt to vendor-induced changes in the underlying components from which their applications are built. In all cases, maintainers are faced with handling the different types of maintenance (corrective, enhancement, adaptive, predictive) to their component-based systems.

Our experience shows that component-based development and the subsequent use of visual development environments, while reducing an application's total development time, can actually increase certain maintenance problems. A majority of the code in such an application resides in the individual components being reused, rather than having been written by the developer. Further, much of the source code for components is not available to the component user. Portions of the code are introduced automatically by the visual development environment (e.g., code generated when controls are dragged and dropped onto forms, or by programming 'wizards' in response to parameters entered by the developer). Moreover, we have found that problems such as the proliferation of dead code can be a common outgrowth of typical maintenance activities in these component-based environments. Consequently, the resulting overall application becomes more difficult to understand and maintain [106].

In this research, we examine closely the characteristics of a component-based system and document many of the issues that we feel make the development, analysis, testing and maintenance of such systems more difficult. We distinguish between component providers and component users and discuss the issues facing each. We classify component-based solutions by the type of information made available to the component user by the provider in an attempt to highlight the difficulties associated with analyzing a component-based system.

We identify situations where standard analysis techniques provide misleading or incomplete information when used on a component-based solution and show how traditional static analysis techniques can be extended in several to analyze component-based systems. By analyzing additional summary information for the components as well as any available source code for the application, we show ways in which traditional static analysis techniques may be augmented, thereby increasing the accuracy of static analysis results and ultimately making the maintenance of component-based applications a manageable task.

Our first technique leverages the minimal information typically available for components but which traditional static analysis techniques do not utilize. This approach aids component users attempting to analyze a component-based system by leveraging the semantic information about component properties contained in the type library or interface definition language (IDL) files associated with a component. We then show how this information can be used to augment traditional static analysis techniques for analyzing a typical component-based system. This technique can be useful for improving traditional tasks such as dead code detection.

While this technique is useful, we discuss additional ways to improve the analysis of component-based systems. Our second technique represents one such way a component developer could provide extended static analysis summary information with each component. This extended interface not only includes the standard interface information, but other information that could be useful for gaining insight into the component without having access to the source code. In this approach, the component provider uses analysis techniques to gather summary information that facilitates further

analysis and testing of those components by users without requiring access to the source code. The component provider makes the summary information available with the component. It is important to note that component providers are often unwilling to distribute source code for proprietary reasons. This technique requires cooperation of component providers but does help safeguard proprietary information about the component. The component user integrates the components with the user application, and queries the summary information during the analysis of the integrated system. Our technique summarizes global data flow analysis through variable def-use, first use/last use and parameter couplings. This technique can be useful for tasks such as interface-level coupling analysis and testing, slicing and ripple analysis, and integration testing.

We then demonstrate the effectiveness of these techniques in several case studies using real COTS component-based systems developed and maintained by the Department of Defense (DoD).

1.1. Contributions of this research

This research will make direct contributions to component-based solutions and in particular, those based on COTS products. COTS component-based solutions are becoming the norm. COTS products and distributed web-based components in particular are being used in DoD, Industry and Academia for all types of applications. Maintaining such systems in a cost-efficient manner is quickly becoming a concern to many organizations. The results of this work help establish a foundation for preparing organizations to tackle this problem by identifying some of the issues that need to be addressed as well as some tools and techniques that can be used to address some of them.

Currently, components that are used to construct a component-based system do not typically include the source code or any additional documentation that describes the component at any length. This is especially true for COTS components. This makes the analysis of these systems difficult at best. Today, without component source, often little or no information is available about the component that would be of use to static analysis techniques. For example, variable def-use, first-use/last-use, global reference and definition, and control flow information is not available. In some components, formal

parameter information is available in an external source (i.e., IDL file), but this external information is not being utilized in most of the techniques available today. The key is in leveraging the amount of information that is or can be made available about the components and incorporating that information ways that make analysis of the component-based system more precise. Potential sources for this additional information may include the insight of experienced programmers as well as information automatically generated by static analysis tools.

We discuss approaches that can be used to collect more static analysis insight about components. Using the additional information obtained, we show how traditional static analysis techniques can be augmented to analyze component-based systems. Through examples and case studies, we illustrate how the use of some traditional static code analysis techniques can aid in the understanding of these systems. Such techniques can have application for debugging, testing, integration and maintenance of component-based systems [46, 74, 79, 95, 111].

1.1.1. Goals

The primary goals achieved during the research for this thesis were to:

- Understand and document characteristics and potential issues associated with component-based applications which can make the software analysis, development, maintenance and testing of such applications more difficult.
- Develop new or extend traditional static analysis techniques for improved analysis of a component-based software system.
- Demonstrate that the use of additional information, such as semantic information about component properties, can be used to improve the quality of analyses. Some of this additional information can only be provided by experienced developers, and some can be extracted automatically.
- Validate the techniques on existing real systems.

1.1.2. Main results

This thesis contributes the following results:

- We describe distinguishing characteristics associated with component-based applications that can make software analysis, development, maintenance and testing more difficult.

- We develop a component taxonomy that characterizes potential engineering and maintenance problem areas.

- We classify the information necessary for static analysis of component-based solutions in terms of the type of component solution, the information provided by the component developer, and the techniques each classification can support.

- We identify what is lacking in the information available today for most components, and define additional component summary information that is needed from component developers to support system-level analysis using several traditional static analysis techniques.

- We define a schema for component developers to represent the static analysis summary information for a component using a standardized extensible markup language (XML) format.

- We demonstrate how several existing static analysis techniques may not be sufficient when applied to component-based solutions.

- We develop a technique to use semantic information about component properties obtained from type library and interface definition language files, and demonstrate the effectiveness of this technique by extending a traditional unreachable code algorithm.

- We develop a technique for component developers to provide summary information about a component that can be integrated with several traditional static analysis techniques to analyze component-based systems more precisely. We then show how a component user can integrate this information for system-level analysis.

- We develop several tools to illustrate these techniques in analyzing Visual Basic component-based systems.

- We provide experimental results that demonstrate the effectiveness of these techniques.

1.1.3. How this work differs from other work

The results reported in this thesis differ in several respects from other work. First, this work reports on new techniques for improving the static analysis of component-based systems. These are systems for which during analysis there usually is no source code available for some of the components to analyze. The first technique illustrates ways to do this using the information typically available today, but which current static analysis techniques generally do not exploit. We realize that better ways to improve the static analysis of such systems may exist, and our second technique presents one approach. This approach involves component providers distributing with their components summary information about that component which then can be used with traditional techniques to gain more precise information about the component-based system as a whole. The literature [55] shows some work identifying the need for component developer information without specifying any particular approach. We specify an approach to do this.

Throughout this work, three underlying design concerns are efficiency, reality and usability. With respect to efficiency, the literature contains many examples on program analyses that need hours to analyze even a small program. We are concerned with efficiency as much as accuracy, meaning that if extra precision is not likely to result, then an efficient solution is the better choice. Storage usage is also a concern. Many analyses simply generate too much information to be useable by maintainers. The other important design decisions are reality and usability. A main purpose of this work is to demonstrate that the static analysis of component-based systems for realistic languages is possible, and to transfer academic results to a realistic context. Thus, our experimental results were obtained by using these techniques on several real systems.

1.1.4. Assumptions and Disclaimers

This research focuses on the analysis of solutions based on Microsoft technology, namely using the Component Object Model (COM) and Visual Basic to integrate COTS applications. This technology is widely used in real systems, especially the DoD systems

used in our test cases for this research. Since our desire was to validate our work with real systems, we feel that our focus on this component technology is both key to validation and of immediate benefit to industry. While we believe that what is developed here will have general application to other component-based technologies, such as Common Object Request Broker Architecture (CORBA), validation in other domains is left for future research.

Two common categories of analyses that can be performed on programs are static and dynamic [120]. A static technique typically is used to analyze the source code of a program and produce some form of report. This requires some initial overhead in constructing a set of tools to do this analysis, but does not typically require any modification, augmentation, or execution of the original program to perform the analysis. Because of this, the results of a static analysis do not show the effect of the program execution, which means that the results may contain a certain amount of imprecision. However, it is generally accepted that the results of static analyses can be useful even with a certain amount of imprecision. In contrast, dynamic techniques are very precise as they monitor and analyze a program as it executes. Such techniques often require some form of instrumentation and recompilation of programs and possibly the creation of comprehensive test suites that provide complete coverage of all functionality [56, 60, 61, 122]. Those tests would be executed and reports collected. For a highly interactive system of even moderate complexity, that type of dynamic analysis, while potentially beneficial, can be prohibitively time consuming [8, 53, 54, 67, 123, 134]. In this work, we are interested in examining the feasibility and utility of analyzing component-based systems; therefore we limit our research to static data flow analysis techniques.

1.2. Overview of this thesis

This thesis is organized as follows. The next section gives background in the traditional static analysis techniques relevant to this work. We then discuss component-based solutions in terms of object-oriented design models, focusing on the competing COM/DCOM, CORBA and Java component models. We also characterize some aspects of component-based development that make analysis of such systems challenging. We

discuss the information typically available to describe a component interface such as the type library file. This is followed by a detailed discussion of issues in analyzing and testing COTS component-based systems.

Section 3 discusses the analysis of component-based systems. It starts with an examination of the literature to illustrate current research in this area. We then classify the information necessary for static analysis of component-based solutions in terms of the type of component solution, the information provided by the component developer, and the techniques for which each classification can support. We then demonstrate how several existing static analysis techniques may not be sufficient when applied to component-based solutions and suggest ways to augment these techniques for such systems.

Section 4 describes automated analysis techniques that are based on component property information. We develop and discuss a technique to use semantic information about component properties obtained from type library and interface definition language files and demonstrate the effectiveness of this technique by extending a traditional unreachable code algorithm.

Section 5 describes automated analysis techniques that are based on component developer summary information that can be generated by the component provider during development and testing of the component and then distributed to the component user. This can then be integrated with several traditional static analysis techniques to analyze component-based systems more precisely. We illustrate the effectiveness of this technique by modifying an existing analysis tool to generate extended call graphs embedded with summary information for global data flow analysis, and show the use of this information in support of a data dependence report.

Section 6 describes the tools developed or modified to support the automated analysis techniques developed as part of this research on component-based systems. The first tool provides the ability to capture component information from a type library or interface definition language file for use in integrating that information with Project Analyzer, a commercially available Visual Basic code analysis tool. The next tool is a set of extensions made to Project Analyzer to analyze particular component property criteria

of interest in support of detecting unreachable code. Next, another set of extensions to Project Analyzer is described to support the generation of summary information graphs that contain global data flow information such as variable def-use, first-use/last-use, and parameter couplings for each call graph node in a system being analyzed. The extended call graph with this summary information embedded is then stored in an extensible markup language (XML) format. The call graphs from separate and distinct components and a user application are then merged into an integrated system for further analysis. Finally, a number of extensible stylesheet language (XSL) scripts that provide additional analyses and views applied to the XML graphs are described, such as parameter coupling analysis, ripple analysis, and call graph metrics.

Section 7 demonstrates the effectiveness of these techniques on a number of case studies. Seven case studies represent real COTS component-based systems developed and maintained by the Department of Defense (DoD). Two additional case studies are used to represent academic examples designed to illustrate some interesting aspects of component-based development.

The remaining sections provide recommendations for future work and give some concluding remarks for this research. In addition, several supporting appendices provide detailed results of this research. Appendix A contains a comprehensive taxonomy listing of the issues associated with the analysis of a component-based system. Appendix B lists a sampling of the detailed case study reports from the experimentation. Finally, Appendix C documents the specific extensions made to Project Analyzer.

2. BACKGROUND

It is well understood that one of the greatest challenges in the maintenance and adaptation of long-lived software systems is the comprehension of system structure after months or years of modifications. Typically few structural aspects are reflected in changes to the corresponding portions of the system design documents. Maintainers are often left with a daunting task of discerning the design and functionality of the system from the only remaining trustable artifact, the source code. The fact that COTS component-based solutions represent a new methodology and a new set of challenges does not change this fact. Clearly, automated support for program comprehension is still required. In this section, we provide background information in the key traditional static analysis techniques that have relevance to this work. We then discuss component-based solutions in terms of object-oriented design models, focusing on the COM/DCOM component models. We also characterize some aspects of component-based development that make the analysis of such systems challenging. We discuss the information that is typically available to describe a component interface, such as the type library file. This is followed by a detailed discussion of issues in analyzing and testing COTS component-based systems.

2.1. Traditional program analysis techniques

Static code analysis encompasses a mature set of techniques for helping maintainers understand and optimize programs. Most of these techniques use information from one or more data flow problems, such as reaching definitions, available expressions, live-variable analysis, and definition-use chains. Traditional approaches using data flow information include: dead code elimination, program slicing, and partial evaluation (code specialization) [14, 15, 107, 133].

To implement such techniques for practical use, many static code analysis tools have been developed, ranging from source documenters to debuggers to language

checkers [31]. Most of the tools focus on the structure of the program itself, presenting diagrammatic representations of the block structure of the program, its calling structure, the flow of control, and related components. Some tools can detect anomalies and inconsistencies that normally are associated with errors and can be used by a maintainer to aid in program understanding. For example, a program slicer is one such tool [41-43, 76, 121]. Described below are some of the techniques that have relevance to our research.

2.1.1. Control dependence analysis

Control-dependence analysis determines, for each program statement, the predicates that control the execution of that statement. Control-dependence information is required for analyses, such as slicing, that are used for software engineering tools, such as debuggers, impact analyzers, and regression testers [55].

2.1.2. Data-flow analysis

Data-flow analysis determines information about variable definition and usage throughout a program. In discussing data flow analysis, it is important to identify three fundamental categories of algorithms: data flow equations, information flow relations, and dependence graph-based approaches [17, 62, 135]. Extensive research exists in applying each of these approaches to the analysis of basic programs with and without procedures, with unstructured control flow, and with composite data types/pointers [75], as well as distributed programs [12, 25, 35, 58]. Tip [120] does a fine job of providing a survey of these algorithms along with a detailed analysis and comparison of each.

2.1.3. Dependence graphs

Different ways of computing control and data-flow information have been proposed. An interprocedural analysis is typically carried out using a graph data structure to represent the program. Originally, techniques like slicing were computed by solving data-flow equations iteratively over a control flow graph [133]. Ottenstein and Ottenstein [91] then introduced a slicing technique that used a graph reachability algorithm on a

program dependence graph. Horwitz et al. [64] developed an interprocedural program representation, the system dependence graph, and an efficient two-pass slicing algorithm that uses it. Their two-pass algorithm computes more precise interprocedural slices than previous algorithms because it uses summary information at call sites to account for the calling context of procedures. Horwitz et al. [65] then extended their technique to object-oriented software by introducing the class dependence graph.

2.1.4. Program slicing

Introduced by Weiser [133], a program slice is a subset of an existing program that can provide information about a program variable in a particular statement; executing this subset should produce exactly the same sequence of values for that variable as the original program. Based on iterative data flow equations, two notions of a slice are commonly used: static and dynamic [47, 65]. A static slice represents the set of statements that may affect the value of a variable in a given instance in an arbitrary execution. In contrast, dynamic slices represent the set of statements that actually determine the value of a variable for a particular execution with a particular set of inputs. Slices have been used to represent both executable portions of a program as well as a set of statements that can be affected by a given slice criterion. Using data flow analysis techniques, program slicing has been shown to be useful to debugging, testing, program integration and maintenance [81, 97, 100, 101, 103, 104]. There are numerous variations on program slicing described in the literature [80, 116, 117, 120]. Two variants, ripple analysis and inter-modular slicing, have direct relevance to our research and are described separately.

2.1.5. Ripple analysis

Ripple analysis is a variant on program slicing [29]. When maintaining code, it is useful to know all procedural dependencies, data input dependencies, data modified by the code, and other program dependencies (e.g., constants, user-defined types) about the code [18, 40, 66, 119]. For example, it is useful to know that when module ***B*** is invoked by the main program, variable ***i*** may be assigned a value, and the values for variables ***x***, ***y***,

j and *i* may be used by the module. These dependencies may result directly from the execution of module ***B*** or from ***B***'s invocation of module ***A***. For a given statement in a program, it is also useful to identify the other statements in the code that contribute to these dependencies. A ripple analysis program can be used to identify such statements.

As part of previous Master's Degree research at Old Dominion University, ripple analysis was used to determine its usefulness as an aid to software maintenance [29, 31]. We developed a proof-of-concept ripple analysis tool to provide a means of identifying potential side effects of changing source code. Using a call graph and control flow graph of a program, a global data flow analysis was conducted to calculate flow-insensitive inter-procedural summary information similar to that described by Reps et al. in [96]. This information was then used to determine both reverse and forward ripples in response to queries posed by the maintainer. A reverse ripple describes the data flow associated with a given variable at a particular statement in the code. Using this analysis, a maintainer can examine the sequence of statements that execute in order to produce the value of the variable at that point. In addition, this analysis will show the data dependencies of other program variables that relate to the variable in question. A forward ripple can help answer the question: What will happen if this line of code is modified? Using this analysis, a maintainer can examine the source statements that will be affected as a result of a proposed change. It can provide insight into the resulting values of other variables that are dependent on the variable being analyzed.

In our maintenance activities, we were not concerned with producing executable slices but with identifying the nodes that were related to a given query. It is our experience that a report of this nature is useful because it provides a roadmap of the code to a maintainer contemplating a modification, particularly if that code is unfamiliar.

We have extended the notion of statement-level ripple analysis here to interface-level ripple analysis. This is important for showing the potential side effects across component or module boundaries. Throughout this thesis, we will use this form of ripple analysis to help illustrate the static analysis of component-based systems.

2.1.6. Intermodular slicing

Intermodular slicing is a variant of program slicing, and can also be referred to as interface slicing. Intraprocedural slicing is restricted to the statements within a procedure; it cannot derive information that is valid in the presence of procedure calls. Interprocedural slices have to model parameter passing from the call site of the procedure to the definition of the call. Reference parameters, global variables, and dynamically allocated objects complicate the computation of interprocedural slices due to the possibility of aliases. Intermodular slices can span across module boundaries. In languages that support separate compilation, the computation of slices on a per-module-basis should also be possible. Intermodular slices also allow slicing of programs that use libraries and slicing of incomplete programs [116, 117].

2.1.7. Data-flow testing

Data-flow testing uses data-flow information to guide the selection of test cases and to measure test-suite coverage for a program. In data-flow testing, an assignment to a variable in a program is tested by executing subpaths from the assignment (definition) to points where the variable is used (use) [55, 59].

2.1.8. Dead code detection

Aho et al. discuss the technique of dead code elimination [9]. This technique uses data-flow analysis to examine variables in a program to determine whether they are dead. A variable is live at a point in a program if its value can be used subsequently; otherwise it is dead at that point. This technique is also applied to statements and procedures. Bodek and Gupta [20] illustrate a slicing variant that introduces a prediction algorithm for achieving partial dead code elimination. The process of prediction embeds a statement in a control flow structure such that the statement is executed only if the execution follows a path along which the value computed by the statement is live.

2.1.9. Partial evaluation

Jones et al. [69] discuss partial evaluation, a program optimization technique also known as program specialization. A partial evaluator is an algorithm which, when given a program and values for some of its input data, produces a so-called residual or specialized program. Running the residual program on the remaining input data will yield the same result as running the original program on all of its input data. This technique is often employed for efficiency. Applications that benefit from this technique are computer graphics, database queries, neural networks and of course compilers. During the binding time analysis that occurs during partial evaluation, techniques such as program slicing are used extensively.

2.1.10. Coupling analysis & testing

Offutt et al. [68] discuss the use of coupling analysis for integration testing by examining the couplings between software components. Coupling measures the dependency relations between two units by examining the interconnections between them. They define twelve specific types of coupling and then summarize those into four types:

- Call coupling – A calls B or B calls A but there are no parameters, common variable references, or common references to external media between A and B.
- Parameter coupling – refers to all parameter passing. This type combines data and control coupling.
- Shared data coupling – refers to procedures that both refer to the same objects. This type combines non-local and global coupling.
- External device coupling – refers to procedures that both access the same external medium, such as a database.

The purpose of this section was not to discuss any of the above techniques in detail as the literature does this very well. The intent is to provide a brief overview of

some of the techniques that were considered during this research for the analysis of component-based systems.

2.2. Component-based solutions

Taking a page from hardware designers, software developers have gone from writing large systems in a particular language (e.g., C) to building systems out of prepackaged software components, each of which performs a particular function, and each of which provides a defined set of services through well-specified interfaces. A component-based solution typically has several distinguishing characteristics that are germane to its success [137]. Some of these are described below using Microsoft's COM architecture to illustrate these characteristics:

- Component Object Model (COM) [26]. A COM defines a common way to access software services, some of which may hitherto have been needlessly complex. It does this by providing a common service architecture across libraries, applications, system software and networks. COM is an object-oriented model that supports encapsulation, polymorphism, and interface-level inheritance. Component software implements its services through one or more COM objects via methods that are grouped into interfaces.

- Enabling COM technologies [26]. Technologies such as OLE Automation, ActiveX, and Distributed COM (DCOM) provide the mechanisms by which components can be integrated into a solution.

- COM-based applications. Client and server-based applications (e.g., MS Office, MS BackOffice, and Web applications) that support the COM model (thus exposing their services through COM interfaces in order to be programmable) provide reliable COTS-based functionality in the form of reusable components.

- COM-based development environments. Development environments (e.g., MS Visual Basic) that support the COM model provide easy-to-use tools to construct 'front-ends' to solutions that integrate components via code scripting.

Companies and developers are engaged in a major Internet-oriented thrust to build client-server applications based on distributed object models using various implementations, such as of Java/Remote Method Invocation (RMI), Common Object

Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) [11]. In [34] we provide a more detailed discussion and comparison between these approaches. The move away from large mainframe applications is explained by the advantages offered by distributed object technologies. Advocates of distributed object application development expect the enabling of:

- the re-use of existing functionality promoting Rapid Application Development (RAD) with plug-and-play-type interaction of distributed objects;
- isolated development and implementation of objects without adversely affecting other components;
- effective code maintenance including code augmentation and systematic distribution of updates; and
- lightweight (thin) client-side interfaces that connect to comprehensive server applications and data repositories in multiple locations.

To understand the potential issues in analyzing component-based systems, it is important to have some knowledge about the component object model being used, particular ways the components being used were designed and developed to interact with other components, and what information is typically available about the components to aid component users. The focus for this research is in analyzing systems based on the COM approach. The areas mentioned above are briefly described below with respect to COM-based components.

2.2.1. Distributed Component Object Model

DCOM was introduced in 1996 as Microsoft's solution to distributed object architectures. DCOM, previously known as Network Object Linking and Embedding (OLE), is an extension of the COM design to networked applications. DCOM possesses its own core network protocol Object Remote Procedure Call (ORPC). Key features engineered into the DCOM architecture comprise language independence (including strong bindings with Java), integrated Windows NT wire-level security, transport

neutrality (with the ability to communicate using TCP/IP, UDP/IP, IPX/SPX, AppleTalk, and HTTP), and static/dynamic invocation of objects.

The biggest gain that DCOM provides in developing a distributed application is its tight integration with Microsoft operating systems and applications. For example, distributed systems using DCOM as its middleware will be able to leverage the resources of components like Microsoft's Transaction Server and Internet Information Server (IIS) 4.0. Both of these technologies rely on DCOM for remote communications. Furthermore, numerous vendors are currently building COTS products and tools that are DCOM compatible, thus reducing the system development time by reusing these plug-and-play components. DCOM-based applications can also take full advantage of Microsoft NT security mechanisms. Since the first release of DCOM, the application programming interface (API), including the security model, has been made available to developers, making security implementations in DCOM highly configurable.

DCOM is obviously a proprietary solution and is well suited for the Microsoft-centric environment. However, it is currently the most widely used technology. The DCOM object model supports several attributes to support distributed component interoperability, but also contribute to making the analysis of systems based on this approach challenging. Some of these attributes include:

- Late binding mechanisms
- Encapsulation
- Polymorphism
- Interface inheritance
- Compound documents
- A component transfer format (e.g., COM structured storage)
- Uniform data transfers, including drag and drop
- Events and event connections or single and multicasting channels
- Some form of persistence.

COM defines binary calling conventions and binary interface definitions of a set of standard interfaces, typically referred to as the COM IDL [115]. This is important

because independent vendors can provide language implementations, including COM bindings, using this binary standard. COM still offers significant leeway to developers for different implementations of: COM libraries, type libraries, proxies and stubs, and standard services for a particular component.

2.2.2. Component-based development

Component software development promises to cut programming time and produce more robust applications, by allowing developers to assemble applications from tested, standardized components. The Component Object Model (COM) is a popular object technology designed to make it possible for software components that are custom developed to work with software components that are purchased off the shelf [51, 94, 98, 108, 136].

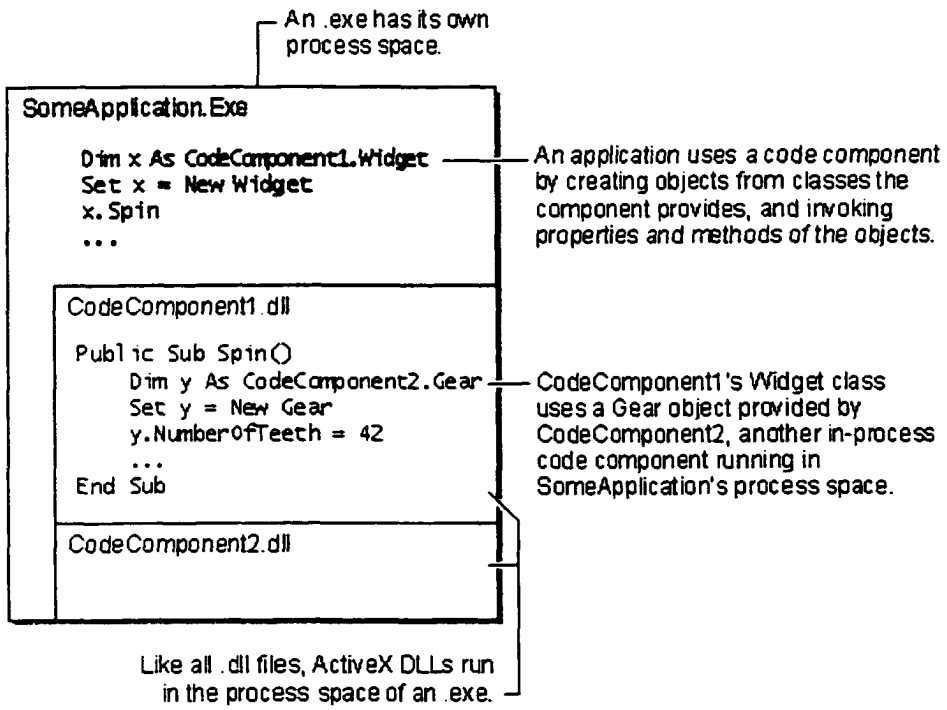

Figure 1. In-process components

An ActiveX component is a unit of executable code, such as an .exe, .dll, or .ocx file, that follows the ActiveX specification for providing objects [44, 45, 49]. ActiveX technology allows programmers to assemble these reusable software components into applications and services. ActiveX components can also be purchased off-the-shelf often to provide generic services such as numerical analysis or user interface elements. Custom components can be created that encapsulate business transactions and logic, and combined with generic components. Reusing tested, standardized code in this fashion is called component software development.

Components provide reusable code in the form of objects. An application that uses a component's code, by creating objects and calling their properties and methods, is referred to as a client. Components can run either in-process or out-of-process with respect to the clients that use their objects [70, 71].

Figure 1 depicts an in-process component. An in-process component, such as a .dll or .ocx file, runs in the same process as the client. It provides the fastest way of accessing objects, because property and method calls do not have to be marshaled across process boundaries. However, an in-process component must use the client's thread of execution.

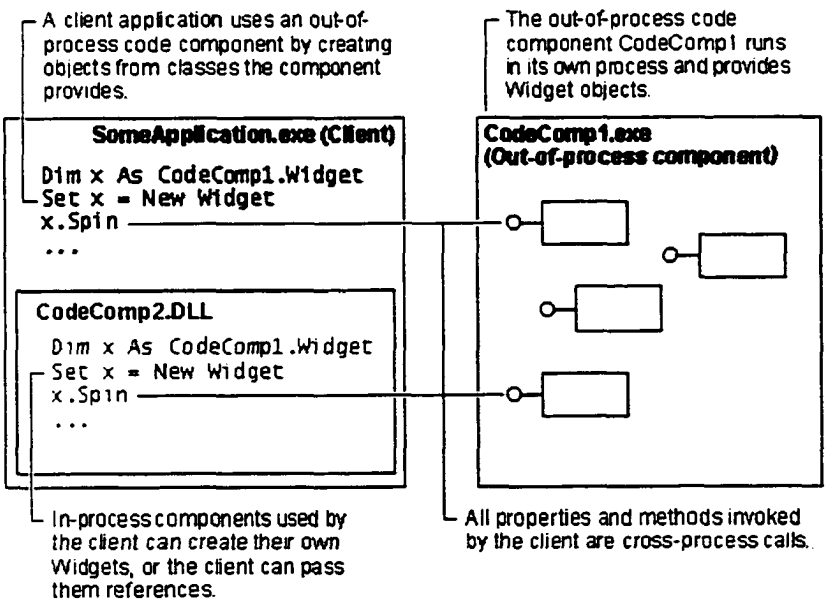

Figure 2. *Out-of-process components*

Figure 2 depicts an out-of-process component. An out-of-process component is an .exe file that runs in its own process, with its own thread of execution. Communication between a client and an out-of-process component is therefore called cross-process or out-of-process communication. Some reasons to create an out-of-process component include:

- The component can run as a standalone desktop application, like Microsoft Excel or Microsoft Word, in addition to providing objects.

- The component can process requests on an independent thread of execution, notifying the client of task completion using events or asynchronous callbacks. This frees the client to respond to the user.

A client and an in-process component share the same address space, so calls to the methods of an in-process component can use the client's stack to pass arguments. This is not possible for an out-of-process component; instead, the method arguments must be moved across the boundary between the two processes. This is called marshaling.

A client and an out-of-process component communicate via a proxy/stub mechanism. The proxy and stub handle the marshaling and unmarshaling of arguments passed to methods of the component; they are completely transparent to the client.

Marshaling is slower than passing parameters within a process, especially when parameters are passed by reference. For example, it is not possible to pass a pointer to another address space, so a copy must be marshaled to the component's address space. When the method is finished, the data must be copied back.

With respect to static analysis, the techniques described in this thesis will work for both in-process and out-of-process components. Both types of components provide the same information about the component, namely the IDL as described next.

2.2.3. Component Interface Design Language (IDL) and Type Library Interfaces

Components deliver services by providing classes from which clients can create objects. Clients use services by creating objects and calling their properties and methods. Information about the classes provided by a component is contained in a type library. In Visual Basic, for example, the type library is included as a resource in the compiled component. Clients access the type library by setting references to it [39].

A type library is best thought of as a binary version of an IDL file. It contains a binary description of the interfaces exposed by a component. An interface is a set of properties and methods, or events. Every class provided by a component has at least one interface, called the default interface, which is composed of all the properties and methods, along with their parameters and return types, that is declared in the class module. Events are outgoing interfaces, as opposed to the incoming interfaces composed of properties and methods. In other words, clients make requests by calling into a class's properties and methods, while the events raised by the class call out to event handlers in clients.

The COM IDL can be obtained from a type library for a component. For example, one way to do so is by using a Microsoft tool called OLE Viewer [5]. It provides a user with a list of type libraries that have been installed on the user's machine. Once a type library is selected, the text-based IDL can be displayed. Besides declaring COM interfaces, the IDL is also used to describe COM classes and dynamic link library (DLL) modules [82]. Several entities are used in the IDL file to describe this information:

- Interface – An interface represents a vtable interface, and describes the order, names and signatures of the methods and properties that make up that interface.

- Dispinterface – A dispinterface represents an automation interface, and describes the same information as an interface.

- Coclass – A coclass describes a COM class in terms of the interfaces and dispinterfaces that it exposes, and the outgoing interfaces that it supports. The coclass statement is also used for type information.

- Library – A library represents libraries imported or exported.

- Module – A module describes a DLL module by listing the names and ordinals of exported functions and global variables.

Many COM-based development tools, such as Visual Basic, are also capable of reading type libraries [4, 108, 112]. Since a type library provides information about component interfaces, tools can read this information and present it to programmers in an accessible format. Visual Basic, for example, has a feature named Auto List Members that displays a drop-down list while you are writing code and makes code suggestions for the methods and properties of a component being referenced.

In a later section, we show how the information can be obtained from a type library in a more user-friendly way, and incorporated into the Project Analyzer tool mentioned previously.

2.3. Issues in analyzing, testing and maintaining component-based solutions

It has been claimed that the use of Object-Oriented Design coupled with COTS component implementation constitutes a paradigm shift in software development. What effect will these new approaches to implementation have on software analysis and testing? In [34], we explore in detail issues in analyzing and testing component-based systems with particular emphasis on COTS systems. We found that many of the issues documented relate to testing [50]. In fact, examining testing concerns closely allowed us to uncover many of the issues that would have to be addressed to some extent for static analysis techniques to be useful. This is true because many static analysis techniques are used in testing tools. Below, we highlight some of the issues that could contribute to making the static analysis of component-based systems difficult.

2.3.1. Basic Object Analysis Issues

What distinguishes object-oriented programming from functional programming? Are objects not just a way of packaging data and functions? If so, what is fundamentally

different about analyzing and testing objects? Many of the techniques used for analyzing programs written in a functional style apply equally well to programs written using an object style. However several features unique to objects and differences in style and usage affect the analysis of objects.

Inheritance and the related concept of polymorphism are two features that many proponents claim distinguish objects from other programming concepts (such as abstract data types and modules). Inheritance permits the rapid development of new objects from existing objects whereby those features of the existing object that do not require modification can be reused and those requiring modification can be overridden. Polymorphism allows the same program to invoke one of several objects (which are related by inheritance) through run-time object identification.

Objects encapsulate both functions and data members. The values of the data members constitute the state of the object. The existence of Object State complicates testing. Many traditional testing approaches, for example, treat functions as stateless. That is, a function is considered a mapping from input to output and test cases can be defined as input/output pairs. However the state of an object is another possible input and/or output of the function. Techniques which trace data variables through a program from the point of their definition to their use overlook the variables defined in the Object State, since they are hidden from def-use analysis. Similarly the changing of an object's state as the result of a call to a method of the object is hidden. If that change is to a private data member, then it is not even possible to examine the value of this output variable to verify a test case.

Object-oriented programming encourages a different style of programming in which the problem domain is analyzed to identify the natural objects and their relationships and in which the programming attempts to reflect these natural objects and relationships. In addition, certain generally useful program infrastructure functions are cast into objects and made available as part of standard libraries. With the risk of oversimplifying, we can say that the use of object oriented design and programming affects analysis and testing in the following ways:

- Object implementations favor more and smaller functions, many of which make changes to, or report the value of, the object's internal state. For example, the existence of more functions will increase the amount of integration testing necessary.
- Encapsulating most data in objects means that most input/output parameters will be objects. In general, this will increase the number of input/output states that need to be tested. Because of the arbitrary size of objects, the preferred style of argument passing will be by reference. This opens opportunities for unanticipated changes to input parameters.
- The desire to produce objects that can be used in a variety of settings encourages passing the decision about exception and error handling from the object to the object's user. Most object languages support a structured exception handling mechanism, and its widespread adoption will increase the amount of code that needs to be tested. Moreover, the exception handling code is used infrequently and it is difficult to generate test cases that exercise exceptional conditions. This in turn increases the likelihood that faults will be present in the exception case code.

2.3.2. Object collections, components & patterns

While the use for object-oriented programming has gained wide acceptance with the availability of C++, Java, and, in some respects, Visual Basic, individual objects may not be in themselves the best units of reusability. The term component has been suggested by some as a level of abstraction and as a unit of reuse above the object level [118]. Components organize objects into services usually with a communications infrastructure and access model. Available COTS components range from simple graphical display components to embedded components encapsulating word processing objects, spreadsheets, etc. One problem with the increased size of components is the increasing number of ways in which those components can be used. A component as large as a spreadsheet may have too many paths and input/output cases to use traditional analysis approaches effectively.

2.3.3. Event-based programs

With the increased use of Graphical User Interfaces (GUIs) as well as distributed systems, the calling relationships among functions and components is a forest of threads, transactions and events. For static analysis this affects the utility of reachability analysis.

It is harder to identify dead code, for instance, where function invocation can be event driven and where a function's address can be placed in a table for use by various callback routing mechanisms. In fact much of the message passing to invoke functions in Microsoft Foundation Classes (MFC) is table based. The ability to bind a function to a message or event queue makes traditional calling charts less useful for testing purposes.

The trend towards distributed systems also favors a more event driven invocation of functionality than has traditionally been the case. Transaction oriented system favor a threaded implementation where threads are invoked to handle individual transaction and the state of the system is distributed among active threads and persistent data objects.

2.3.4. COTS components

The development of distributed systems using COTS component technology is relatively new. If component-based systems are to be successfully built and deployed, developers and users of components need to define design principles for such systems. The user of COTS components must be able to identify the best components for satisfying some mission need. Since it is unlikely that these components will be perfect matches, there will be parts of them that are not needed. In addition, there may be faults in these components that will require workarounds. Existing techniques for describing component functionality and environmental restrictions are inadequate. For example, assuming that not all features of a component are tested with equal rigor (perhaps because certain features are inherently more difficult to test), it would be desirable for a component provider to identify those features and usage patterns of a component that have been rigorously tested. Then a component user could restrict the usage of a component to this rigorously tested subset.

Numerous other issues relate to the particulars of the component development environment or the middleware standard used. For COM-based systems, some of these include:

- Component behavior on client machines. The behavior of the individual application being interfaced with is a function of its API and properties. When new versions of an integrated application are released, the expectant behavior,

API and properties may change. For example, loading a spreadsheet in a particular manner depends on the settings of its properties (like virus checking which may disable macros from working).

- Component Interface Availability. Dynamic Link Libraries (DLLs), vendor-supplied controls and the like may not provide a window into their APIs. If not, we cannot apply slicing for impact analysis. Even if so, lack of access to the source code makes the program understanding problem more difficult. For example, COM components (such as Microsoft Excel) do not provide information needed for static analysis. Component Vendors should supply the necessary interface information.

- Event Visibility. Events can be made inaccessible by particular values of object properties. For example, changing the width property of a form may 'hide' controls on that form. The code associated with those controls is still compiled into the program but events (e.g., mouse-click) for the control may be prohibited from running because no way exists for a user to interact with the control. Changing the visibility property of a control can have the same effect.

- Availability of Source Code. Software components may be built in-house or used off-the-shelf. The developer of a component has access to its source code. The user of an off-the-shelf component usually does not have access to the source code. For example, when using the Microsoft Excel component, there is no source code available. Depending on the availability of code, different testing and analysis techniques need to be used.

- Heterogeneity of Language, Platforms and Architectures. The components of a system may be written in different programming languages and for use on different hardware and software platforms. With middleware conforming to standards like CORBA and DCOM, components can interact with each other independent of the language and the platforms. When a system composed of such components is tested or analyzed, the methodology and tool used must be independent of the language and the platforms.

- Deadlocks and Race Conditions. Distributed or concurrent systems occasionally have problems related to race conditions and deadlocks. This is true for components. A good example of this is component callbacks. Consider the case where a client component calls a server component and waits. The server component calls back to the client (say for status notification purposes). This could potentially lead to deadlock. It would be good for testing and analysis tools to catch this. Standards such as COM state that the developer should not do this; however, there is no enforcement.

Although many more issues influence the analysis and testing of a component-based system, we feel that the issues we have discussed above are related to static analysis. Appendix A lists a more comprehensive laundry list of specific issues associated with the analysis and testing of component-based systems. In Appendix A, we discuss each issue by examining a number of viewpoints:

- Class – This classifies the issue with respect to the categories we have defined in this paper.
- Lifecycle – This describes which software engineering lifecycle an issue impacts.
- Issue – This is a description of the issue.
- Impact – This describes the potential impact the issue may have if not addressed.
- Example – This is an example of an occurrence of the issue.
- Mitigation – This attempts to provide a general solution to addressing the issue.
- Tools – This identifies specific tools and/or techniques that may be used to address the issue.
- Solution Risks – This describes any potential risks associated with using the solutions describe in the Mitigation and Tools section.

Many of the issues discussed above are not addressed in this research. However, we felt that it is important to have some general understanding of them as they relate to component-based solutions. The next section will examine specific issues that are addressed in this research.

3. ANALYSIS OF COMPONENT-BASED SOLUTIONS

This section discusses the analysis of component-based systems. It starts with an examination of the literature to illustrate current research in this area. We then classify the information necessary for static analysis of component-based solutions in terms of the type of component solution, the information provided by the component developer, and the techniques which each classification can support. We then demonstrate how several existing static analysis techniques may not be sufficient when applied to component-based solutions, and suggest ways to augment these techniques for such systems.

3.1. Current research in analyzing and testing component-based solutions

While the challenges of developing COTS component-based systems are great and many of the traditional analysis and testing methodologies are inadequate to handling component development solutions, many areas of research and development can be brought to bear on the problem.

It is important to point out that the most notable progress in this area actually comes from the testing community. This is more than likely due to the fact that testing a component-based system is primarily an integration testing concern where the interfaces of various components are being examined. As mentioned previously, static analysis techniques play an important role in many testing tools and techniques, so we feel that highlighting testing as much as static analysis is justified.

Verification and validation of complex computer systems is a very difficult undertaking. In the dozen or so years since David Parnas' impassioned arguments that concerns over software validation advise against the development of the Strategic Defense Initiative (STAR WARS) [92], little has changed. No fundamental breakthroughs have slayed the software failure dragon. Software reliability is currently addressed by a series of established 'best practices.' As the infrastructure upon which we build software systems has continually improved, best practice must also evolve. Given

the nature of computer systems that involve a mixture of legacy and newly developed components, we will need a mixture of best practices.

We may take some reassurance from the fact that while practically all complex systems contain residual faults, it is possible to make effective use of these flawed systems by avoiding, or compensating for, these faults.

This section surveys a number of promising approaches, techniques and tools that can be applied to the analysis and testing of COTS component-based systems, including:

- Augmenting components with testability features
- Developer providing testing documents
- Developer having component certified
- Formalizing integration testing and assertions
- User defining component usage patterns
- Use of state abstractions
- Component wrapping
- Extended static analysis techniques
- Regression Testing.

This research discussed in this thesis is focused on extended static analysis. However, we feel that it is important to have some knowledge about work in related areas. Some of this work is discussed in more detail below.

3.1.1. Augmenting components with testability features

While this approach requires additional work on the part of the developer, it may be necessary to assure certain testability conditions are met. It can be argued that object-oriented methods force the developer towards domain abstraction that will ease the testing process. On the other hand, it has been argued that encapsulating and hiding the internal state in an object complicates testing [89]. What is most likely is that proper design and formal documentation could help ease the testing problem.

Design for exhaustive testing: Because most testing involves a sampling of the behavior space of a program, it has been argued by proponents of formal verification

proofs that testing is inadequate to assure the correctness of mission-critical systems [23]. However exhaustive testing is adequate to assure the correctness of a program. In most cases, exhaustive testing is either infeasible or prohibitively expensive because of the size of the input space. In an interesting and relevant case study of a safety-critical computer controlled surgery system, John Knight points out that if the input space can be sufficiently reduced in cardinality, then exhaustive testing becomes possible [72]. Designing for exhaustive testing requires analyzing those features of a mission-critical system that are safety-critical and designing a solution (if possible) where the implementation of those features can be exhaustively tested.

Designing gray box components: Currently COTS components are delivered as black boxes (that is, without the source code). This limits possible testing strategies. While it is sometimes possible to obtain source, it is not clear that testing at this level of detail is needed or desired. Source code over-specifies the component. Changes in the component's implementation that do not change its perceived functionality should be allowed. The user of COTS components is not necessarily interested in testing the component itself but rather in assessing whether that component is being properly used. Williams [137] suggests that component developers provide a special interface for testers giving access to information that would ease the job of integration testing. The use of a state-based approach has been advocated by some [128]. Since states can be defined in various level of detail, providing access to some internal, but abstract state of a component may form a compromise between black and white box testing. The nature of this interface is a design issue.

3.1.2. Developer providing testing documents

Gray box testing [22] is a level of testing between black and white box testing. One advantage of gray box testing is that it forces a level of abstraction on the component. It lets the users know a little more about the implementation of the component. If this abstraction is described formally, it may be possible to automate the testing process. Buchi [22] argues that the theory of program refinement provides formalism for specifying properties of the component that can be used in testing.

Marick [83] argues that developers often have insights into how a user's misuse of the abstractions that underlie a software component will manifest itself in program failures. Since many programmers use common clichés for developing programs, they often fall into common traps. Thus a developer could provide testing specifications, test cases, or assertions that would discover these misuses.

3.1.3. Developer having component certified

As part of NIST's Advanced Technology Program in the area of 'Component-Based Software,' Jeffery Voas and Jeffery Payne have proposed a certification scheme based on a 'Test Quality Rating' (TQR). TQR measures the thoroughness of testing using previously published 'Squeeze Play' techniques. Squeeze Play measures the ability of a program to hide faults using the Propagate, Infect, and Execute (PIE) analysis. The authors claim that market pressures could drive developers to submit their components to independent certification analysis [126, 130].

3.1.4. Formalizing integration testing and assertions

Noting the difference between traditional hierarchically structured software systems based on information passing by function calls and distributed client/server and peer-to-peer systems based on message passing, Mercier et al. [88] define an 'information space' formalism to use as the basis for integration testing. This formalism allows a decomposition of the system to identify subspaces of the component/methods interrelated by information sharing. Testing focuses on these subspaces.

Assertions have been proposed as a mechanism for assisting both integration testing and operational reliability. Assertions have often been touted as a defensive programming technique that can be used to uncover problems in software systems. However, the effective use of assertions remains an art and is therefore infrequently used. Voas and Kassab [131] argue for the use of assertions at those places in a program where testability analysis indicate that it will be difficult to provide adequate testing. In [73], John Knight et al. propose a reversal check for testing complex numerical calculations that could be used as an assertion on operational reliability.

3.1.5. User defining component usage patterns

It has been recognized for a long time that the use of complex systems is simplified through the adoption of certain patterns of usage. Such usage patterns not only reduce the complexity of these systems but also integrate a number of individual interactions into larger abstractions that more appropriately fit the underlying business processes of the organization. Employment of usage patterns can be used not only to guide the development process [13, 84, 85] but they can also form the basis of test specifications.

3.1.6. Use of state abstractions

One of the distinguishing characteristics of components is the presence of internal state. This state can be defined as a vector consisting of the value of all the component's data members. This state is persistent between calls to the component's member functions. For testing purposes, the internal state can be considered as input to a member function call. Thus internal state can contribute significantly to the number and nature of test cases to be considered. In fact, the need to create a known internal state for testing purposes means that test input sequences must be used as the test cases instead of a single input/output test pair.

The direct use of the internal state is important in the process of unit testing. However for the purposes of integration testing, it may be more productive to use abstract component states that summarize the internal state [19, 87]. If these states are related to the typical ways a component is used, then they may support usage coverage metrics.

3.1.7. Component wrapping

In the absence of certification of the reliability of a component, several authors have suggested that wrappers be used to isolate the component. John Knight suggests the concept of a shell that wraps the component and is used to assure certain properties of that component [73, 125]. Jeffery Voas also suggest the use of software wrappers to isolate possibly malicious code.

3.1.8. Extended static analysis techniques

Previously we discussed the importance of static analysis in program understanding, debugging and testing. The focus of this research is in extending static analysis techniques so that they may do a better job at analyzing a component-based system. This is especially true for components that are COTS-based and for which no source is available. In this case, many analysis techniques would be more useful if they could be extended to incorporate more information about the component. One method discussed deals with obtaining more information about components from component.

For example, the component provider could provide extended static analysis summary information. This means that the component provider does not provide source code, but does provide an extended interface and possibly documentation. This extended interface may include not only the standard interface information, but other information that would be useful for gaining insight into the component without having access to the source code (e.g., an application programming interface for summary static analysis information). Harrold, Liang, and Sinha [55] do a nice job of suggesting this concept and show potential ways that this information could be used to support program slicing, control-dependence analysis, and data flow testing. Their approach separates the analysis and testing of the user application from the analysis and testing of the components. In this approach, the component provider tests the components and, using analysis techniques, gathers summary information that facilitates further analysis and testing of those components by users without requiring access to the source code. The component provider then makes the summary information available with the component. The component user integrates the components with the user application, and queries the summary information to drive the analysis and testing of the integrated system. The summary information obviates the need to access the components' source code. One issue that needs to be addressed by this approach is that the summary information provided with the component should be represented in a standard notation that is independent of the language in which the component is implemented. Another issue is how to provide access to this information. The component must then also provide

suitable query facilities (e.g., methods or operations) to retrieve the summary information. Harrold et al. provide no specific ways or implementations to solve this problem. We do so in this thesis.

There are other related approaches being researched as well. For example, in [126, 132], Jeffery Voas describes a mitigation technique for defending against COTS software failures that relies on the developers to supply static fault tree analysis and backward static slicing.

3.1.9. Regression testing

Winter [138] argues that object-oriented programming encourages more incremental development. This, in turn, increases the need for regression testing. Winter's work uses the class message diagram for change impact analysis. It also emphasizes the use of good architectural principles to ease testing.

See [105] for other work on regression testing. Many researchers have addressed the selective retest problem for procedural-language software [17, 33, 53, 76, 78, 101, 102, 116]. Prior to the development of the algorithm discussed in this report, the only technique to address the problem with respect to object-oriented software is by Harrold [55, 57] and applied only to test selection for derived classes. The emphasis on code reuse in the object-oriented paradigm both increases the cost of regression testing, and provides greater potential for obtaining savings by using selective retest methods. When a class is modified, the modifications impact every applications program that uses the class and every class derived from the class; ideally, we should retest every such program and derived class [111, 134]. The object-oriented paradigm also alters the focus of test selection algorithms, emphasizing and creating different concerns. For example, since most classes consist of small interacting methods, selective retest approaches for object-oriented programs must work at the inter-procedural level. Also, since many methods for testing object-oriented software treat classes as testable entities and design or employ suites of class tests for classes [46, 62, 122], selective retest methods must support the use of class tests.

3.1.10. Other related work

Jeffrey Voas argues for certification of COTS components by independent testing labs [132]. Buy et al. combine static analysis and symbolic execution for the testing of object-oriented components [24].

Steindl [116] is doing research with intermodular slicing. Intermodular slices can span across module boundaries. In languages that support separate compilation, the computation of slices on a per-module-basis should also be possible. Intermodular slices also allow slicing of programs that use libraries and slicing of incomplete programs.

Offutt et al. [68] discuss the use of coupling analysis for integration testing by examining the couplings between software components. Coupling measures the dependency relations between two units by examining the interconnections between them. They have developed a technique for conducting a dynamic analysis of instrumented Java code as a way to monitor coupling across module interfaces. This technique does a nice job at handling polymorphic call sites.

In general, much research has identified issues associated with the development, analysis and testing of a component-based system. Also much discussion exists pertaining to components in general and potentially better ways to develop them, to certify them, and to provide more support to component users for testing them. Many of the techniques and literature cited above address these same concerns. Many of the techniques also suggest the use of dynamic analysis to aid the testing process. A good example of this is the coupling analysis and testing method discussed above. However, static analysis, mainly due to its lower implementation cost compared to dynamic analysis and the utility it provides, is still an important method to be considered. When dealing with components for which no source code is available, static analysis becomes difficult to perform. The issues have been brought up in the literature as noted above, but methods and implementations are now needed. This thesis offers two methods.

3.2. Classification of component-based analysis techniques

Component-based software is one class of software for which efficient and effective program analysis based testing and maintenance tools will be useful. As we have seen, a component-based system is composed primarily of components: modules that encapsulate both data and functionality and that are configurable through parameters at run-time.

The issues that arise in the analysis and testing of component-based systems can be viewed from two perspectives: that of the component provider and that of the component user. The component provider perspective addresses analysis and testing issues that are of interest to the provider of the software components. The component user perspective, in contrast, addresses analysis and testing issues that concern the user of software components. The component provider views the components independently of the context in which components are used. The provider must, therefore, effectively test all configurations of the components in a context-independent manner. The component user views the components as context-dependent units because the component user's application provides the context in which the components are used. The component user is thus concerned with only those configurations or aspects of the behavior of the components that are relevant to the component user's application. Another factor that distinguishes the pertinent issues in the two perspectives is availability of the source code of the components: the component providers have access to the source code, whereas the component users typically do not. What is usually available from components, however, is some sort of interface specification. This interface usually consists of one or more method signatures and is specified in an Interface Description Language commonly known as IDL. Clients can invoke these methods on the corresponding server. A method signature specifies its name, parameters passed between the server and its clients when the method is invoked, a return type, and zero or more exceptions that could be raised during its execution. Furthermore, each component is a program written in one or more programming languages. Components are assumed to be distributed over a network of machines that may not all have the same run time environment. As such, the system is

considered dynamic if the components can change after the system is launched or during its execution.

A component-based system then can be classified by the type of information provided to component users by component providers. For example:

- No component provider information (Worst). Component provider does not provide source code, interface or documentation for a given component.

- Minimal component provider information (Average). Component provider does not provide source code, but does provide the standard interface (e.g., type library) and possibly documentation for a given component.

- Extended component provider information (Better). Component provider does not provide source code, but does provide an extended interface and possibly documentation. This extended interface not only includes the standard interface information, but other information that would be useful for gaining insight into the component without having access to the source code. Examples of this might include: an API to obtain summary static analysis information that was collected by the component provider and made available; a testing interface (gray box) for conducting various tests by the component user; a set of reusable test cases for the component; and information about the states and exceptions of the component.

- Full component provider information (Best). Component provider does provide full source and any documentation for the component.

System solutions would typically comprise a mix of these classes of components. The best case probably will rarely happen since that goes against the proclaimed advantages of component-based development in the first place. The third case is really an ideal case, but it is one that will require significant research and support by component providers. Most of the non-traditional approaches discussed in this thesis fall into this classification. A hard problem here is determining how we can specify components better to support automated analysis and testing. In any case, it is still true that in a development environment one needs a methodology to test both components and systems.

With respect to the two perspectives mentioned above, a more detailed examination of the issues in the analysis and testing of component-based systems is warranted. Component users typically develop component-based systems by integrating

their applications with independently developed components; this presents several challenges for adapting traditional static analysis techniques.

First, as we mentioned above, source code is usually not available for the independently developed components. Traditional static analysis (e.g., alias analysis and control-dependence computation) and testing (e.g., data-flow testing) techniques require access to the source code of the system being analyzed or tested. One way to employ these techniques without the source code is to make conservative approximations about the analysis relations that hold in the components and about analysis relations caused in the user application by the components' code. Such approximations, however, can cause the analysis results to be too imprecise to be useful. Second, in a component-based system, even if the source is available, the components and the user application may have been implemented in different languages. Current analysis tools typically only work on the implementation language. Third, a component often provides more functionality than that which is used by a particular application. Without identifying that portion of the functionality exercised by a particular user application (a process we refer to as usage analysis), a testing or analysis tool can report imprecise results or require more testing than necessary in the context of a particular application.

Component providers develop and test software components independently of the applications that use the components. Unlike component users, the component providers have access to the source code of the components. Therefore, testing a component is similar to traditional unit testing. However, traditional unit testing criteria, such as statement or branch testing, may not be sufficient for testing a component because of the weak fault-detection capabilities of those criteria. Fixing a fault subsequently revealed in a component by the component user would typically entail a much higher cost than fixing a similar fault detected during integration testing of a non-component-based system because of the potential use of such a component by many user applications. The component provider must really address two issues. First, the provider must effectively test the components as context-independent units of software. Effective and adequate testing of software components in this manner increases the user's confidence in the quality of those components. Second, the component provider must support better testing

and analysis of the user applications so that faults related to components can more readily be revealed before the user applications are released. This can improve the quality of a component-based system and reduce the cost of testing and maintenance of the system.

Using the classifications made above, the methods developed as part of this research address the minimal component provider information and the extended component provider information classes respectively.

```
Option Explicit
Public x As Integer
Public y As Integer
Public i As Integer
Public j As Integer

Public Sub a(x As Integer, ByVal y As Integer)
Dim a As Integer, b As Integer
    a = j - y
    b = x - y
    If a < b Then
      x = 0
    Else
      x = 1
    End If
End Sub

Public Sub b(ByVal x As Integer, ByVal y As Integer)
    Call a(i, y)
End Sub

Public Sub Main()
    x = 5
    y = 3
    i = 8
    j = 10
    Call b(x, y)
    Call a(x, y)
    MsgBox "x=" & x
    MsgBox "y=" & y
    MsgBox "i=" & i
    MsgBox "j=" & j
End Sub
```

Figure 3. A simple test.bas example

3.3. Applying traditional techniques on a component-based solution

To help understand the impact of applying traditional techniques on a component-based system, it is important to examine what happens when such techniques are applied to a more traditional example in comparison to one based on using components. In a component-based system, two key problems pertain to applying traditional static analysis techniques:

- Often no source code is available for many of the components being used.

- The information that is available about a component, such as the type library or IDL information, is often not utilized to augment the static analysis of a component-based program.

To illustrate, we will examine two key examples that address each of the two fundamental problems listed above. The first key example represents a traditional non-component based program that can be used to show the effect of both key problems. Figure 3 lists the source code for a simple test program, test.bas, written in Visual Basic. This program has three procedures including one main procedure. Several global variables are also defined to show the effect of global variable usage in the analysis. If we apply some traditional static analysis techniques to this program, we should observe results similar to Figure 4 below.

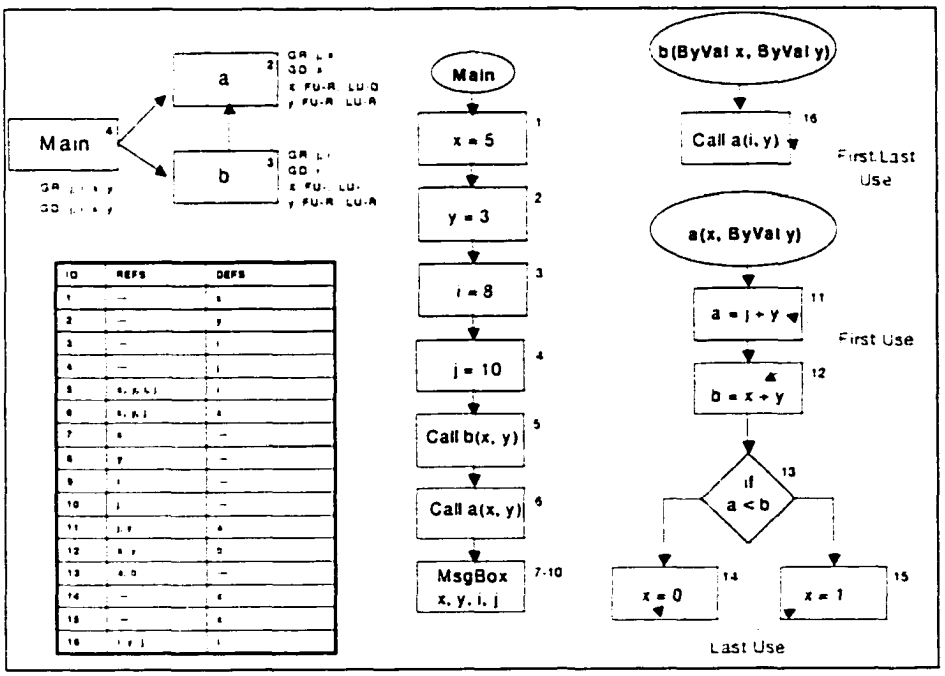

Figure 4. Traditional analysis of test.bas

In Figure 4, we actually see the culmination of several analysis techniques. The first analysis that is typically done is to address call and control flow paths throughout a program. The result of this analysis is usually some form of a call graph and a series of control flow graphs, one for each node in the call graph. The call graph for test.bas shows the *main* procedure calling both procedures *a* and *b*. Procedure *b* in turn calls procedure *a*. The control flow depicts paths of execution and is represented by the control flow graphs for each procedure, containing nodes for each statement of code. The next analysis done for this example is data flow. First, an analysis was conducted to construct variable reference and definition information for each node in each control flow graph. The results of this statement-level analysis are shown in the table listed in the figure and identifies for each node the variables defined and referenced at that node. It is important to note that the analysis included global effects across the system. For example, the *globalref* list for node 5 contains the variables *x*, *y*, *i*, and *j*. Variables *i* and *j* are included due to the global effects of the procedure call. As part of this analysis, a *globalref* and *globaldef* list was also conducted for each node in the call graph and represents static analysis summary information about each module. For example, the *globalref* list for module *a* contains the variables *x* and *j* to identify the global effects of variable references associated with module *a*. The variable *x* in this case is a formal parameter that is passed by reference (i.e., ByRef in Visual Basic terms). Since the value of ByRef parameters could extend beyond the scope of the module, it is included as a global effect and is eventually mapped to the appropriate actual parameters during the analysis. The next analysis being shown is variable first use and last use information for each of the formal parameters. This analysis identifies for each formal parameter whether the first and last use of that variable is a reference or a definition and stores that as part of the summary information for each call graph node. For example, looking at module *b*, we see that the first and last use of formal parameter *y* is a reference at node 16, which is a procedure call to module *a* where the formal parameter is ByVal. We also see the formal parameter *x* is not used at all, representing a dead variable usage. This information is stored in the summary for module *b*, which is node 3 in the call graph. Once this basic analysis is completed, and the appropriate summary information obtained, it can be used to support

several useful techniques to aid testing, impact analysis and program understanding of such systems. Two of these techniques are briefly exemplified below.

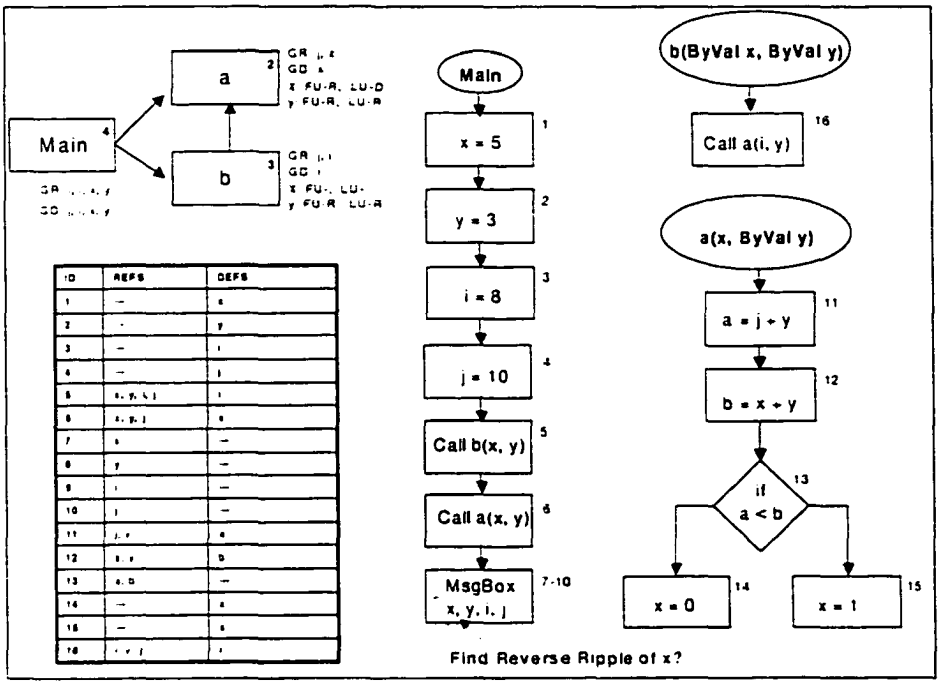

Figure 5. Reverse ripple analysis of test.bas

Figure 5 illustrates a static analysis technique that leverages the variable global usage analysis that was done on test.bas to calculate statement-level ref-def lists as well as the module-level *globalref* and *globaldef* lists. The technique demonstrated is the statement-level reverse ripple analysis that was discussed previously. This example performs a reverse ripple analysis with respect to the variable reference to *x* on node 7 of module main's control flow graph. The reverse ripple analysis is used for impact analysis on referenced variables to discover all potential nodes of interest that have contributed to the current value of the variable being examined. In effect, starting with the node the variable is on, the analysis works backwards through the program execution paths to find all places that define the value of the variable, and recursively continues the ripple analysis on each of the variables that contribute to the variable definition at that point. In this example, nodes 1, 5, 6, 15, 15 and 16 represent the potential impacts.

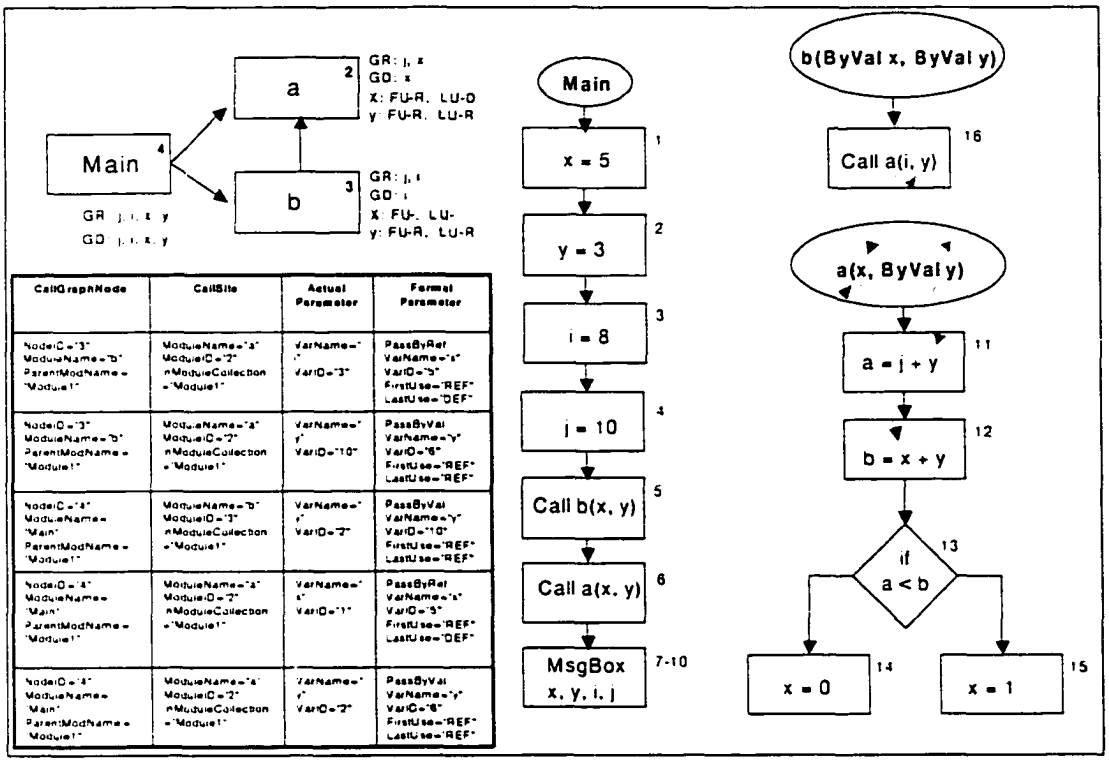

Figure 6. Coupling analysis of test.bas

Figure 6 illustrates a static analysis technique that leverages the variable first-use and last-use analysis that was done on the test.bas program. The technique demonstrated is the coupling analysis technique discussed previously. This example uses the first-use information stored in the summary nodes of the call graph for test.bas to calculate parameter coupling paths. Parameter coupling paths exist between call sites and called modules if for each actual parameter at a call site, a path exists from the last definition of that actual parameter prior to the call site to the first reference usage of the mapped formal parameter within the called module. In the example above, five coupling paths are identified. In the call site to module *a* in module *b*, two paths exist between modules *a* and *b* for each of the parameters. Likewise, at the call site to module *a* in module main, two paths exist between modules *main* and *a* for each of the parameters. The final path is between the actual parameter *y* in module *main* to the formal parameter *y* in module *b* at the call site to *b* in module *main*. No path exists between the actual parameter *x* and

formal parameter *x* because *x* is not used in module *b*. Coupling analysis is an effective aid to testing the interfaces between modules as it can be used to help determine which tests need to be accomplished.

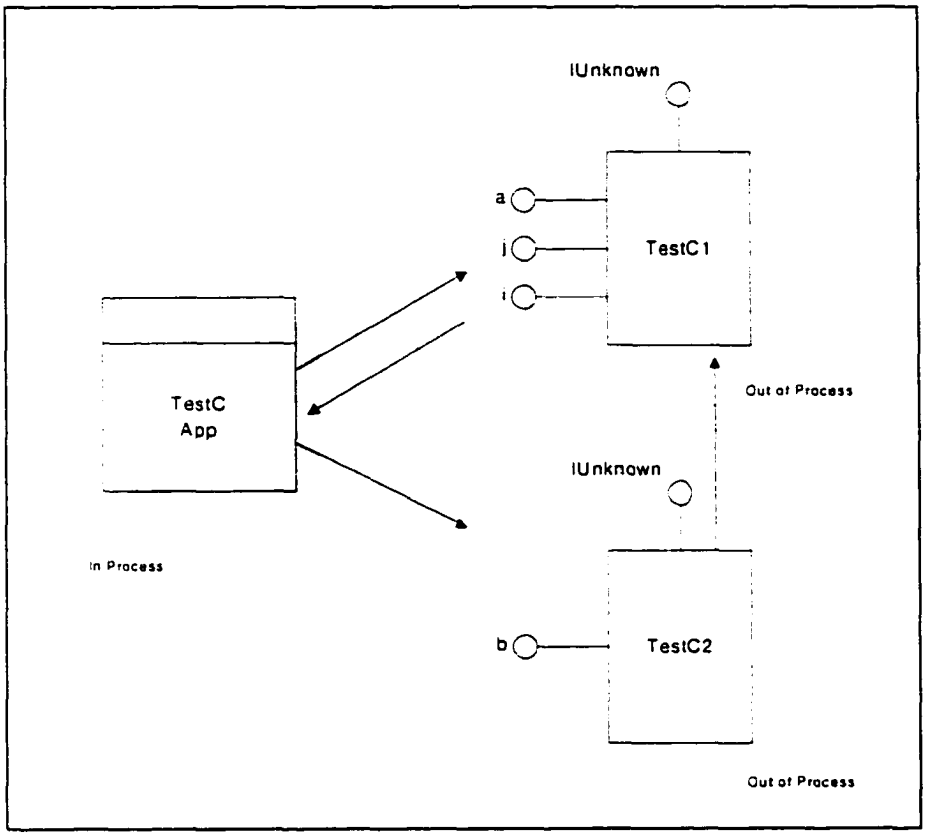

Figure 7. Component-based example of test.bas

The above techniques represent very effective capabilities that can be accomplished when the appropriate static analysis information can be obtained from the program being analyzed. Many techniques, such as the ripple and coupling analysis discussed, require information that can be obtained through a detailed control and data flow analysis of the program. The techniques discussed can be very effective when applied to non-component-based programs, such as the test.bas example. But what if the test.bas program were component-based?

Figure 7 depicts a component-based version of the test.bas program, called TestC. In this example, the TestC.bas program has one main module that has calls to several interfaces in two separate ActiveX out-of-process components TestC1 and TestC2. The TestC1 component offers an interface to the method *a* which corresponds to the module *a* from the test.bas example. It also has methods for setting and obtaining the value of the *i* and *j* properties. The TestC2 component has an interface to method *b*. As in most component-based systems, the components are made available as binary objects without the availability of source code. The interface information for each component can be obtained through the type library information for the component.

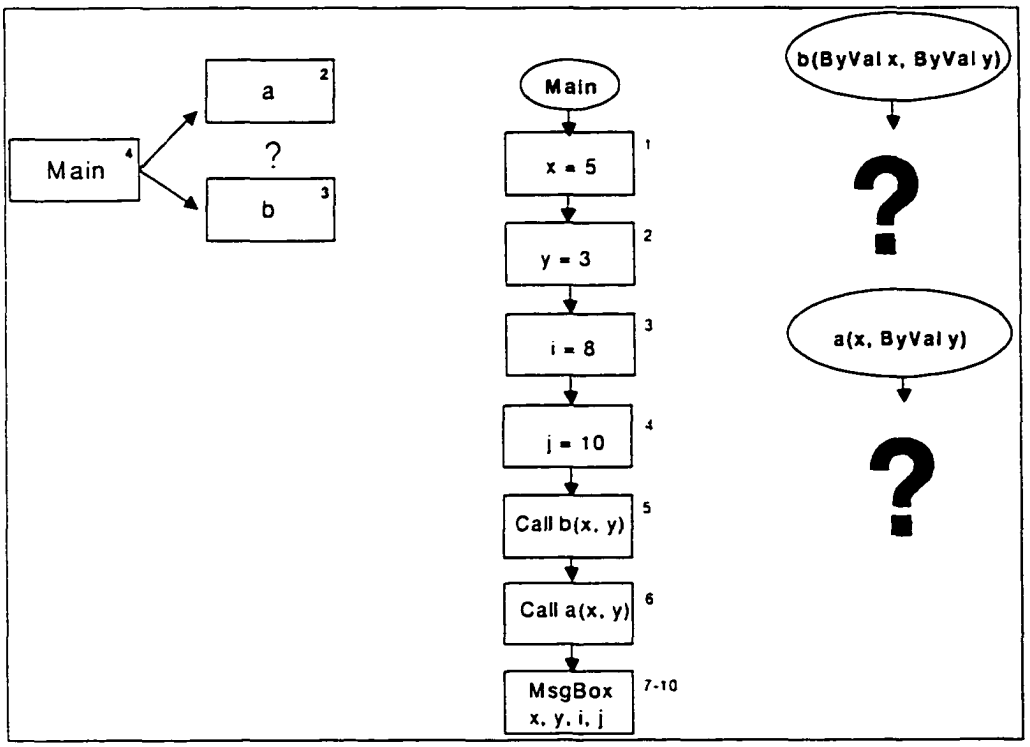

Figure 8. Traditional analysis of testc.bas

The example depicted in Figure 8 shows the effect of attempting to apply the same analysis techniques that were applied to the test.bas program previously. Since the only source that is available for analysis is the module *main* in the user application TestC.bas, the call and control flow analysis will be limited. The call graph can be

generated to show the calls from module *main* to the interfaces in the components. However, the call graph is limited because the call from interface *b* in the TestC2 component to interface *a* in the TestC1 component cannot be detected. The only control flow graph that can be constructed is the one for module *main*. If the analysis takes into account the type library information, then the formal parameter information for each interface can be obtained. Although useful for some basic analysis techniques such as interface-level parameter dependence and call coupling, this information is not sufficient to support the global ref-def analysis, first-use and last-use analysis, ripple analysis and detailed coupling analysis techniques that were applied to the test.bas program.

Having no source code available for some or all of the components in a component-based system means that the types of detailed static analysis information necessary to support advanced analysis techniques cannot be obtained. Likewise, not using the information that is available in the type library for a component, for example, means that the results of any analysis may not be precise. The next example will illustrate this point further.

The second key example is a specific examination of a real DoD system that is constructed using a number of COTS components. In [33], we make a claim that traditional approaches to automated testing techniques are not sufficient for analyzing component-based solutions. To support our claim, we cited some preliminary research conducted in conjunction with real-world maintenance and testing of component-based systems [52, 124, 127, 129]. We then described our experience in developing and maintaining a Commercial off-the-shelf (COTS) component-based solution for the Department of Defense (DoD), highlighting particular maintenance issues which arose as a result of using this new programming methodology [27, 28]. A brief summary of this experience is warranted here to help illustrate a case where the necessary type library information for the components in a particular application could have been used to augment the traditional static analysis techniques used.

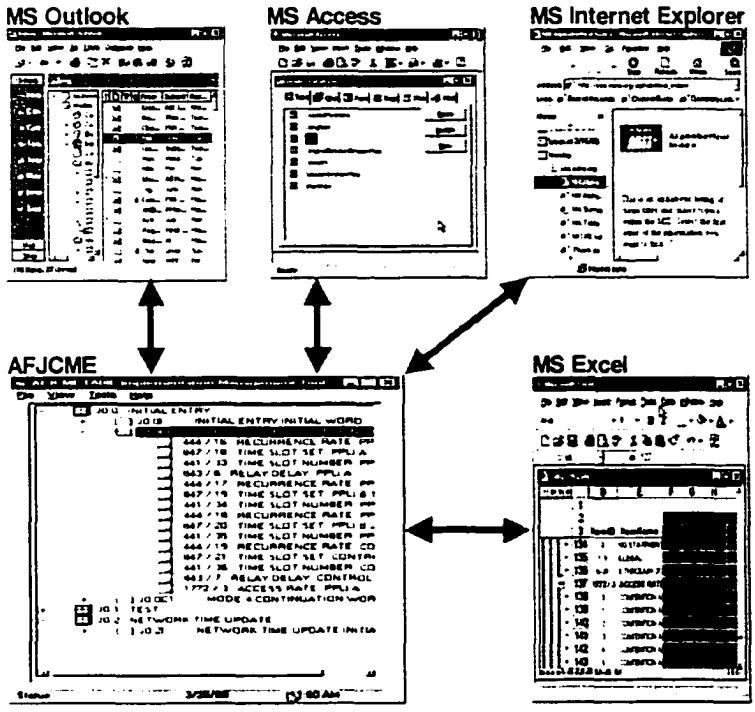

Figure 9. AFJCME system

Our example solution, depicted in Figure 9, is a special-purpose tool (called AFJCME) built to support analysts in the maintenance and design of data interoperability standards and in the creation, maintenance and analysis of system implementations of those standards [30]. The tool extracts implementation data from a database, and presents it to the analyst in spreadsheet form for modification. It further facilitates navigation among the data items by building a tree-structure from the database that can be expanded and compressed as needed. These capabilities are implemented by a collection of COTS components, as Figure 9 illustrates. MS Visual Basic is used to implement the user interface and tree-structure, and to provide overall program control and integration [10]. By the mechanism of OLE Automation, the tool interfaces with MS Access for database management, MS Excel for spreadsheet support, MS Outlook for event logging, and the Internet via a Web browser to provide access to online DoD standards information [112, 114].

Our experience was based on taking the original developed baseline of an application through a five-month maintenance cycle full of changing user requirements,

error corrections, and a need to produce a specialized version of the program. In this specialized version of the tool ('AFJCME_Lite'), links to Outlook and the Internet are eliminated, and the database provides only the navigation tree data. Next, we showed our use of basic traditional static code analysis techniques, in particular live variable analysis and dead code identification, to help our maintenance effort.

An informal survey of available tools led us to Project Analyzer, a shareware Visual Basic analysis tool [109]. We used Project Analyzer to help us perform some basic static analysis techniques such as detailed cross-reference reports and the identification of dead code and variables. Project Analyzer makes a full, two-phase source code analysis. In phase 1, basic information about the structure of the project is collected, including procedure names, procedure parameter names, variable names, constant names, type definitions, and control names. Some basic metrics (e.g., lines of code) are calculated as well. In phase 2, cross-references are detected. The whole project is scanned to find where procedures are called, where variables are assigned a value or used, where constants are referenced, etc. Other metrics, such as nested conditionals, are also calculated in this phase. Finally, Project Analyzer calculates additional information based on the cross-reference data to determine if any procedures, variables, constants, and types are dead.

Project Analyzer defines a dead procedure as one that cannot be executed at run-time. It tags a procedure as dead if either: (1) it is not called anywhere in the program; or (2) it is only called by other dead procedures. We define this notion of dead code as invocation dead, or I-dead. In a later section, we show cases that fit that definition of dead procedure while satisfying neither (1) nor (2). Project analyzer defines a live procedure as the opposite of a dead procedure. Further, it always considers an Event Sub (e.g., the procedure associated with a control's mouse click event) as live, on the grounds that at design-time one cannot determine whether an event will occur at run-time. We suggest that augmenting conventional static analysis with semantic analysis could modify that conclusion as discussed below.

Project	Baselined AFJCME Project summary	Modified AFJCME_Lite Project summary
Files		
Total files	37	39
Source files	14	15
File types		
Modules	6	6
Forms	8	9
Resource Files	1	1
Libraries	4	4
Binary Property Files	5	5
Referenced Files	11	12
Project Workspaces	1	1
Project Files	1	1
Code size		
Lines of code	2277	2636
Lines of comment	1446	1915
Lines of whitespace	695	669
Total source lines	4418	5250
Total source bytes	207 kB	254 kB
Averages		
Source lines per module	246	275
Source bytes per module	14.8 kB	17.0 kB
Source lines per procedure	32	31
Source bytes per procedure	1.9 kB	1.9 kB
Number of identifiers		
Total number of identifiers	679	795
Forms and controls		
Forms	8	9
Controls	104	130
Procedures		
Procedures	119	144
- Dead procedures	21	32
Procedures, Basic	107	132
Procedures, DLL	12	12
Procedures, Global	42	49
Procedures, Private	77	95
Subs	99	121
Functions	20	23
Variables and constants		
Total variables + constants	391 + 48 = 439	454 + 49 = 503
- Dead variables + constants	60 + 28 = 88	85 + 27 = 112
Global/module-level variables + constants	36 + 46 = 82	33 + 46 = 79
- Global variables	22	25
- Global constants	34	34
- Module-level variables	14	8
- Module-level constants	12	12
Procedure-level variables and constants	357	424
- Procedure-level variables	277	341
- Procedure parameters	78	80
- Procedure-level constants	2	3

Table 1. Results of applying traditional static analysis to AFJCME

Table 1 presents a subset of the results of applying Project Analyzer to the original baseline AFJCME tool, and the reduced (AFJCME_Lite) version. Of particular interest are the comparisons of source program size as measured in lines of code, total count of live and dead procedures, and count of live and dead variables.

First we note that AFJCME_Lite, while a reduced version of AFJCME, is nonetheless appreciably larger: from 2277 to 2636 lines of code and from 119 to 144 procedures. This results from the way in which many of the modifications were implemented – controls and routines were added to support different modes of operation without first removing previously existing controls and routines. In many cases obsolete/superseded code was rendered inaccessible to the user by means of making its invoking control invisible or by changing a form's geometry (i.e., its attributes of size, shape, position, etc.).

The count of invocation dead procedures is seen to have risen from 21 to 32. This represents the dead code that is detectable by traditional static analysis. It does not include procedures that, while present and associated with controls in the project, are nonetheless inaccessible to the user at run-time. Similar effects are seen in the variable counts.

The results show that the traditional static analysis techniques we applied were useful in detecting much of the dead code that was injected in the specialized version as a result of numerous changes. However, as mentioned previously, we encountered several maintenance examples where we modified certain semantic properties of some objects that caused the underlying code to become dead in the sense of being inaccessible at run-time. The traditional dead code algorithm used in the Project Analyzer tool does not flag such code. That is because traditional static code analysis techniques do not take semantic information about component properties into account. This experience prompted us to examine component-based systems more closely and to look for more efficient ways to analyze them.

3.4. Extending traditional static analysis techniques for component-based solutions

Currently, components that are used to construct a component-based system do not typically include the source code or any additional documentation that describes the component at any length. This is especially true for COTS components. This makes the analysis of these systems difficult at best and in the previous section, we have shown some simple examples to illustrate this. We feel that traditional static analysis techniques can be augmented in a number of ways to analyze component-based systems.

Today, without component source, often no information is available about the component that would be of use to static analysis techniques. For example, variable def-use, first-use/last-use, global reference and definition, and control flow information is not available. In some components, formal parameter information is available in an external source (i.e., IDL file), but this external information is not being utilized in most of the techniques available today. This means that techniques such as coupling analysis that map actual to formal parameter information would fail. In general, because of the lack of insight into components either by having no source code or not utilizing the external IDL files, many of the current static analysis techniques would fail or provide conservative results that would not be useful.

The key is in leveraging the amount of information that is available about the components and incorporating that information in such a way as to provide value to existing static analysis techniques to allow them to analyze the component-based system more precisely. As discussed previously, we distinguish between minimal and extended information that is provided by the component developer.

We outline three approaches that can be used to collect more static analysis insight about components. This insight can then be used to augment many of the traditional static analysis techniques mentioned previously to analyze a component-based system.

The first approach is the as-is approach. This states that we do nothing to improve the insight of a component and use any techniques as-is. This obviously is not the best choice because it results in no improvement in any of the analysis. However, this also

means no additional work is necessary. The results of any analysis will be conservative at best, but for some uses this might be all that is necessary. A good example of this is if a user is simply interested in obtaining a library interface report that just lists the components that a system is using. The lack of additional static analysis information will not have an effect. In effect, this approach supports limited static analysis techniques.

The second approach is to leverage the minimal component provider information that is available for a component to gain a little more insight. 'Minimal component provider information' means that the source code of the components is typically not available, but that standard interface information (e.g., type libraries for COM components) and possibly some documentation is. Systems in this category might be considered the 'legacy' systems of the component-based domain. In this case, as much information as possible should be gleaned from the IDL for the components and used to extend traditional analysis algorithms. For example, we know that from IDL we can observe the intended public interface of the component, showing the methods, parameter passing, and exception handling options. A potential use of this information is to summarize it in some fashion and relate it to the control dependence and data usage information obtained from analysis of the rest of the system. Of course, this may mean making conservative approximations about the analysis relations that hold in the components and about analysis relations caused in the user application by the code in the components.

The risk with this approach is that such approximations can cause the analysis results to be too imprecise to be useful. However, a conservative answer may often be better than no answer. Subject matter experts familiar with the system can then decide on which information that is available from the type libraries and incorporate that into any existing tools and techniques they are using. With this approach, the user still does not gain insight into component and variable usage information, but the availability of the parameter and method information can be used to support techniques such as call graph coupling, basic usage pattern analysis, and dead code detection. This last technique is important because it can be used to reduce the number of nodes in a call and control flow graph, which means that it potentially reduces the testing requirements as well. A better

solution is to identify the necessary additional information that may be needed to support more precise analysis techniques and obtain buy-in from the component developers to either extend the IDL with this additional information or to provide other means to make it available to users. In effect, this is the extended component provider information classification.

The third approach is to have component developers provide extended static analysis summary information along with their component when it is distributed. 'Extended component provider information' means that the component provider does not provide source code, but does provide an extended interface and possibly documentation. This extended interface not only includes the standard interface information, but other information useful for gaining insight into the component without having access to the source code (e.g., an API for summary static analysis information). In this approach, the component provider tests the components and, using analysis techniques, gathers summary information that facilitates further analysis and testing of those components by users without requiring access to the source code. The component provider then makes the summary information available with the component. The component user subsequently integrates the components with the user application, and queries the summary information to drive the analysis and testing of the integrated system. The summary information obviates the need for access to the components' source code. Typical summary information which would be useful for static analysis would be extended call graphs, global variable reference and definition analysis, variable first-use/last-use information, mappings between input parameters, states of the component, and output parameters, as well as a list of the exceptions that components can raise. This information would be useful for many techniques, such as coupling analysis and testing, interface level slicing, integration testing, and the detection of usage patterns and subsequent use of component wrappers. It is important to note that the success of this approach depends on component developers being able to generate this information with little effort. It is envisioned that an analysis tool such as the one described later in this thesis could be provided to component developers so that they can generate the necessary summary information. Then, the component user can use a similar tool to merge the

summary information into an integrated system view for analysis. This is basically the technique we describe later.

In the next section, we discuss a new technique based on the second approach discussed above to analyze a component-based system by using type library/IDL information to gain insight into component properties. The technique is then used to significantly enhance a dead code detection algorithm. In a later section, we discuss a new technique that is based on the third approach discussed above. This technique defines a standard format for static analysis information to be provided by component developers. An analysis tool is then modified to collect this summary information and generate an extended call graph in eXtensible Markup Language (XML) [6]. Such a graph is generated for each component used in a system, as well as the main application. The various graphs are merged into one integrated system view, and from their subsequent static analysis techniques can be applied. The effectiveness of both of these techniques was then validated by applying them to several real COTS component-based systems.

4. AUTOMATED ANALYSIS TECHNIQUES BASED ON COMPONENT PROPERTIES

This section describes automated analysis techniques based on component property information. We develop and discuss a technique to use semantic information about component properties obtained from type library and interface definition language files, and demonstrate the effectiveness of this technique by extending a traditional unreachable code algorithm.

4.1. Using component information that is typically available today

Previously we discussed the importance of static analysis in program understanding, debugging and testing. We also discussed reasons why traditional techniques may not be sufficient for analyzing component-based systems. Referring to our previous classifications of component-based systems, we feel that traditional static analysis techniques can be extended to analyze both average (minimal component provided information) and better (extended component provided information) classifications.

The technique described here is an approach to extend traditional static analysis techniques on systems with 'minimal component provider information' available, which is the typical component-based system found today. 'Minimal component provider information' means that the source code of the components is not available, but that standard interface (e.g., type libraries for COM components) information and possibly some documentation is. Systems in this category might be considered the 'legacy' systems of the component-based domain. In this case, information should be gleaned from the IDL for the components and used to extend traditional analysis algorithms. For example, we know that from IDL we can observe the intended public interface of the component showing the methods, parameter passing, and exception handling options. A

potential use of this information is to summarize it and relate it to the control dependence and data usage information obtained from analysis of the rest of the system. Of course, this may mean making conservative approximations about the analysis relations that hold in the components and about analysis relations caused in the user application by the code in the components. The risk with this approach is that such approximations can cause the analysis results to be too imprecise to be useful. However, a conservative answer may often be better than no answer.

As mentioned, a better solution is to identify the necessary additional information that may be needed to support more precise analysis techniques and obtain buy-in from the component developers to either extend the IDL with this additional information or provide other means to make it available to users. A technique based on this concept is described in the next section.

Using this available documentation, the approach described here then entails using a subject matter expert on the component-based system to help identify several key pieces of semantic information associated with the component-based architecture which would help to promote a better understanding of the overall system or to enhance a particular analysis capability.

4.2. Obtaining useful component IDL information

An important part of this approach is to attempt to leverage the information available from the type library or IDL associated with a component and to use that information to examine closely the interfaces of that component.

To illustrate, consider the TestC example used previously that depicted a component-based application with a TestC.bas application and the two components TestC1 and TestC2. Figure 10 lists the IDL for the TestC1 component obtained using the Microsoft OLE Viewer tool that comes with the Microsoft Visual Studio developer's suite [4]. The IDL provides information about the methods and properties for the component, in this case the method *a* and the properties *j* and *i*.

```
// Generated .IDL file (by the OLE/COM Object Viewer)
// typelib filename: TestCompl.exe
 [
  uuid(BE08C865-6AF7-11D4-8099-00A0CCE27EBB),
  version(1.0),
  helpstring("TestCompl"),
  custom(50867B00-BB69-11D0-A8FF-00A0C9110059, 8495)
]
library TestCompl
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
    importlib("stdole2.tlb");
    // Forward declare all types defined in this typelib
    interface _TestC1;
    [
      odl,
      uuid(BE08C866-6AF7-11D4-8099-00A0CCE27EBB),
      version(1.0),
      hidden,
      dual,
      nonextensible,
      oleautomation
    ]
    interface _TestC1 : IDispatch {
        [id(0x60030002)]
        HRESULT a(
                        [in, out] short* x,
                        [in] short y);
        [id(0x68030001), propget]
        HRESULT j([out, retval] VARIANT* );
        [id(0x68030001), propput]
        HRESULT j([in] VARIANT );
        [id(0x68030000), propget]
        HRESULT i([out, retval] VARIANT* );
        [id(0x68030000), propput]
        HRESULT i([in] VARIANT );
    };
    [
      uuid(BE08C867-6AF7-11D4-8099-00A0CCE27EBB),
      version(1.0)
    ]
    coclass TestC1 {
        [default] interface _TestC1;
    };
};
```

Figure 10. Interface definition language for testC1

The OLE viewer is a good tool for obtaining the IDL from a type library of a given component [5]. This information is used by many development tools, such as MS Visual Studio, to provide developers with object browsing and auto-completion capabilities. However, as can be seen, the IDL is not very human friendly. There are alternatives to getting at this information. The method used for this research is a freeware custom-developed Visual Basic tool, called the ActiveX Documenter [1]. Figure 11 shows an example of the output from this tool. The result is a more human-friendly view of the type library information for the TestC1 component. The methods and properties with their associated parameter information are clearly articulated.

```
TestCompl Interface Definition
General Information
Library: TestCompl (TestCompl)
File:    D:\work\PhD\Test and Validation\PhD_Example\TraditionalComponent
Example\Step 2-Using IDL\TestCompl.exe
GUID:    {BE08C865-6AF7-11D4-8099-00A0CCE27EBB}
Version: 1.0
Enumerations
This section lists enumerations exposed by TestCompl.

Interfaces
This section lists the Classes exposed by TestCompl.  For each class, the methods
and events are listed.

TestCl {BE08C867-6AF7-11D4-8099-00A0CCE27EBB}

Methods
Sub a(ByVal x As Integer, ByVal y As Integer)

Property Get j() As Variant

Property Let j(RHS  As Variant)

Property Get i() As Variant

Property Let i(RHS  As Variant)

Events
None
```

Figure 11. Type library documentation

This information can be used as an aid to component users and developers alike. For example, it provides additional documentation that can be used by the subject matter expert in deciding on the important component properties and criteria for augmenting specific techniques. In section 4.3, we discuss an example of using this information to select key component properties for which the semantic information about those properties can be analyzed to enhance the detection of unreachable code. Later, in section 5, we show how to take this information and incorporate it into the analysis of a system to improve the generation of system-wide call graphs.

4.3. An example of augmenting a static analysis technique

In this research, the test cases we used were based on the Microsoft Component Object Model (COM) technology. The primary development tool and glue code for these systems is Microsoft Visual Basic. Programming languages like Visual Basic have added a whole new dimension to static analysis of program source code. With traditional

structured languages, it was difficult to design analyses that could predict the nature of run-time behavior without significant and program-specific parsing and queries. The object-based nature of Visual Basic-type languages include object attributes that in some cases dictate how the program will behave or how a user may proceed through the program when it is executed. For example, Visual Basic attributes like *visible* and *enabled* hint at possible run-time execution path restrictions based on statically coded attributes. This is to say that the code associated with an invisible program control cannot be executed. With some analysis of the visible attribute, it may be possible to determine if the control's code is ever reachable. Indeed, this object-based property allows a general analysis of the object visible attributes regardless of program or application. The most obvious use of this general attribute analysis is to extend the search for invocation-dead code. We define the term object-attribute dead, or OA-Dead to represent this form of code.

4.3.1. OA-dead code analysis

Using knowledge of COM objects that we obtained from the component IDL information, we selected several key criteria for semantic information that may be exploited. Visual Basic attributes that may affect an object's availability are enabled, visible, top, left, width, and height. If an object is invisible or disabled, the object is OA-dead. Likewise, the top, left, width, and height attributes set correctly (perhaps incorrectly), will also make an object invisible and therefore unavailable, thus OA-dead. If the unavailability of an object persists throughout run-time regardless of program flow, the object and its associated code are OA-dead.

As defined previously, invocation dead code in a program is code that cannot be executed because it cannot be reached through normal program control flow. A program's syntax may render some of its code dead. In VB-type languages, code may be unreachable due to object attribute constraints. In this case, it is semantic constraints that caused the code to be dead. For example, a VB event procedure that executes code following an event on a VB control is not reachable (executable) if that control is never available in the VB program. By available, we mean that the control is visible and

enabled for user input. Invocation-dead code analyses typically do not check for these object attributes and their effect on code execution.

To differentiate between dead code (syntax) analysis and the search for unexecutable code associated with object attributes (semantics), we use the term 'OA-dead' to refer to code that is dead due to semantic properties. An object that is OA-dead is one whose attributes or semantics make it unreachable at any point during run-time.

The importance of identifying OA-dead code has the same importance as identifying invocation-dead code. In software development and maintenance environments, the utility of identifying dead code is well understood. Identifying OA-dead code has added utility in software development in that it may help to predict the software's behavior at run-time. One of the difficulties in predicting run-time behavior is predicting user input. Identifying a control that is OA-dead shows a program path that will never be traversed. Given that controls are designed for user input, an unavailable control limits the possible run-time paths. Verification of the limitation may be the goal. Perhaps, however, the limit was not intended and identifying it allows early correction of a software bug.

During software maintenance, the utility of OA-dead analysis increases dramatically because a project's maintainers are often different from its developers. Again, the ability to determine and/or verify code paths is of great importance. Maintainers may actually disable a control and analyze the resulting OA-dead code to determine a project's control flow. Understanding the impacts of these types of semantic properties is extremely useful.

4.3.2. Results of applying OA-dead analysis to AFJCME_Lite

To illustrate further the criteria chosen we continue with the OA-dead analysis of the AFJCME_Lite program discussed previously. We describe in detail the maintenance issue pertaining to component object events that have been made inaccessible by modifying object semantics within the Visual Basic development environment. These examples show the important role semantic information can play in the static analysis of a program.

The routine depicted in Figure 12 is from the specialized version of the AFJCME_Lite tool. This code is attached to a click event of a command button object, named cmdMoveUp, located on a form named frmSessions.

```
Private Sub cmdMoveUp_Click()
    'When user wishes to reorder the system list for a session by moving a given
system
    'up in the list:
    '    - Reorder right pane systems list (lstSystemSelect) for this session by
moving
    '      highlighted system up one position in the list.
    '    - Set this system's  dirty flag to 1; means queries will have to be rebuilt.
    Dim moverSysName As String, moveeSysName As String
    Dim moverSysNo As String, moveeSysNo As String
    Dim targetListIndex As Integer

    With lstSystemSelect
        'Don't move unless it's  a selected (checked) system and it
        'isn't already first in the list.
        If (.ListIndex > 0) And (.Selected(.ListIndex)) Then
            moverSysName = .List(.ListIndex)
            moveeSysName = .List(.ListIndex - 1)
            moverSysNo = .ItemData(.ListIndex)
            moveeSysNo = .ItemData(.ListIndex - 1)
            targetListIndex = .ListIndex - 1
            Call swapEm(moverSysNo, moveeSysNo, targetListIndex)
            's et the dirty flag for this session to 1; no need to check for no
systems
            's elected because we wouldn't have gotten here if that were the case.
            lstSessionSelect.ItemData(lstSessionSelect.ListIndex) = 1
        End If
    End With
End Sub
```

Figure 12. Source code for click event of cmdMoveUp

It is not a trivial matter to determine whether this code can/will ever be executed when the tool is in operation. The code will run only in response to a mouse click event issued to the form element named cmdMoveUp. The user may be prohibited from causing that event to take place in several ways including, for example, program logic that would prohibit the form from ever being displayed. This condition is in the realm of standard analysis. However, other conditions are more specific to the class of visual programming tools and component-based maintenance under consideration here. Two examples are: 1) The cmdMoveUp object might be invisible; and 2) The cmdMoveUp object might be inaccessible altogether. Each of these conditions can, in turn, come about in more than one way, because the properties of objects can be set at design time, and/or modified at run time.

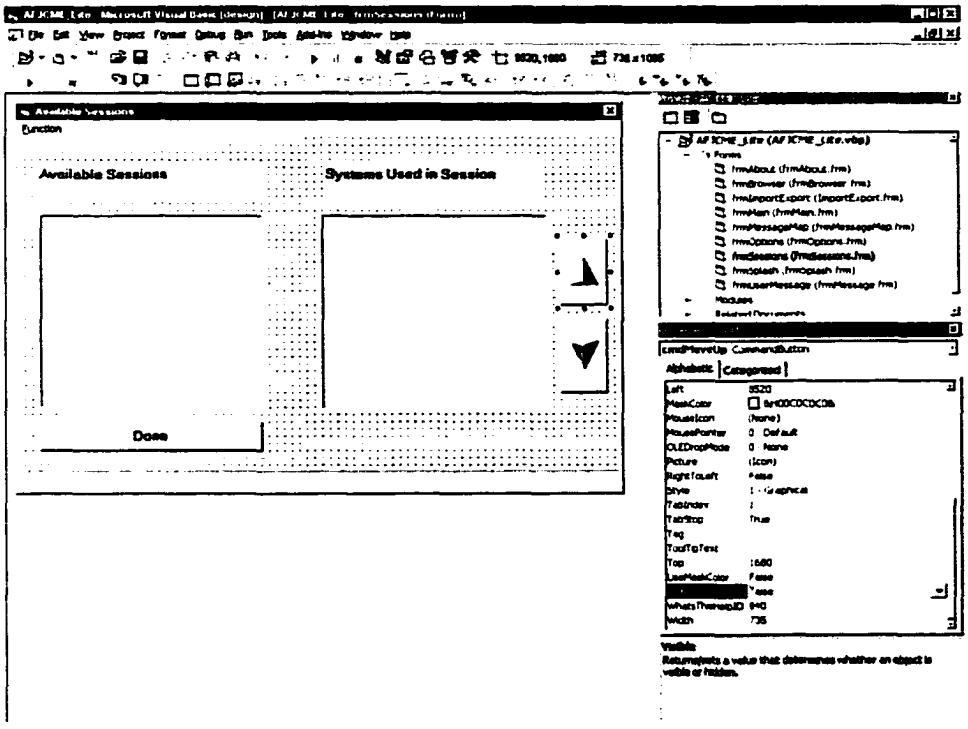

Figure 13. MS Visual Basic 5 development environment

It is not clear from an examination of the code when either of these conditions applies. For the (static) case of properties set at design time, examination of the properties would be required. To illustrate the first example, consider Figure 13. Here we see a view of the MS Visual Basic 5 development environment with the form frmSessions open in design-mode. The command button, cmdMoveUp, is selected and a property window is open to allow the semantic values of the properties for the button to be modified.

At this point suppose we set the visible property to false in order to hide the command button during run-time. This type of modification actually occurred on numerous occasions because the maintainer was under a time constraint and was uncomfortable with making extensive changes that may have had other side-effects (i.e., a typical maintenance patch).

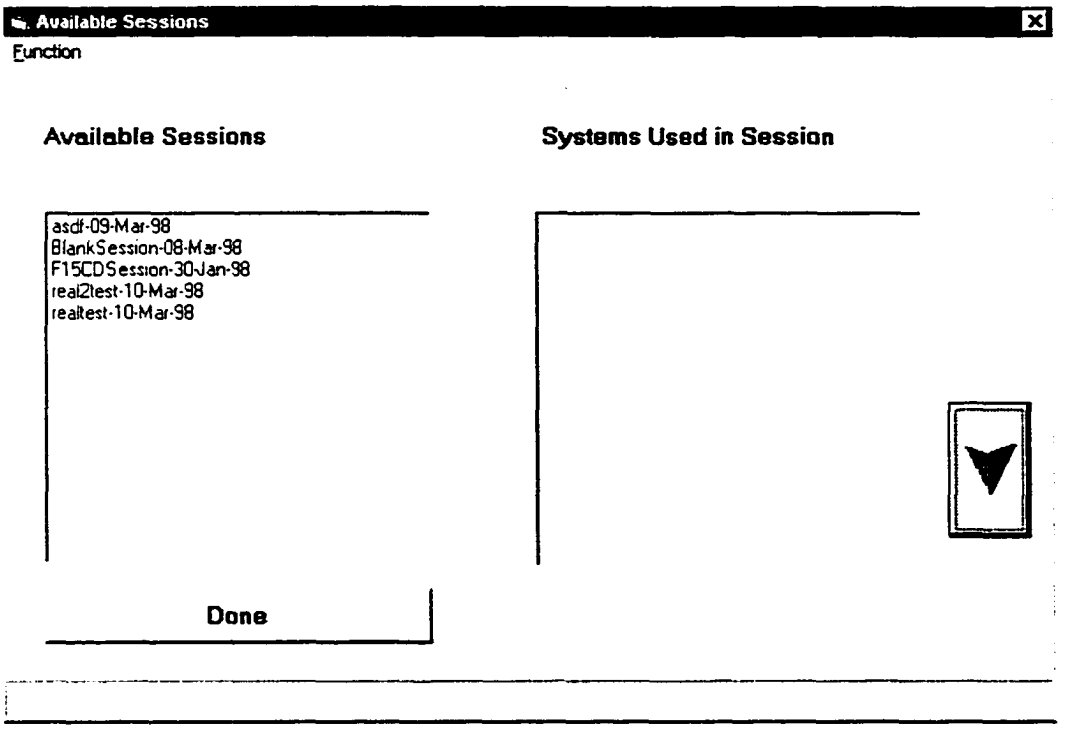

Figure 14. Modified form without the cmdMoveUp button

Figure 14 shows the runtime result of having set cmdMoveUp.Visible to false. We see that the control is not displayed, and therefore cannot receive a click event. For an analysis tool to detect this would require that it examine object properties based on specific criteria to address these conditions, or to examine the object properties and then draw inferences from the results. The inferencing engine could apply a number of artificial intelligence techniques that go beyond the scope of this research. For example, examination of the form file frmSessions.frm, shows the results in Figure 15. As expected, the Visible property is set to 0 (false).

```
Begin VB.CommandButton cmdMoveUp
   Height          =   1095
   Left            =   8520
   Picture         =   "frmSessions.frx":044A
   Style           =   1  'Graphical
   TabIndex        =   1
   Top             =   1680
   Visible         =   0   'False
   WhatsThisHelpID =   840
   Width           =   735
End
```

Figure 15. Source code from frmSessions

A much more difficult analysis problem would arise in the second example, where the dimensions of the form have been changed so that the portion of the form containing the control is hidden at runtime. Figure 17 shows the effect at runtime of shrinking the dimensions of the form, rendering the cmdMoveUp button inaccessible (hidden). The semantic information to discover this is available as the following extract from frmSessions.frm shows in Figure 16:

```
Begin VB.Form frmSessions
   BorderStyle     =   1  'Fixed Single
   Caption         =   "Available Sessions"
   ClientHeight    =   5850
   ClientLeft      =   30
   ClientTop       =   645
   ClientWidth     =   4410
   HelpContextID   =   840
   LinkTopic       =   "Form1"
   MaxButton       =   0   'False
   MinButton       =   0   'False
   ScaleHeight     =   5850
   ScaleWidth      =   4410
   Tag             =   "Activate a session or change participating systems"
```

Figure 16. Semantic information from frmsessions

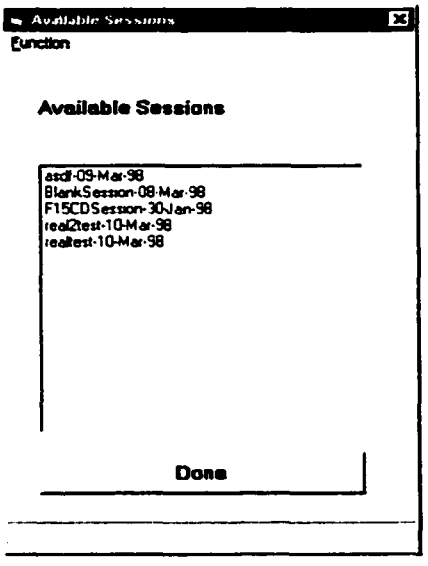

Figure 17. Modified form after resize

The analysis would have to include recognizing the fact that setting frmSessions.ScaleWidth to 4410 precludes the control named cmdMoveUp from being accessed at run time. For example, we know that the original frmSessions.ScaleWidth was 9960 before the resize. We also know that the button, cmdMoveUp, has a container left-hand starting position of cmdMoveUp.Left = 8520 and a width of 735. Comparing the modified frmSessions.ScaleWidth of 4410 shows that the right-hand side of the form is less than the starting position of the button (8520), meaning that the button is now hidden.

To demonstrate this technique in our research, we modified Project Analyzer with extended algorithms to discover and analyze each of the above criteria as part of an OA-dead analysis report [113]. Details of the OA-dead extensions made to the Project Analyzer tool are discussed in a later section.

Metric	AFJCME_Lite
PROGRAM SUMMARY	
Total SLOC:	5250
Code Size in kB:	254
Components:	25
Total Procedures:	144
Dead Procedures:	32
OA-DEAD ANALYSIS	
OA-Dead Files:	2
OA-Dead Procedures:	43
Reasons:	
Event of OA-Dead Control:	32 (74.42%)
Called by OA-Dead Procedure:	9 (20.93%)
In OA-Dead File:	2 (4.65%)
OA-Dead Controls:	72
Reasons:	
Disabled:	3 (4.17%)
Invisible:	17 (23.61%)
Too Narrow:	52 (72.22%)
Too Short:	52 (72.22%)
Too Far Right:	0 (0%)
Too Far Down:	0 (0%)
Too Far Left:	3 (4.17%)
Too Far Up:	0 (0%)
In OA-Dead File:	13 (18.06%)

Table 2. Results of applying OA-dead analysis on AFJCME_Lite

Table 2 represents the results from running the OA-dead code detection algorithm on the AFJCME_Lite DoD system we discussed previously. A significant number of new dead procedures (i.e., OA-dead) has been detected as a result of taking into account the semantic information of object properties. An additional 43 procedures were identified as OA-dead, as well as 72 controls and 2 files. The significance of this is that the code related to the 43 procedures identified as OA-dead could represent a significant reduction in the overall amount of code that needs to be analyzed, maintained or tested. For example, the test cases associated with the OA-dead procedures may not have to be run, which may represent a significant reduction in the amount of testing to be accomplished. Such high numbers on a 5250 line program also indicate that typical maintenance activities still have the potential to induce complexities in a system irrespective of whether that system is component-based. Of the OA-dead procedures, it is interesting to point out that over 74% were events to OA-dead controls, of which over 72% of those were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. This implies that many of the modifications made to the program could have been quick fixes. For example, as an alternative to commenting out code for which the maintainer was not quite sure was needed anywhere else, we observed that they just resize a form or a control to hide it from the user's view.

Although this is just one example, a more exhaustive examination of the results of using this technique on several case studies is discussed in a later section. However, this example does support that the OA-dead analysis report appears to be an effective tool for understanding particular usage patterns and maintenance actions pertaining to a particular system. It also validates the effectiveness of the technique for putting component IDL information to good use to augment static analysis techniques for very specific capabilities.

The technique and the OA-dead tool extensions discussed above were developed with a specific application in mind. Unreachable code was determined to be an important characteristic to look for in the DoD systems being maintained. It turned out that the OA-

dead analysis is an effective technique, and that the criteria for component property information that was selected represented a good choice for analyzing the Visual Basic language. In general, it may be costly in terms of resources for organizations to customize a tool for specific purposes. There needs to be better ways to analyze component-based systems which could allow many more of the advanced, mature static analysis techniques to be used. The next section describes a technique to help address this.

5. AUTOMATED ANALYSIS TECHNIQUES BASED ON COMPONENT DEVELOPER SUMMARY INFORMATION

This section describes automated analysis techniques based on component developer summary information generated by the component provider during development and testing of the component, and then distributed to the component user. This can then be integrated with several traditional static analysis techniques to analyze component-based systems more precisely. We illustrate the effectiveness of this technique by modifying an existing analysis tool to generate extended call graphs embedded with summary information for global data flow analysis, and show the use of this information in the analysis of a sample system.

5.1. Additional component information that is needed to support static analysis

Using the available IDL information about a component as a documentation aid to the component user or developer, the previous approach showed a way to use this information to support some specific static analysis capabilities. To support a broader set of capabilities, more additional information about component is necessary.

As we have seen, the IDL information provides a list of the public interfaces that a component offers, including methods, properties, events and exceptions. The current form of IDL documentation does not provide any insight into control flow, data flow, component states, or other similar information that would be useful when performing static analysis of a system. For example, the ripple analysis and coupling analysis techniques that we demonstrated previously could not be performed on a component-based system without imprecise results.

We know that traditional static analysis can be performed on a system that is not component-based. In a component-based system, many of the components are in effect black boxes for which little information outside of the IDL is known. This lack of

information about components is a fundamental problem with performing advanced static analysis on such systems. This research addresses this problem by identifying key information that could be summarized about a component through some automated analysis performed on that component by the component developer. A way to represent this information is also suggested using several summary graphs stored in XML. These XML graphs can then be distributed with the components. The component user subsequently integrates the summary graphs for each component along with the graph for the user application to form an integrated system view. Once this integrated system graph has been constructed, traditional static analysis techniques can be applied as before. The summary information obviates the need for access to the source code for a component.

Information	Description	Supports
IDL	Interface Description Language provides additional information about component interfaces, such as methods, properties, events and exceptions that can be exploited in static analysis.	Basic component understanding, specific static analysis techniques, such as dead code detection, basic usage analysis, and testing.
Call and control flow	Call and control flow will provide information on modules, call graph node, called modules, called by modules, detailed call site information, variables and constants, and formal parameters.	Most static analysis techniques, such as: coupling analysis and testing, interface slicing, interface ripple analysis, and integration testing.
Variable usage	Variable usage includes information on ref-def analysis, global analysis, and variable first-use and last-use information.	Most static analysis techniques, such as: coupling analysis and testing, interface slicing, interface ripple analysis, and integration testing.
Parameter mapping dependence	Parameter mapping data dependence provides the relationships between formal parameters and all actual mappings throughout the system.	Coupling analysis and testing.
Impact dependence	Impact dependence information provides the relationships between global referenced and defined variables and the modules of interest that contribute to or use that variable.	Reverse and forward interface-level ripple analysis
Statement-level control flow graph information	Statement-level control flow information	Statement level techniques, such as program slicing, ripple analysis, partial evaluation, and testing
Component states	Identification of the various states a component can be in.	Techniques like: usage patterns, component wrappers, interface slicing

Information	Description	Supports
		and integration testing
Dependence information	Dependence information between input, output and state variables	Techniques like: usage patterns, component wrappers, interface slicing and integration testing
Exception information	Exception information and its dependence to input and state variables	Techniques like: usage patterns, component wrappers, interface slicing and integration testing

Table 3. Static Analysis Summary Information

The information in Table 3 depicts a set of static analysis summary information that could be generated for a component and distributed to a component user to perform many of the techniques mentioned in the third column. The IDL is information available today. The other information would have to be generated as summary information about the component. The call and control flow along with the variable usage information provide the fundamental control and data flow information necessary to support many static analysis techniques. An important distinction should be made between module-level control flow and statement-level control flow as summary information. Statement-level control flow is necessary for performing most types of static analysis. However, we feel that component developers may be reluctant to provide complete statement-level control flow graphs for their components because of proprietary concerns. The reason for this is that the regeneration of program source from control flow graphs is a heavily researched area and many techniques exist for doing this. We also feel that interface-level analysis is more important when dealing with a component-based system than statement-level analysis. The reason is that realistic component-based systems will more than likely contain a hybrid of components for which additional summary information is provided, along with components for which the IDL is the only information available. Another reason is testing. Component-based testing by component users is primarily an integration test that examines the interfaces throughout the system.

The technique that we have implemented here computes call and control flow, variable usage, parameter mapping dependence and impact dependence summary information.

5.2. Representing additional component summary information

Just as important as identifying the summary information that should be collected for each component is an efficient mechanism for representing it and making it available to the component user. For this research, we felt that it was important for the information to be some form of documentation that could be made available to component users. Since we felt this information should be incorporated into a tool for automated analysis, it is also important to use a format that can support this and be easily read by a human if need be. We chose the extensible markup language (XML) for this representation because it represents a standards-based method that meets all of our objectives [6]. Using XML allowed us to separate the data from the presentation of that data such that one data file could easily be rendered into many different forms. This makes it easy to incorporate this information into various tools or human views as necessary. Three graph structures were defined. The summary information call graph is the primary graph structure that is used to contain the call and control flow as well as the variable usage information. A second graph structure is the parameter mapping dependence graph. This is used to store the relationships between formal parameters and actual parameters at all related call sites. The third graph structure is the impact dependence graph that is used for ripple analysis. Each of these is briefly described below.

5.2.1. Summary Information Call Graph

```
<?xml version ="1.0"?>
<!--Generated by XML Authority. Conforms to w3c http://www.w3.org/TR/xmlschema-1/-->
<schema targetNamespace = "CallGraph.xsd"
   xmlns = "http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd">
  <element name = "CallGraph">
         <type content = "elementOnly">
                <group order = "seq">
                       <element ref = "ReportTitle"/>
                       <element ref = "ProjectTitle"/>
                       <group order = "seq" minOccurs = "0" maxOccurs = "*">
                              <element ref = "ModuleCollection"/>
                              <element ref = "CallGraphNode" minOccurs = "1"
maxOccurs = "*"/>
                       </group>
                </group>
         </type>
  </element>
  <element name = "ReportTitle" type = "string"/>
  <element name = "ProjectTitle" type = "string"/>
  <element name = "CallGraphNode">
         <type content = "elementOnly">
```

```
                <group order = "seq">
                        <element ref = "FormalParameters" minOccurs = "0" maxOccurs
= "*"/>
                        <element ref = "ConstantDeclarations" minOccurs = "0"
maxOccurs = "*"/>
                        <element ref = "VariableDeclarations" minOccurs = "0"
maxOccurs = "*"/>
                        <element ref = "CalledModules" minOccurs = "0" maxOccurs =
"*"/>
                        <element ref = "CallSites" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "GlobalRefs" minOccurs = "0" maxOccurs =
"*"/>
                        <element ref = "GlobalDefs" minOccurs = "0" maxOccurs =
"*"/>
                        <element ref = "LocalRefs" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "LocalDefs" minOccurs = "0" maxOccurs = "*"/>
                </group>
                <attribute name = "NodeID" minOccurs = "1" type = "integer"/>
                <attribute name = "ProcName" minOccurs = "1" type = "string"/>
                <attribute name = "ParentModName" minOccurs = "1" type = "string"/>
        </type>
</element>
•
•
•
<element name = "GlobalRefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "ConstRefs" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "VarRefs" minOccurs = "0" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "GlobalDefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "ConstDefs" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "VarDefs" minOccurs = "0" maxOccurs = "*"/>
                </group>
        </type>
</element>
•
•
•
<element name = "FirstUse" type = "string"/>
<element name = "LastUse" type = "string"/>
<element name = "Parameter">
        <type content = "textOnly">
                <attribute name = "VarID" minOccurs = "1" type = "string"/>
                <attribute name = "VarName" minOccurs = "1" type = "string"/>
                <attribute name = "FirstUse" minOccurs = "1">
                        <datatype source = "string">
                                <enumeration value = "ref"/>
                                <enumeration value = "def"/>
                        </datatype>
                </attribute>
                <attribute name = "LastUse" minOccurs = "1">
                        <datatype source = "string">
                                <enumeration value = "ref"/>
                                <enumeration value = "def"/>
                        </datatype>
                </attribute>
        </type>
</element>
<element name = "CallSites">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "CallSite" minOccurs = "1" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "CallSite">
        <type content = "elementOnly">
```

```
                <group order = "seq">
                        <element ref = "Module"/>
                        <element ref = "StatementLineNumber"/>
                        <element ref = "CallSiteAnalysisCompleted"/>
                        <element ref = "ParameterMapping"/>
                </group>
        </type>
</element>
•
•
•
</schema>
```

Figure 18. Portion of XML summary call graph schema

Figure 18 depicts a portion of the XML schema for the summary call graph. The included portions highlight some key aspects of the graph to illustrate our intended representation for the summary information discussed previously. The first point to notice is the hierarchy of the graph. A call graph is comprised of zero or more call graph nodes, where each call graph node is comprised of child elements for formal parameters, constant declarations, variable declarations, called modules, call sites, global and local variable references, global and local variable definitions, as well as three attributes for a unique node identification number, the name of the module, and the name of any parent module. The next point to notice is the storage of the *GlobalRefs* and *GlobalDefs* that contain a list of the constants or variables that have global impacts outside of a particular analysis. The variables in these lists represent the result of a global variable def-use analysis that is conducted on the module and includes the effects of modules calls made from within the module being examined. For example, if a module *a* may call module *b* and inside module *b*, a global variable *i* may be defined, then *i* will also be included in the *GlobalDefs* list for module *a*. This is important for supporting techniques like program slicing and ripple analysis. The next point to notice is the parameter information which includes the first-use and last-use for each formal parameter for the module. This is important for techniques like coupling analysis. The final point to notice is the call site element that contains the actual to formal parameter mappings.

The complete XML summary call graph schema is listed in Appendix C.

5.2.2. Data Dependence Graph

```
<schema targetNamespace = "DataDependenceRpt.xsd"
        xmlns = "http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd">
        <element name = "DataDependenceReport">
                <type content = "mixed">
                        <element ref = "ModuleName"/>
                </type>
        </element>
        <element name = "ModuleName">
                <type content = "elementOnly">
                        <group order = "seq">
                                <element ref = "DataDependence"/>
                        </group>
                        <attribute name = "ID" type = "string"/>
                        <attribute name = "Name" type = "string"/>
                </type>
        </element>
        <element name = "DataDependence">
                <type content = "elementOnly">
                        <group order = "seq">
                                <element ref = "FormalParameter" minOccurs = "0"
maxOccurs = "*"/>
                        </group>
                </type>
        </element>
        <element name = "FormalParameter">
                <type content = "elementOnly">
                        <group order = "seq">
                                <element ref = "Variables" minOccurs = "1" maxOccurs
= "*"/>
                                <element ref = "Constants" minOccurs = "1" maxOccurs
= "*"/>
                        </group>
                        <attribute name = "ID" type = "string"/>
                        <attribute name = "Name" type = "string"/>
                </type>
        </element>
  •
  •
  •
</schema>
```

Figure 19. Portion of XML parameter mapping schema

Figure 19 depicts a portion of the XML schema for the parameter mapping dependence graph. This graph contains one or more module to data dependency pairings. For each module, a data dependence set relates each formal parameter of the module to actual parameters for all call sites to this module throughout the system being analyzed. This graph is a secondary graph that is produced from an analysis of the summary call graph for the system being analyzed. It is also a good example of how large XML documents can be transformed into smaller XML documents designed for a specific purpose.

The complete XML parameter mapping dependence graph schema is listed in Appendix C.

5.2.3. Ripple Analysis Graph

```
<?xml version ="1.0"?>
<!-- Conforms to w3c http://www.w3.org/TR/xmlschema-1/-->
<schema targetNamespace = "ReverseRipple.xsd"
        xmlns = "http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd">
       <element name = "ReverseRipple">
              <type content = "elementOnly">
                     <group order = "seq">
                            <element ref = "CallGraphNode" minOccurs = "1"
maxOccurs = "*"/>
                     </group>
              </type>
       </element>
       <element name = "CallGraphNode">
              <type content = "elementOnly">
                     <group order = "seq">
                            <element ref = "GlobalRefVar" minOccurs = "0"
maxOccurs = "*"/>
                     </group>
                     <attribute name = "ID" type = "string"/>
                     <attribute name = "Name" type = "string"/>
                     <attribute name = "ParentModName" type = "string"/>
              </type>
       </element>
       <element name = "GlobalRefVar">
              <type content = "elementOnly">
                     <group order = "seq">
                            <element ref = "ImpactedBy" minOccurs = "0"
maxOccurs = "*"/>
                     </group>
                     <attribute name = "ID" type = "string"/>
                     <attribute name = "Name" type = "string"/>
              </type>
       </element>
       <element name = "ImpactedBy">
              <type content = "elementOnly">
                     <group order = "seq">
                            <element ref = "Module"/>
                            <element ref = "Variable"/>
                     </group>
              </type>
       </element>•
  •
  •
  •
</schema>
```

Figure 20. Portion of XML ripple analysis schema

Figure 20 depicts a portion of the XML schema for the ripple analysis graph. This particular schema illustrates the reverse ripple analysis. A similar graph exists for a forward ripple analysis. Recalling a previous discussion, a reverse ripple looks at each global variable reference in a module, and examines the calling hierarchy backwards looking for all definitions to that variable and any additional variables used to define them. To support this, the graph contains one or more call graph nodes, where each node contains an element for each variable in the *GlobalRefs* list for that node. Then, for each

of these variables, an *ImpactedBy* set is created which includes the module and variable where a particular definition has occurred in the calling path. A variable is included in the impact set to account for formal ByRef parameters that need to be mapped to actual parameters to complete the analysis.

The complete XML ripple analysis graph schema is listed in Appendix C.

```
<CallGraphNode NodeID="4" ModuleName="Main" ParentModName="Module1">
   <CalledModules>
      <Module ModuleName="b" ModuleID="3"> </Module>
      <Module ModuleName="a" ModuleID="2"> </Module>
   </CalledModules>
   <CallSites>
      <CallSite>
         <Module ModuleName="b" ModuleID="3" InModuleCollection="Module1"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>
            <ActualParameter VarName="x" VarID="1"> </ActualParameter>
            <PassByVal>
               <Parameter VarName="x" VarID="9" FirstUse="" LastUse="">
</Parameter>
            </PassByVal>
            <ActualParameter VarName="y" VarID="2"> </ActualParameter>
            <PassByVal>
               <Parameter VarName="y" VarID="10" FirstUse="REF" LastUse="REF">
</Parameter>
            </PassByVal>
         </ParameterMapping>
      </CallSite>
      <CallSite>
         <Module ModuleName="a" ModuleID="2" InModuleCollection="Module1"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>
            <ActualParameter VarName="x" VarID="1"> </ActualParameter>
            <PassByRef>
               <Parameter VarName="x" VarID="5" FirstUse="REF" LastUse="DEF">
</Parameter>
            </PassByRef>
            <ActualParameter VarName="y" VarID="2"> </ActualParameter>
            <PassByVal>
               <Parameter VarName="y" VarID="6" FirstUse="REF" LastUse="REF">
</Parameter>
            </PassByVal>
         </ParameterMapping>
      </CallSite>
   </CallSites>
   <GlobalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="j" VarID="4"> </Variable>
      <Variable VarName="i" VarID="3"> </Variable>
      <Variable VarName="y" VarID="2"> </Variable>
      <Variable VarName="x" VarID="1"> </Variable>
   </VarRefs>
   </GlobalRefs>
   <GlobalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="j" VarID="4"> </Variable>
      <Variable VarName="i" VarID="3"> </Variable>
      <Variable VarName="y" VarID="2"> </Variable>
      <Variable VarName="x" VarID="1"> </Variable>
   </VarDefs>
```

```
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
```

Figure 21. Portion of XML summary call graph for test.bas

5.3. A tool for generating component summary information graphs

Now that we have defined the summary information to be computed and a representation for this information, the next step is to provide a tool that constructs the graphs and then has a way to merge graphs from multiple components into an integrated system view. Once this integrated system view is reached, then various static analysis techniques can be applied on the component-based system for more precise results. In effect, this technique neutralizes the black box aspect of components with respect to having no source code available for analysis.

For this research, several tools were developed or modified to support this technique. In particular, we modified the Project Analyzer tool discussed in Section 3.3 with extensions to:

- Perform additional analysis of a system to construct a summary call graph in accordance with the XML schema discussed above. The additional analysis includes among other information a global variable ref-def analysis, variable first-use and last-use analysis, and an actual to formal parameter mapping.

- Provide the ability to merge multiple XML summary call graphs into an integrated system view.

- Provide additional analysis capabilities for supporting parameter mapping dependence analysis and reverse and forward interface ripple analysis.

Figure 21 depicts a portion of the XML summary graph for the Test.bas example used previously. Once the various graphs are generated and saved in XML, they can be

used in a number of ways. The obvious use is to load them back into the extended Project Analyzer and merge them into an integrated system graph that can be analyzed and saved in XML as a new graph. However, since we chose XML for our format, we are able to leverage the many commercially and publicly available tools for processing such files. Using the extensible stylesheet language (XSL), for example, we constructed several scripts to perform additional analysis on the graphs and render the following views:

- Tabular display of a summary call graph
- Call graph metrics report
- Coupling analysis report
- Parameter mapping dependency report
- Ripple analysis report

Each of these tools is described in more detail in a later section. In addition, another tool described previously is used for documenting the type library information that is available for components. Below, we show the importance of using this tool to allow our extended Project Analyzer's analysis to be more precise when constructing call graphs and in performing the parameter dependency analysis.

5.4. Evolving the summary call graph

As we have seen, many static analysis techniques use call and control flow graphs during their analysis. This work is based on the development of a summary call graph. For this reason, it is important to have a graph that is as precise as possible. This is particularly important when dealing with distributed components, as we will now illustrate.

To produce a more precise summary call graph for a system with distributed components, it is necessary to ensure that the IDL information that can be obtained by using the type library tool mentioned above is integrated appropriately into the analysis

tool. To illustrate this, we define three stages for the evolution of a summary call graph and refer back to the TestC component example used previously.

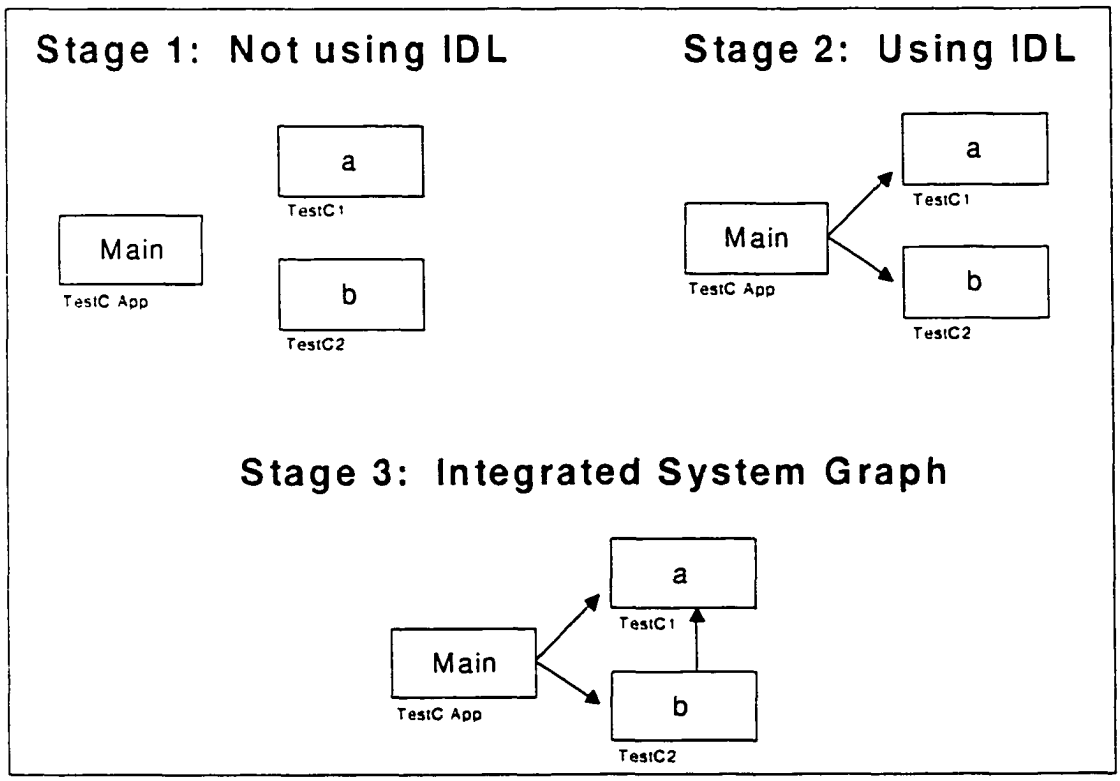

Figure 22. *Stages of summary call graph evolution*

Recall from Figure 7 that the example had a TestC.bas user application and two out-of-process components, TestC1.exe and TestC2.exe. Component TestC1 had interfaces for a method *a* and properties *i* and *j*. Component TestC2 had an interface for method *b*, which called method *a* in TestC1 by the way. The main application, TestC.bas, had calls to interfaces in both components.

In Figure 22, stage 1 represents the initial state of analysis if IDL information is not taken into account. Using Project Analyzer on TestC, TestC1, and TestC2 to generate summary call graphs for each will result in the calling sequence shown in stage one, namely the graph does not show any paths from the main application to either component. Several reasons may explain why this might happen. One is that the developers for TestC

and TestC2 may not have referenced the dependent components in their development tool. In the case of Visual Basic, it will provide a warning to the developer when loading source that is dependent on external components. However, Visual Basic will not enforce this, so a developer may by-pass the warning and continue development thinking they will add the reference at a later time. If they forget, the reference may not be placed. Even if the component is registered, the other side effect here is if a new version of a component gets registered, the referenced component is no longer available and the reference is classified as missing. A more common reason for the missing call paths is if the developer did not declare calls to external methods and properties in the code. Visual Basic, for example, is very forgiving in this case and does not require developer's to do this as long as the component is referenced. The problem is if this is not enforced, then no indication exists in the source or any other project files that indicates to which component libraries a particular object refers, and which interfaces are used. As a result, a tool that analyzes source code will be unable to distinguish this information and will effectively ignore the calls in the call graph. This is the case for Project Analyzer.

```
  Attribute VB_Name = "App_Declares"
  Declare Sub a Lib "TestC1" (ByVal x As Integer, ByVal y As Integer)
  Declare Sub b Lib "TestC2" (ByVal Caller As TestC1, ByVal x As Integer, ByVal y As
Integer)
  Declare Function MsgBox Lib "VBA" (ByVal Buttons As VbMsgBoxStyle, Optional ByVal
Title As String, Optional ByVal HelpFile As String, Optional ByVal Context As
String) As VbMsgBoxResult
  Public Property Get j() As Variant
  End Property
  Public Property Let j(RHS As Variant)
  End Property
  Public Property Get i() As Variant
  End Property
  Public Property Let i(RHS As Variant)
  End Property
```

Figure 23. Declaration module for TestC application

```
  Declare Sub a Lib "TestC1" (ByVal x As Integer, ByVal y As Integer)
  Public Property Get j() As Variant
  End Property
  Public Property Let j(RHS As Variant)
  End Property
  Public Property Get i() As Variant
  End Property
  Public Property Let i(RHS As Variant)
  End Property
```

Figure 24. Declarations module for TestC2 component

To account for this, a component developer or user should include the appropriate declarations in the source. To help obtain this information from the IDL for a component, we show how to use the type library documentation tool to obtain the information and convert it into an appropriate declaration file for use with Project Analyzer.

Recall from Figure 11 the type library information obtained for TestC1 to show its available interfaces. From the previous discussion, we know that both the TestC main user application and the TestC2 component call interfaces in component TestC2. To avoid the call graph problem shown in stage 1 of Figure 22, both the TestC2 component developer and the component user should include the proper declarations for TestC1 in their analysis. This can be done by converting the type library information in Figure 11 to the appropriate declaration modules shown in Figure 23 and Figure 24 for the component user and TestC2 component developer respectively. After doing this and running the Project Analyzer analysis again, the result is stage 2 in Figure 22. In stage 2, separate summary call graphs have been generated for the main TestC application and each of the components TestC1 and TestC2. Because the appropriate declaration modules have been used, each summary graph is more precise in that the individual call graphs now show calls to the external interfaces. However, unless the individual summary call graphs get sent to the component user and integrated, the component user's system view, shown as stage 2, is still imprecise. For example, because the component user integrated the appropriate declarations, the summary call graph shows the paths to the interfaces in both components. However, if the summary call graph for component TestC2 never gets integrated, then the component user does not see the call path from TestC2 to TestC1.

To improve this situation, all of the summary call graphs should be integrated into one system graph for subsequent analysis. Stage 3 in Figure 22 reflects this where the integrated view now shows a more complete calling path by showing the path from TestC2 to TestC1.

Once the integrated system graph is constructed, it can be used to support many traditional static analysis techniques. For example, recall from our previous discussion

when the TestC example was first introduced that because of not having the source code for the components we could not perform techniques like the coupling analysis or ripple analysis that we did on the original Test.bas program. Table 4 below shows the results of a coupling analysis on the TestC integrated system graph.

CallGraphNode	CallSite	Actual Parameter	Formal Parameter
NodeID="2" ModuleName="Main" ParentModName="Module1"	ModuleName="a" ModuleID="5" InModuleCollection="TestC1"	VarName="x" VarID="1"	PassByRef VarName="x" VarID="7" FirstUse="REF" LastUse="DEF"
NodeID="2" ModuleName="Main" ParentModName="Module1"	ModuleName="a" ModuleID="5" InModuleCollection="TestC1"	VarName="y" VarID="2"	PassByVal VarName="y" VarID="8" FirstUse="REF" LastUse="REF"
NodeID="2" ModuleName="Main" ParentModName="Module1"	ModuleName="b" ModuleID="12" InModuleCollection="TestC2"	VarName="gC1" VarID="3"	PassByRef VarName="Caller" VarID="13" FirstUse="REF" LastUse="REF"
NodeID="2" ModuleName="Main" ParentModName="Module1"	ModuleName="b" ModuleID="12" InModuleCollection="TestC2"	VarName="y" VarID="2"	PassByVal VarName="y" VarID="15" FirstUse="REF" LastUse="REF"
NodeID="12" ModuleName="b" ParentModName="TestC2"	ModuleName="a" ModuleID="5" InModuleCollection="TestC1"	VarName="Caller" VarID="13"	PassByRef VarName="x" VarID="7" FirstUse="REF" LastUse="DEF"

Table 4. Coupling analysis of TestC

5.5. An example of using the component summary information technique

To illustrate this technique a bit further, we examine a component-based system that calculates shipping costs for mail orders [44]. The component diagram for this example is shown in Figure 25.

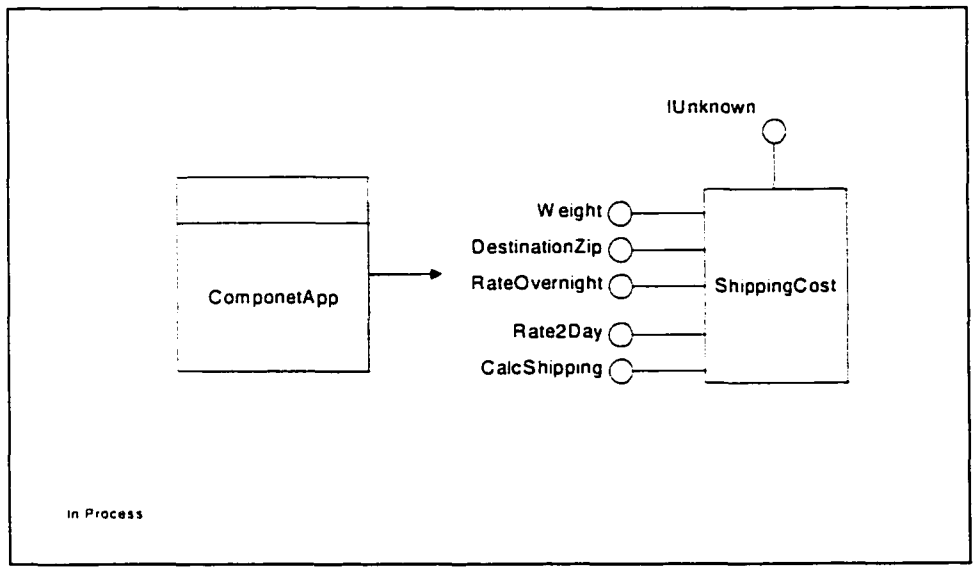

Figure 25. ShippingCost component diagram

A ShippingCost component provides interfaces for the *Weight*, *DestinationZip*, *RateOvernight*, *Rate2Day* properties and the *CalcShipping* method. The component application has three modules, *main*, *a* and *b* each of which communicate with the component. The summary call graph evolution for this system is shown in Figure 26.

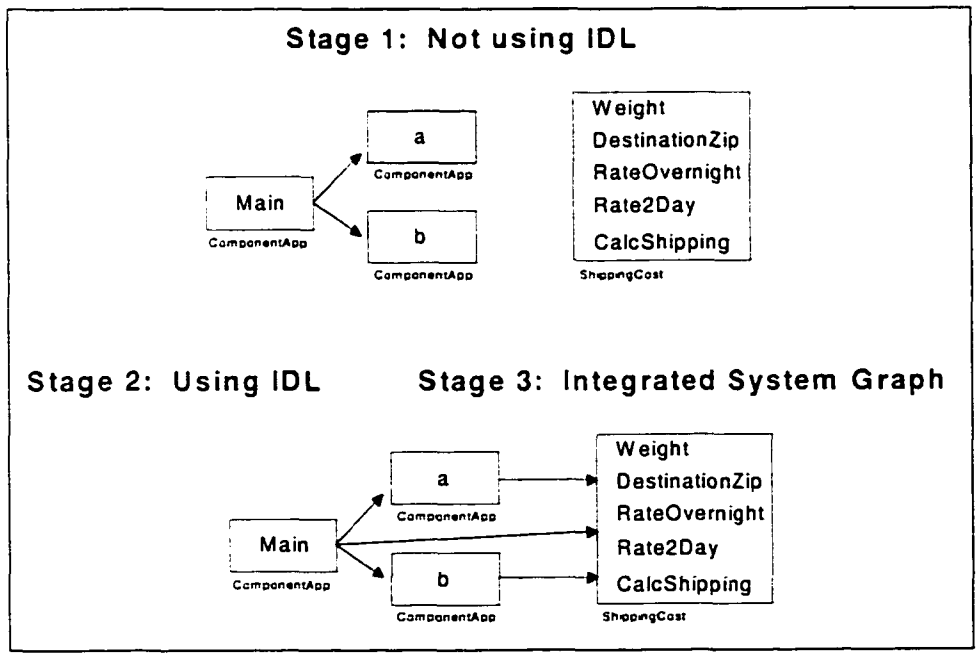

Figure 26. Summary call graph evolution for ShippingCost

As in the previous example, the stage 1 call graph does not show any paths from the modules in the component application to the ShippingCost component. After integrating the appropriate declarations we see that stage 2 actually completes the call graph. This is because no additional call paths come from the ShippingCost component to other components. The advantage of producing the stage 3 integrated system is that the correct parameter mapping information is incorporated. This can be seen by looking at a coupling analysis report for the summary graphs at both stages in Table 5 and Table 6. The stage 3 analysis showed two additional couplings. Here we see that the integrated system graph represented more precise information than the non-integrated graph in stage 2.

CallGraphNode	CallSite	Actual Parameter	Formal Parameter
NodeID="4" ModuleName="Main" ParentModName="Simple1"	ModuleName="a" ModuleID="2" InModuleCollection="Simple1"	VarName="sys1" VarID="6"	PassByRefVarName="x" VarID="2" FirstUse="REF" LastUse="REF

Table 5. Stage 2 coupling analysis of ShippingCost

CallGraphNode	CallSite	Actual Parameter	Formal Parameter
NodeID="2 ModuleName="a" ParentModName="Simple1"	ModuleName="weight (Property Let)" ModuleID="8" InModuleCollection="ShippingCost"	VarName="y" VarID="3"	PassByVal VarName="vNewValue" VarID="12" FirstUse="REF" LastUse="REF"
NodeID="3" ModuleName="b" ParentModName="Simple1"	ModuleName="weight (Property Let)" ModuleID="8" InModuleCollection="ShippingCost"	VarName="x" VarID="4"	PassByVal VarName="vNewValue" VarID="12" FirstUse="REF" LastUse="REF"
NodeID="4" ModuleName="Main" ParentModName="Simple1"	ModuleName="a" ModuleID="2" InModuleCollection="Simple1"	VarName="sys1" VarID="6"	PassByRefVarName="x" VarID="2" FirstUse="REF" LastUse="REF

Table 6. Stage 3 Coupling analysis of ShippingCost

The simple examples above illustrate the effectiveness of this technique. Probably the most important contribution of this technique is that it provides a way for

many useful static analysis techniques to be applied more precisely on a component-based system where previously, many of those techniques could not have been applied at all due to the lack of source code for components. In the validation section, we apply this technique to several case studies to examine its effectiveness more closely.

It is important to note that the success of this approach depends on component developers being able to generate this information with as little effort as possible. To that end, it is envisioned that an analysis tool such as the one modified for this research could be provided to component developers so that they can generate the necessary summary information graphs and distribute them appropriately. In the short term, this means that several similar tools could be developed to handle the more common languages that are typically used in building components and component-based systems. This would support, for example, components written in Visual C++ or Visual Basic, and a user application written in Java. This research has defined a foundation for making such a task straightforward. A significant effort would still be necessary to obtain the necessary buy-in from component developers to get them to use the tools and distribute the necessary summary information. In the long term, this technique presents a good example of additional information that is necessary to augment components. Whether the specific techniques or schemas are used, this results of this thesis do show that better ways to develop, describe, analyze and test distributed components for both developers and component users are necessary.

6. TOOLS TO SUPPORT THE AUTOMATED ANALYSIS OF COMPONENT-BASED SOLUTIONS

This section describes the tools developed or modified to support the automated analysis techniques developed as part of this research on component-based systems. The first tool provides the ability to capture component information from a type library or interface definition language file for use in integrating that information with Project Analyzer, a commercially available Visual Basic code analysis tool. The next tool is a set of extensions made to Project Analyzer to analyze particular component property criteria of interest in support of detecting unreachable code. Next, another set of extensions to Project Analyzer is described to support the generation of summary information graphs that contain global data flow information such as variable def-use, first-use/last-use, parameter couplings and more for each call graph node in a system being analyzed. The extended call graph with this summary information embedded is then stored in an extensible markup language (XML) format. The call graphs from separate and distinct components and a user application are then merged into an integrated system for further analysis. Finally, a number of extensible stylesheet language (XSL) [6] scripts that provide additional analyses and views applied to the XML graphs are described, such as parameter coupling analysis, ripple analysis, and call graph metrics.

6.1. Type library documenter

Figure 27 depicts a freeware custom-developed Visual Basic tool, called the ActiveX Documenter, which was used in this research to capture the type library information for a component [1]. The result is a more human-friendly view of the type library information, showing the methods and properties with their associated parameter.

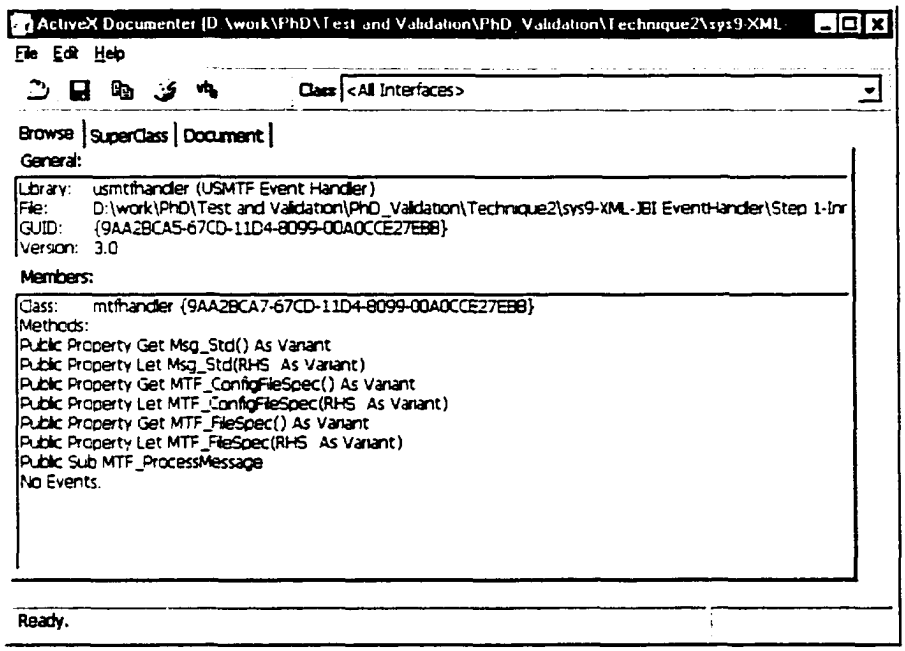

Figure 27. Type library documentation tool

This application uses the type library component, TLBINF32.DLL that gets installed with MS Visual Basic to investigate the interfaces of compiled ActiveX components. It acts as a complement to the object browser in Visual Basic, allowing access to object interfaces without needing to use Visual Basic or to add a reference to the object. In addition, it produces well-formatted documentation for an object. Using this tool, one can:

- Quickly browse an ActiveX object's members;
- Copy the member definitions as fully formatted VB code for use in other applications;
- Create documentation about a component using the procedure attributes built into the ActiveX object's Type Lib.

This tool was used extensively in this research to help gain more insight into the IDL for a component and subsequently import that information into a declarations module to improve the call graph analysis within Project Analyzer.

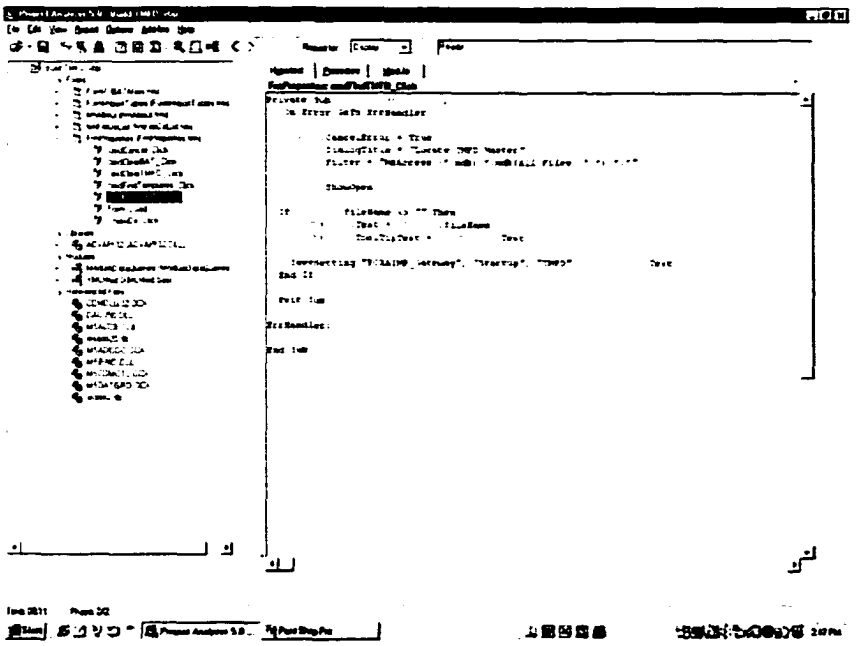

Figure 28. Project analyzer

6.2. Project Analyzer

Project Analyzer is a software maintenance tool that analyzes Visual Basic projects [109]. This shareware tool was used as the basis for analyzing the systems used during this research. Through a research agreement with the tool's author, full source code for the tool was made available for our use in this work. An example of the tool being run is depicted in Figure 28. From this analysis, several reports can be generated that show the project's structure and nature. These reports include dead code, calls and called-from information, several metrics, and more.

Project Analyzer's analysis consists of two phases. Phase one gathers information about the structure of the project. This phase is rather simple in that all 'tokens' are gathered and categorized as global procedures, global identifiers, local procedures, local identifiers, etc. If a token fits one of these categories, its is numbered and stored in the appropriate list. The lists are variable-length array constructs for each type of category.

Phase two collects cross-reference information and calculates some of the report information. This phase again looks at each token to gather more information about it.

During this phase, each token is scoped and cross-referenced against other tokens. Information such as which calls a procedure makes and the number of times a variable is set or referenced is gathered. Nested conditionals are analyzed and dead code is identified.

Project Analyzer first populates a symbol table with identifiers, procedures, and files in one pass. In the next pass, project analyzer gathers information about each symbol and prepares for several of its reports. This analysis is then used for several built-in reports.

The off-the-shelf project analyzer tool did not do several key functions that were important for our research. First, it did not analyze components or provide any tools to assist developers in the analysis of component-based systems. Second, it did not provide any graph generation for call and control flow information. Finally, it did not provide any advanced static analysis capabilities such as global variable ref-def analysis, variable first-use and last-use analysis, ripple analysis, coupling analysis and the like. However, since it did a fine job at doing the basic set of analysis for a language we required for our test cases, we felt that it was a good choice. We made several extensions to the basic tool to support our research.

6.3. Project analyzer extension: OA-dead analysis

To support our technique for leveraging semantic information about component properties to improve specific analysis techniques, extensions were made to the project analyzer tool to conduct an OA-dead analysis of a component-based system. The foundation for this extension is described in detail in R. Sparks Master's project report [113]. Subsequent modifications to this extension were made during this research to collect various metrics needed to help validate the effectiveness of the technique. The OA-dead report addition to project analyzer shows that some program run-time behavior can be predicted in Visual Basic and similar languages with general (non-program specific) analyses. Specifically, this report utilizes analyses of Visual Basic semantic object properties to extend the search for unreachable or dead code. This extension, referred to as OA-dead code, identifies unreachable code that traditional static analysis

would not find, such as code that is unreachable due to run-time characteristics and component property settings.

6.3.1. OA-dead code analysis

It is not possible to verify when an object is made available during run-time. It is possible to determine if an object is initialized as not available and if the object's availability is ever changed during run-time. It is also not possible to determine for certain what the object's availability is changed to unless it is set to TRUE (also 1) or FALSE (also 0) because it is not possible to evaluate the run-time value of expressions. Therefore, the only certain OA-dead object is one that is initialized as not available and whose availability is never set to anything but FALSE. For example, an object that is initialized as invisible and that is never set to visible would be OA-dead. Yet, the OA-dead state of an object that is initialized as invisible but that is somewhere in the code set to visible cannot be determined in general. This is also true for the initially invisible object whose visible attribute is later set to some expression. In these cases, the object may or may not be OA-dead.

Visual Basic attributes that can affect an object's run-time availability are enabled, visible, width, height, left, and top. Objects that are not enabled or not visible are obviously not subject to user input. These objects are OA-dead. Objects whose width or height attributes are zero are invisible to the program user as well. These type objects are also considered OA-dead. Objects whose left attribute is set so that the object is not visible in the active window are also invisible to the user. These objects may be too far to the left or too far to the right and are OA-dead. Similarly, objects may be too far up or down to be visible in the active window as determined by their top attribute. These are also OA-dead.

If a control is OA-dead, then all of its corresponding events are OA-dead as well. For example, if a command button is invisible then it is impossible to execute its 'click', 'double_click' or other event procedures. These procedures are unreachable and therefore OA-dead.

Visual Basic treats forms as files and controls. For the purpose of OA-dead analysis, an OA-dead form is treated as both an OA-dead file and an OA-dead control. OA-dead files are analyzed the same way that dead files are analyzed. All procedures and data structures in an OA-dead file are OA-dead as well. Likewise, OA-dead procedures are analyzed the same as dead procedures. Procedures called only by OA-dead procedures are considered to be OA-dead as well.

Therefore, a Visual Basic object may be OA-dead due to its own semantic nature (the value of one or more of its semantic attributes) or it may be OA-dead because of its syntactic relationship with another OA-dead object. Thus, OA-dead analysis must first evaluate the semantic nature of each object's attributes to determine its OA-dead nature. Then an analysis of each object's syntactic relationship with other objects must be evaluated to determine if the OA-dead nature is inherited. This second check is the same as would be accomplished for normal dead code analysis.

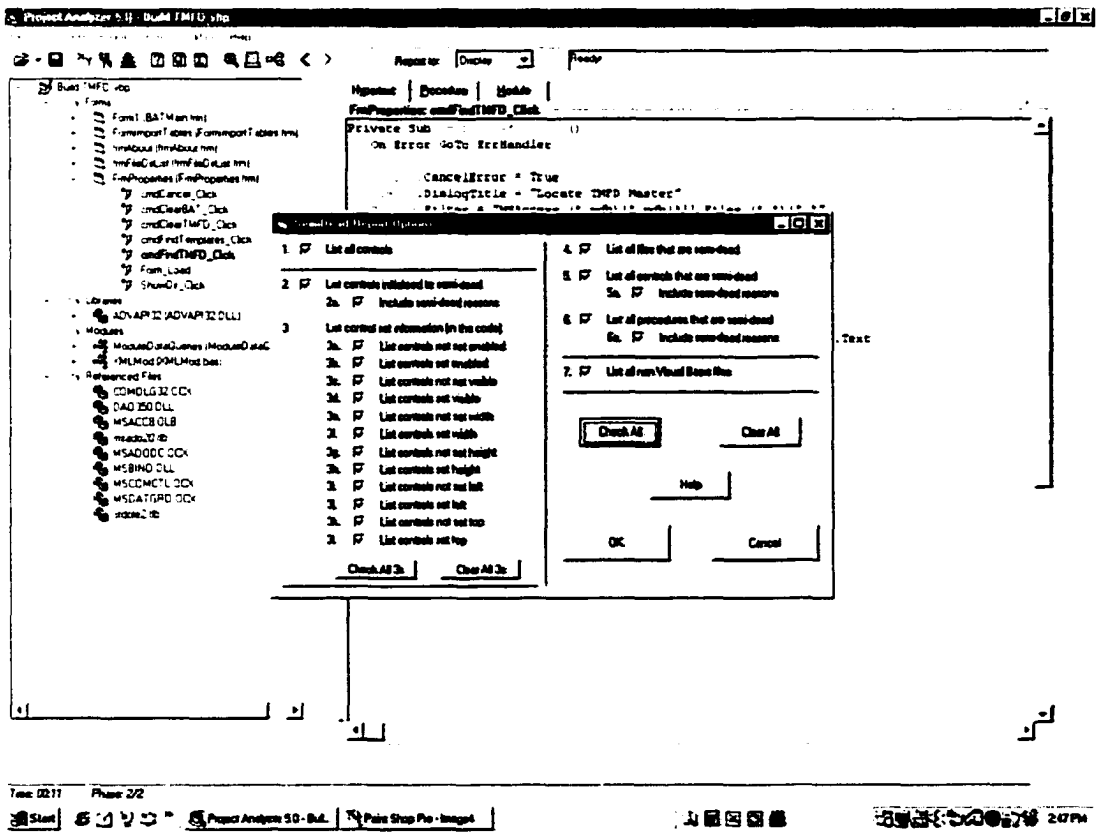

Figure 29. Project analyzer OA-dead analysis extension

6.3.2. OA-dead report

Figure 29 provides an example of running the OA-dead analysis from project analyzer. To perform OA-dead analysis on a VB project, a menu-option is available the modified version of Project Analyzer. This report implements the OA-dead analysis described previously and provides several options as to what and how much resulting information is listed in the report. Each option below enables a list of objects as described. A count of the number of objects in each list is included at the bottom of each list in the report.

- Option 1. List of all controls. This option provides a complete list of all controls in the VB project.

- Option 2. List of all controls initialized to OA-dead. This option provides a complete list of all controls in the VB project that are OA-dead in their initial state. That is one or more of the control's attributes (e.g., enabled, visible, height) are initially set to make the control unreachable. Again, this is the initial semantic state only.

- Option 3. List of all controls set conditions (whether or not the attribute is changed from its initialized state somewhere in the code).

- Option 3a. List of controls set enabled. This option provides a complete list of all controls whose enabled attribute is set to something other than FALSE or 0 somewhere in the code. A control's presence on this list means that it cannot be verified as OA-dead due to its enabled attribute by our algorithm.

- Option 3b. List of controls not set enabled. This option provides a complete list of all controls whose enabled attribute is not set to something other than FALSE or 0 somewhere in the code.

- Option 3c. List of controls set visible. This option provides a complete list of all controls whose visible attribute is set to something other than FALSE or 0 somewhere in the code. A control's presence on this list ensures that it cannot be verified as OA-dead due to its visible attribute.

- Option 3d. List of controls not set visible. This option provides a complete list of all controls whose visible attribute is not set to something other than FALSE or 0 somewhere in the code.

- Option 3e. List of controls with width set. This option provides a complete list of all controls whose width attribute is set to something somewhere in the code. A control's presence on this list ensures that it cannot be verified as OA-dead due to its width attribute.

- Option 3f. List of controls with width not set. This option provides a complete list of all controls whose width attribute is not set somewhere in the code.

- Option 3g. List of controls with height set. This option provides a complete list of all controls whose height attribute is set to something somewhere in the code. A control's presence on this list ensures that it cannot be verified as OA-dead due to its height attribute.

- Option 3h. List of controls with height not set. This option provides a complete list of all controls whose height attribute is not set somewhere in the code.

- Option 3i. List of controls with left set. This option provides a complete list of all controls whose left attribute is set to something somewhere in the code. A control's presence on this list ensures that it cannot be verified as OA-dead due to its left attribute.

- Option 3j. List of controls with left not set. This option provides a complete list of all controls whose left attribute is not set somewhere in the code.

- Option 3k. List of controls with top set. This option provides a complete list of all controls whose top attribute is set to something somewhere in the code. A control's presence on this list ensures that it cannot be verified as OA-dead due to its top attribute.

- Option 3l. List of controls with top not set. This option provides a complete list of all controls whose top attribute is not set somewhere in the code.

- Option 4. List of all OA-dead files. This option provides a complete list of all files that are OA-dead. These files may be form files whose attributes make them OA-dead or they may be files whose only parent files are OA-dead (no non-OA-dead files uses them).

- Option 5. List of all OA-dead controls. This option provides a complete list of all controls that can be verified to be OA-dead. Controls on this list have attributes that make them unreachable in the initial state and that are not set to change this state elsewhere in the code.

- Option 5a. List of OA-dead reasons. This option will display a count of the reasons (invisible, disabled, etc.) that a control is OA-dead somewhere in the system. This option can only be checked if Option 5 is checked. Checking this option will include a frequency count of each reason with the object count at the bottom of this list. These reason counts may total more than the list's object count because an object may have more than one reason for being OA-dead.

- Option 6. List of OA-dead procedures. This option provides a list of all procedures that can be verified to be OA-dead. This list will not show a procedure if it is also dead. Procedures on this list are OA-dead if they are in an OA-dead file or if they are called only by OA-dead procedures (no non-OA-dead procedures use them). Procedures on this list may also be OA-dead if they are the event procedures of an OA-dead control. Note that Project Analyzer treats file declaration blocks as procedures so the declaration block of a OA-dead file may show up on this list as a OA-dead procedure.

- Option 6a. List of OA-dead reasons. This option will display the relationship(s) (file, control, or procedure) that causes the procedure to be OA-dead. This option can only be checked if Option 6 is checked. Checking this option will include frequency count of each reason with the object count at the bottom of this list. These reason counts may total more than the list's object count because an object may have more than one reason for being OA-dead.

- Option 7. List of project files that are not Visual Basic files. These would include binary, DLL, and resource files. This list is useful in OA-dead analysis because Project Analyzer can only analyze Visual Basic files. Calls to (or from) non-Visual Basic files could possibly impact the OA-dead nature of code.

After running the OA-dead analysis report, the results will be display to the user in the selected output manner. For example, Figure 30 shows the results of an OA-dead analysis being displayed on the screen. In this case, we see the start of a long list of OA-dead procedures along with their reasons for being in that state. Details of the extensions made to project analyzer to incorporate this capability can be found in Appendix C.

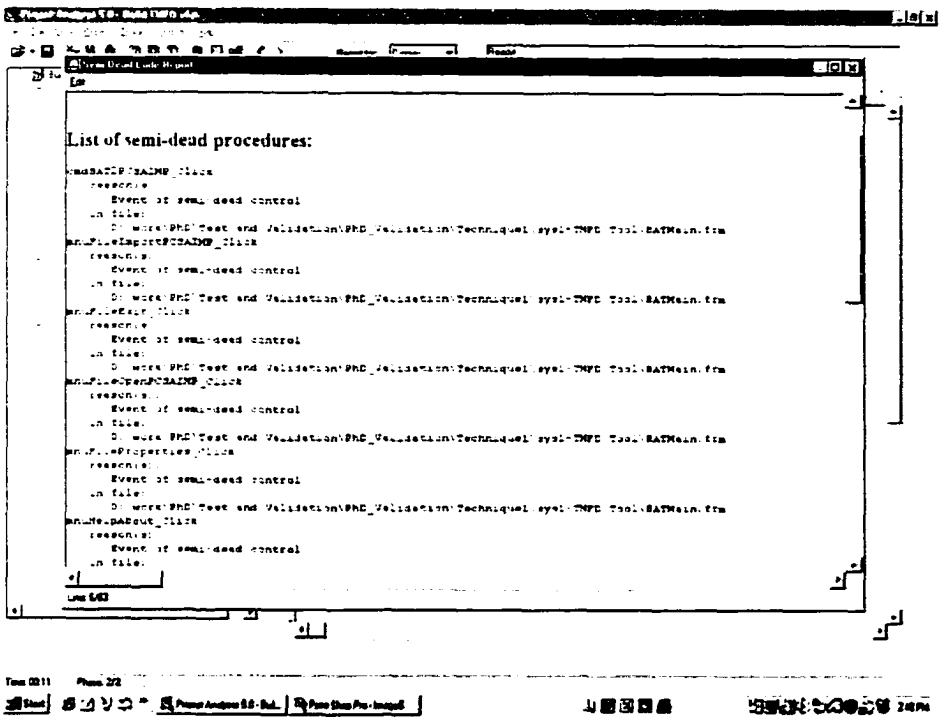

Figure 30. OA-dead analysis report

6.4. Project analyzer extension: call graph summary information

To support our technique for component developer summary information, several extensions were made to the project analyzer tool to perform a global ref-def and usage analysis, summary call graph generation in XML format, parameter mapping and ripple analysis data dependence capabilities.

A phase three analysis for computing the global data flow was added to the existing two analysis phases of project analyzer. This new phase calculates call graph nodes, as well as global ref-def lists, variable first-use and last-use, and call site parameter mappings for all call graph nodes. The results of this analysis are stored in a series of tables in a Microsoft Access relational database to provide the information necessary for the summary graphs and subsequent dependence reports to be constructed quickly.

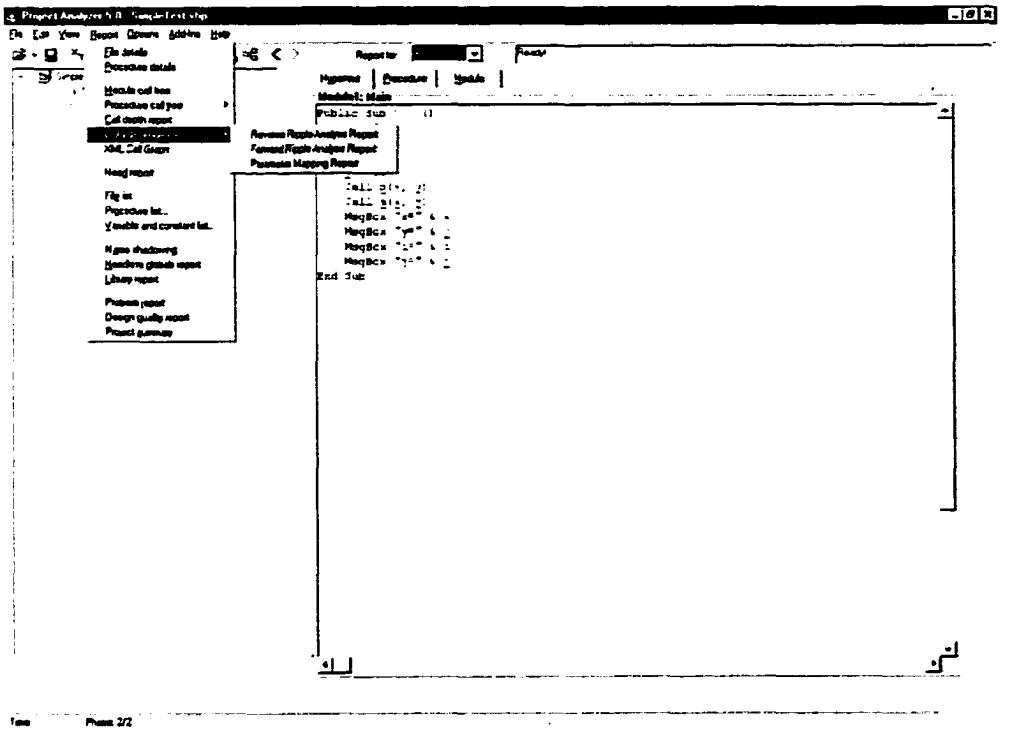

Figure 31. Project analyzer summary graph extension

Once the phase three analysis is completed, a user of the tool has access to all report options from pull-down menus and toolbar buttons, including the normal project analyzer capabilities. Figure 31 shows an example of this. From the procedure call tree menu depicted in the figure, the user has the ability to generate an XML summary call graph as well as the ability to load an existing graph. Under the added data dependence menu, the user also has access to the parameter mapping and ripple analysis graph generation options. Figure 32 shows an example of a summary call graph for a system that was generated from our project analyzer extension and viewed using the Microsoft Internet Explorer web browser.

Details of the extensions made to project analyzer to incorporate this capability can be found in Appendix C.

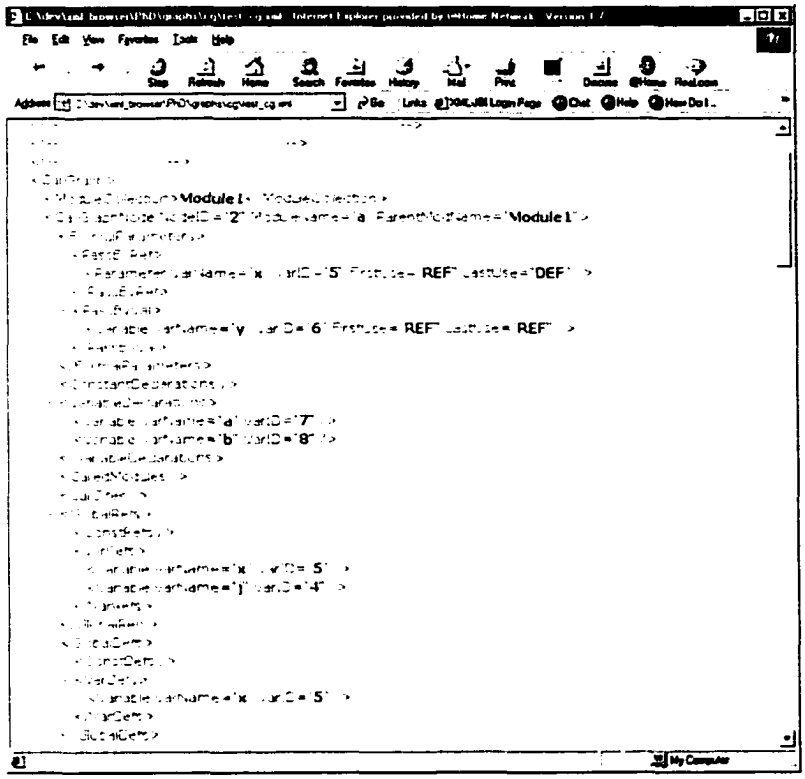

Figure 32. Project analyzer example summary call graph

6.5. Static analysis XSL stylesheets

Since the format chosen for the summary call graph as well as the parameter mapping and ripple dependence graphs was XML, we are able to take advantage of many useful off-the-shelf tools, many of which are free, to process, manipulate, and display the information in many forms. XSL is one such tool that is used for transformations of XML documents. In an XSL transformation, an XSL processor reads both an XML document and an XSL style sheet. Based on the instructions the processor finds in the XSL style sheet, it outputs a new XML document or fragment thereof. There's also special support for outputting hypertext markup language (HTML). With some effort it can also be made to output essentially arbitrary text or other formats, though it is designed primarily for XML-to-XML transformations [6]. In our research, we developed several XSL scripts to provide a number of views, including:

- Call graph table view – used for providing a graphical layout in HTML of the entire summary call graph.

- Call graph metrics view – used to calculate and display various metrics about each call graph node for all modules in HTML format.

- Parameter coupling analysis view – used to compute the parameter based coupling paths and interfaces in a summary call graph. The paths represent couplings between modules for each parameter where there is a path from the last definition of a parameter through the call site to a referenced first use of the formal parameter. Formal parameters whose first use is a definition do not constitute a coupling. This report is useful for testing and is displayed in HTML tabular format.

- Call coupling analysis view– used to compute the call based coupling paths and interfaces in a summary call graph. The paths represent couplings between modules for each call site.

- Parameter mapping dependence view - used to display the results of the parameter mapping dependence graph in tabular HTML format.

- Ripple analysis dependence view - used to display the results of the ripple analysis dependence graph in tabular HTML format.

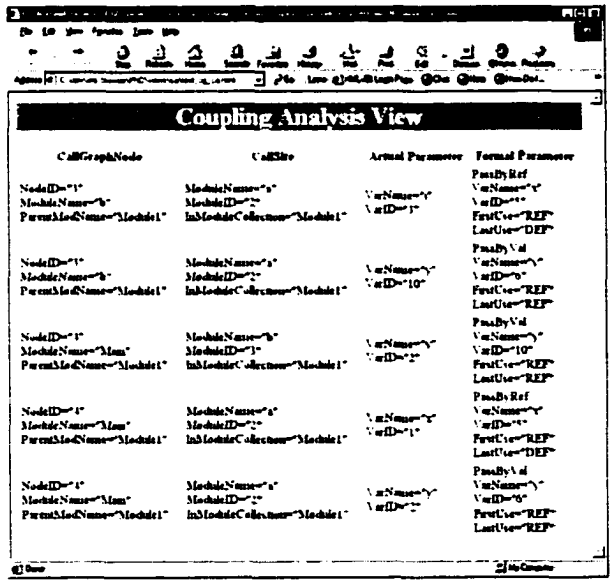

Figure 33. Results of applying parameter coupling XSL

The XSL processor we used for this research is the freely available LotusXSL tool from IBM Alphaworks [2]. Figure 33 shows the results of applying the parameter coupling analysis XSL against the XML summary call graph for the test.bas sample program discussed previously.

Source listings for the XSL scripts developed and used in this research can be found in Appendix C. In the next section, we apply the tools and techniques we have developed for this research on several case study systems to evaluate their effectiveness in practical use.

7. VALIDATION OF THESE TECHNIQUES THROUGH PRACTICE

This section demonstrates the effectiveness of the techniques discussed in sections 4 and 5 on several case studies. Seven case studies represent real COTS component-based systems developed and maintained by the Department of Defense (DoD). Two additional case studies are used to represent academic examples designed to illustrate some interesting aspects of component-based development. The detailed graphs and views generated for case study 7 are listed in Appendix B as a sample. Due to the size of many of the reports, the detailed graphs and views for all case studies can be obtained by contacting the author. A summary of the results from this experimentation is presented here.

To attempt to validate the effectiveness of the techniques discussed previously, we applied both traditional static analysis techniques and our extended techniques on the nine case studies briefly discussed below, and made a comparison of the results. The following criteria were used in this comparison:

- Total source lines of code (SLOC)
- Code size in kilobytes
- Initial analysis time in minutes and seconds
- Total number of components
- Total number of files
- Total number of procedures
- Total number of invocation dead procedures
- Total number of controls
- Total SLOC of XML summary call graph
- File size in kilobytes
- Extended analysis time in minutes and seconds
- Total number of modules collections
- Total number of call graph nodes
- Total number of call sites
- Total number of globally referenced variables
- Total number of globally defined variables
- Total number of parameter coupling paths and interfaces
- Total number of call coupling paths and interfaces

- Total number of parameter mapping dependencies
- Total number of reverse ripple dependencies
- OA-Dead files, procedures and controls (and their reasons for being OA-dead)
- Number of parameter coupling paths, call coupling paths, parameter mapping dependencies and reverse ripple dependencies reduced as a result of the OD-dead analysis

The analysis of the systems was conducted on a 400mgz personal computer with 256k of memory running Microsoft Windows 98. We started by analyzing each of the nine systems using the original Project Analyzer tool to collect program summary information such as total source lines, total number of components and files in the system, total number of live and dead procedures, and total number of controls. We then applied our extended version of Project Analyzer to each of the nine systems to collect global variable usage information, such as ref-def and first-use and last-use information. We then generated the summary call graph, parameter mapping dependence graph, and reverse ripple analysis graph for each system, and applied the call graph metrics, parameter coupling, call coupling, parameter mapping dependence, and reverse ripple analysis XSL views to the graphs to collect a number of call graph and data dependence metrics as a baseline.

To validate our first technique, we applied our OA-dead analysis algorithm to the first six systems to obtain the number of OA-dead files, procedures and controls. Then, using this information, we removed the dead procedures from each of the systems and re-analyzed them to discover the impact to parameter couplings, call couplings, parameter mapping dependencies, and reverse ripple dependencies due to any additional dead procedures being discovered.

To validate our second technique, we applied our summary call graph analysis to case studies 7 through 9 to assess the effect of component developers providing additional information about their components to users. The case studies used each had at least one component for which we had the source code. This allowed us to simulate the component provider in analyzing their component to produce and distribute the summary call graph. For this assessment, we examined and compared each system in three stages:

- Initial – This is the original state of the system that is typical of component-based systems today. Each consists of one main user application that integrates some number of components for which the user typically has no source code or additional information about the components.

- Extended – The extended state represents both component developers and users leveraging the type library information for the components being integrated to enhance the analysis of the component using the extended Project Analyzer tool. In this state, the component developers have not yet generated or distributed the summary call graph to the component user.

- Integrated – The integrated state represents the case where the component providers distribute their summary call graphs to component users, and the user integrates them with their system to produce an integrated system view.

We felt that by using this approach to compare the before and after states of applying first the traditional techniques and then our extended techniques should help to validate the research described in this thesis. A brief description of each of the nine test systems is described below, followed by a summary of the results and a discussion of efficiency.

7.1. System descriptions

7.1.1. Case Study 1 – TMFD

This case study represents a COTS component-based solution for the Department of Defense. The system provides a capability to convert platform implementation files between one particular database format to MS Access and subsequently to MS Excel for analysis. By the mechanism of OLE Automation, it interfaces with MS Access for database management; and MS Excel for spreadsheet support. This system has the following characteristics:

- 4917 SLOC ('glue' code)
- 14 components
- 21 files
- 136 procedures
- 89 controls

7.1.2. Case Study 2 – AFJCME_Full

This case study represents a COTS component-based solution for the Department of Defense. The example solution is a special-purpose tool (called AFJCME) built to support analysts in the maintenance and design of data interoperability standards and in the creation, maintenance and analysis of system implementations of those standards. The tool extracts implementation data from a database, and presents it to the analyst in spreadsheet form for modification. It further facilitates navigation among the data items by building a tree-structure control from the database that can be expanded and compressed as needed. These capabilities are implemented by a collection of COTS components. MS Visual Basic is used to implement the user interface and tree-structure, and to provide overall program control and integration. By the mechanism of OLE Automation, it interfaces with MS Access for database management; with MS Excel for spreadsheet support; with MS Outlook for event logging; and with the Internet via a Web browser to provide access to online DoD standards information. This system has the following characteristics:

- 4418 SLOC ('glue' code)
- 23 components
- 37 files
- 119 procedures
- 104 controls

7.1.3. Case Study 3 – AFJCME_Lite

This case study represents a COTS component-based solution for the Department of Defense. This tool is a variant to the AFJCME system described in case study 2. A requirement change necessitated development of a specialized version of the tool ('AFJCME_Lite') in which the links to Outlook and the Internet are eliminated, and in which the database provides only the navigation tree data. This system has the following characteristics:

- 5250 SLOC ('glue' code)

- 24 components
- 39 files
- 144 procedures
- 32 controls

7.1.4. Case Study 4 – Project Analyzer v5

Project analyzer is a software maintenance tool that analyzes Visual Basic projects. This shareware tool was used as the basis for analyzing the systems used during this research. From this analysis, several reports can be run that show the project's structure and nature. These reports include dead code, calls and called-from information, several metrics, and more. This is an update to the source used in case study 5. This system has the following characteristics:

- 40599 SLOC ('glue' code)
- 38 components
- 130 files
- 1531 procedures
- 667 controls

7.1.5. Case Study 5 – Project Analyzer v4

Project analyzer is a software maintenance tool that analyzes Visual Basic projects. This shareware tool was used as the basis for analyzing the systems used during this research. From this analysis, several reports can be run that show the project's structure and nature. These reports include dead code, calls and called-from information, several metrics, and more. This system has the following characteristics:

- 34102 SLOC ('glue' code)
- 32 components
- 116 files
- 1270 procedures
- 600 controls

7.1.6. Case Study 6 –CopyIt

This case study represents a COTS component-based solution for the Department of Defense. This tool provides a specialized email rule processing capability to perform

email-based publishing of information to a XML data server. At a user-configurable interval, CopyIT! accesses a mailbox through the Outlook 98 client. If an e-mail is found in the Inbox, CopyIT! compares the publisher's address, the message subject and attachment extensions to a set of user definable rules. If a match is found between a rule and corresponding message attributes, the message is processed in accordance with the rule. CopyIt makes extensive use of the MS Outlook component. This system has the following characteristics:

- 1700 SLOC ('glue' code)
- 9 components
- 17 files
- 84 procedures
- 154 controls

7.1.7. Case Study 7 – BookSale Manager

This project demonstrates the use of an ActiveX component to encapsulate the logic of business policies and rules and to provide 'black box' services to an external User Interface component [3]. The client project is dedicated to delivering a clear and intuitive user interface for the user to select control options and view processing results. The client project cares about how the user works and how they use the applications results, but it knows nothing about the business or operational rules of the application. The server project is an ActiveX component dedicated to encapsulating business and data access rules into 'sanctioned' services that client components use to find the information they need. The server component has no idea how the user options are selected or how the results are presented to the user. This lack of specific user knowledge helps keep the server component's functionality general, and as a result should increases its reusability potential (in a real project) for other applications. It also uses Class modules to structure the logic of its business and data access rules in a manner that aids development, debugging, readability, maintainability, and source code reusability [3]. This initial system has the following characteristics:

- 521 SLOC for the client application, and 400 for the server component

- 13 components for the client application, and 7 for the server component
- 17 files for the client application, and 12 for the server component
- 27 procedures for the client application, and 17 for the server component
- 74 controls for the client application, and 8 for the server component

7.1.8. Case Study 8 – Excel Charting Application

This example demonstrates the wrapping of a component by illustrating a user application that constructs charts or graphs based on some user selected criteria. Microsoft Excel provides a powerful off-the-shelf set of capabilities for doing this, however, the user may not need the full set of functionality Excel provides. Furthermore, the user may not have a licensed copy of MS Excel available. With this example, one copy of MS Excel needs to be available on a server for which this component is registered and the component user has access to. The component will wrap the charting capabilities of MS Excel that exposes them to the component user of the component in a very simple to use fashion. The chart would be generated on the server using MS Excel and then exported to a simple GIF file at the desired location on the server. Once the GIF file is ready it can be rendered to a thin client or accessed by a component user application such as the one exemplified here [7]. This initial system has the following characteristics:

- 125 SLOC for the client application, and 380 for the server component
- 6 components for the client application, and 5 for the server component
- 7 files for the client application, and 6 for the server component
- 4 procedures for the client application, and 11 for the server component
- 8 controls for the client application, and 0 for the server component

7.1.9. Case Study 9 – XML-JBI EventHandler

This case study is a COTS component-based solution for the Department of Defense. This tool provides inbox monitoring, subscription processing and transformation services to an XML middle-tier data server that supports a set of common core services, including user personalization services, publishing services, information transformation services, and retrieval and presentation services. A main application, Dispatcher, is used to provide the above services. It detects when information is

published and passes it to an appropriate eventhandler for further processing. Several components are used in this system, including one for each of the various event handlers and for the sendmail component to handle email notifications and content delivery. This initial system has the following characteristics:

- 495 SLOC for the main Dispatcher application, 190 for the GCSS component, 1416 for the sendmail component, 794 for the mtf component, and 199 for the webcop component
- 11 components for the main Dispatcher application, 5 for the GCSS component, 8 for the sendmail component, 7 for the mtf component, and 5 for the webcop component
- 16 files for the main Dispatcher application, 6 for the GCSS component, 11 for the sendmail component, 8 for the mtf component, and 6 for the webcop component
- 30 procedures for the main Dispatcher application, 8 for the GCSS component, 73 for the sendmail component, 29 for the mtf component, and 9 for the webcop component
- 4 controls for the main Dispatcher application, 2 for the sendmail component, and 0 for GCCS, mtf and webcop components

7.2. Summary of results

This section briefly summarizes the results of applying the techniques we have defined on the nine case studies discussed above. In section 7.2.1, we highlight the results of applying the OA-Dead analysis technique discussed previously to the first six case study systems to assess the effectiveness of using semantic information about component properties to augment a specific analysis capability. In section 7.2.2, we highlight the results of applying the summary call graph technique on the remaining three case study systems. In each of these sections, we provide a table showing the metrics we identified earlier, followed by one or more summary charts and a brief discussion of the results. A sample of the detailed graphs and views generated for each system are listed in Appendix B.

7.2.1. Applying the OA-dead analysis technique to case studies 1-6

The results of applying the OA-dead analysis algorithm to the first six case study systems is depicted in Table 7 below.

Metric	CS1	CS2	CS3	CS4	CS5	CS6
PROGRAM SUMMARY						
Total SLOC:	4917	4418	5250	40599	34102	1700
Code Size in kB:	201	207	254	1438	1211	110
Time for Initial Parse and Analysis:	:20	:19	:22	7:30	5:47	:09
Components:	14	23	24	38	32	9
Total Files:	21	37	39	130	116	17
Total Procedures:	136	119	144	1531	1270	84
Dead Procedures:	1	21	32	111	83	0
Total Controls:	89	104	130	667	600	154
CALL GRAPH METRICS						
Total SLOC of Call Graph:	12370	9061	10824	216738	183578	6500
File Size in kB:	296	202	238	7303	6255	150
Time for Global Analysis and Generation:	:34	:30	:24	40:10	33:30	:15
Total # of ModuleCollections:	8	18	19	97	89	8
Total # of CallGraphNodes:	136	119	144	1531	1270	84
Total # of CalledModules:	172	70	83	2986	2485	58
Total # of CallSites:	222	76	91	4761	4017	79
Total # of CallSites with parameter mappings:	222	76	91	4761	4017	79
Total # of GlobalRefs:	259	249	289	38542	34286	228
Total # of GlobalDefs:	30	109	116	10326	9027	191
DATA DEPENDENCE METRICS						
Total # of Parameter Coupling Interfaces:	300	96	86	5620	4748	156
Total # of Parameter Coupling Paths:	150	48	43	2810	2374	78
Total # of Call Coupling Interfaces:	444	152	182	9522	8142	158
Total # of Call Coupling Paths:	222	76	91	4761	4071	79
Total # of Parameter Mapping Dependencies:	203	62	59	6709	5519	78
Total # of Reverse Ripple Dependencies:	90	57	53	12611	9980	340
OA-DEAD ANALYSIS						
OA-Dead Files:	0	2	2	0	0	0
OA-Dead Procedures:	9	26	43	230	176	9
Reasons:						
In OA-Dead File:	0 (0%)	2 (7.69%)	2 (4.65%)	0 (0%)	0 (0%)	0 (0%)
Event of OA-Dead Control:	8 (88.89%)	17 (65.38%)	32 (74.42%)	151 (65.65%)	118 (67.05%)	9 (100%)
Called by OA-Dead Procedure:	1 (11.11%)	7 (26.92%)	9 (20.93%)	79 (34.35%)	58 (32.95%)	0 (0%)
OA-Dead Controls:	19	49	72	249	214	19
Reasons:						
Disabled:	2 (10.53%)	1 (2.04%)	3 (4.17%)	4 (1.61%)	0 (0%)	6 (31.58%)
Invisible:	0 (0%)	3 (6.12%)	17 (23.61%)	70 (28.11%)	72 (33.64%)	1 (5.26%)
Too Narrow:	18 (94.74%)	34 (69.39%)	52 (72.22%)	198 (79.52%)	175 (81.78%)	13 (68.42%)
Too Short:	18 (94.74%)	34 (69.39%)	52 (72.22%)	198 (79.52%)	175 (81.78%)	13 (68.42%)

Metric	CS1	CS2	CS3	CS4	CS5	CS6
Too Far Right:	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Too Far Down:	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Too Far Left:	0 (0%)	3 (6.12%)	3 (4.17%)	0 (0%)	0 (0%)	0 (0%)
Too Far Up:	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
In OA-Dead File:	0 (0%)	12 (24.49%)	13 (18.06%)	0 (0%)	0 (0%)	0 (0%)
Total # of Parameter Coupling Interfaces reduced via OA-Dead Analysis:	0	0	4	0	7	3
Updated Total # of Parameter Coupling Interfaces:	300	96	78	5620	4734	156
Updated Total # of Parameter Coupling Paths:	150	48	39	2810	2367	78
Total # of Call Coupling Interfaces reduced via OA-Dead Analysis:	0	2	1	0	14	2
Updated Total # of Call Coupling Interfaces:	444	148	180	9522	8006	154
Updated Total # of Call Coupling Paths:	222	74	90	4761	4003	77
Total # of Parameter Dependencies reduced via OA-Dead Analysis:	0	4	5	14	36	3
Updated Total # of Parameter Dependencies:	203	58	54	6695	5483	75
Total # of Reverse Ripple Dependencies reduced via OA-Dead Analysis:	2	3	2	536	199	11
Updated Total # of Reverse Ripple Dependencies:	88	54	51	12075	9781	329
SUMMARY						
% I-Dead Procedures:	0.7%	17.6%	22.2%	7.3%	6.5%	0.0%
% OA-Dead Procedures Found:	6.6%	21.8%	29.9%	15.0%	13.9%	10.7%
Updated % of total Dead Procedures (I&OA):	7.4%	39.5%	52.1%	22.3%	20.4%	10.7%

Table 7. Results of applying OA-dead analysis to case studies 1-6

Table 7 categorizes the various metrics into the following groups: program summary, call graph metrics, data dependence metrics, OA-Dead analysis results, and summary averages.

For case study 1, a 4917 line program with 136 total procedures and 89 controls, we see that the OA-Dead analysis resulted in the identification of 9 procedures and 19 controls that are OA-dead. This makes the total I-dead and OA-dead procedures to be 10. Of the OA-dead procedures, it is interesting to point out that over 88% were events to OA-dead controls, of which over 94% of those were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. Over 10% of the OA-dead controls were disabled. There were no significant decreases in any of the dependency metrics.

For case study 2, a 4418 line program with 119 total procedures and 104 controls, we see that the OA-Dead analysis resulted in the identification of 26 procedures and 49

controls that are OA-dead. This makes the total I-dead and OA-dead procedures to be 47. Of the OA-dead procedures, over 26% were called by another OA-dead procedure, and over 65% were events to OA-dead controls, of which over 69% of those were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. Over 24% of the OA-dead controls were in the 2 OA-dead files discovered. A small decrease in the number of call coupling, parameter mapping and reverse ripple dependencies.

For case study 3, a 5250 line program with 144 total procedures and 130 controls, we see that the OA-Dead analysis resulted in the identification of 43 procedures and 72 controls that are OA-dead. This makes the total I-dead and OA-dead procedures to be 75. Of the OA-dead procedures, over 26% were called by another OA-dead procedure, and over 74% were events to OA-dead controls, of which over 72% of those were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. Over 23% of the OA-dead controls were invisible. A small decrease in the number of call coupling, parameter mapping and reverse ripple dependencies.

For case study 4, a 40599 line program with 1531 total procedures and 667 controls, we see that the OA-Dead analysis resulted in the identification of 230 procedures and 249 controls that are OA-dead. This makes the total I-dead and OA-dead procedures to be 341. Of the OA-dead procedures, over 34% were called by another OA-dead procedure, and over 65% were events to OA-dead controls, of which over 79% of those were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. Over 28% of the OA-dead controls were invisible. There was a decrease of 14 parameter mapping and 536 reverse ripple dependencies.

For case study 5, a 34102 line program with 1270 total procedures and 600 controls, we see that the OA-Dead analysis resulted in the identification of 176 procedures and 214 controls that are OA-dead. This makes the total I-dead and OA-dead procedures to be 259. Of the OA-dead procedures, over 32% were called by another OA-dead procedure, and over 67% were events to OA-dead controls, of which over 81% of

those were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. Over 33% of the OA-dead controls were invisible. There was a decrease of 14 call coupling, 36 parameter mapping and 199 reverse ripple dependencies.

For case study 6, a 1700 line program with 84 total procedures and 154 controls, we see that the OA-Dead analysis resulted in the identification of 9 procedures and 19 controls that are OA-dead. This makes the total I-dead and OA-dead procedures to be 93. Of the OA-dead procedures, 100% were events to OA-dead controls, of which over 68% of those were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. Over 31% of the OA-dead controls were disabled and over 5% invisible. There was a decrease of 2 call coupling, 3 parameter mapping and 11 reverse ripple dependencies.

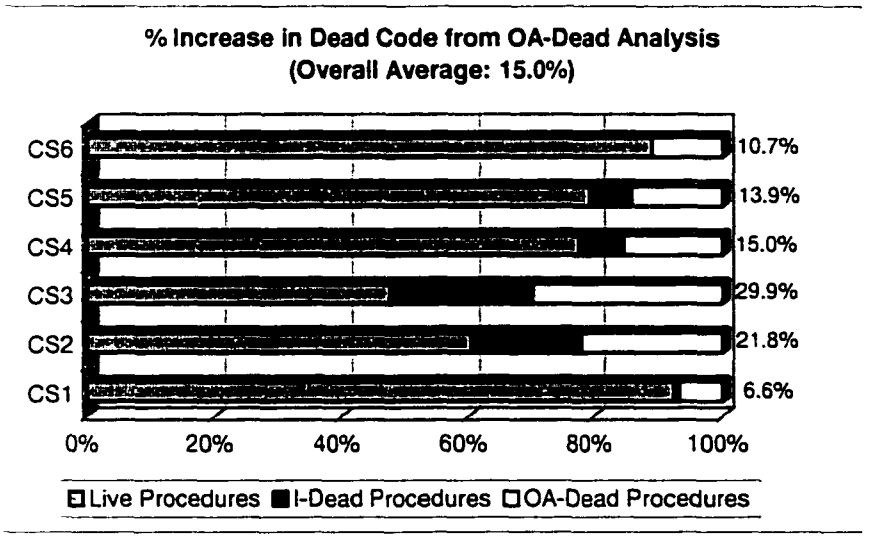

Figure 34. Increase in dead code from OA-dead analysis

To summarize these results, Figure 34 shows the percentage increases in OA-dead procedures discovered for each of the six case studies. It is interesting to point out that the percentages are very similar for the case studies that are nearly the same SLOC size. This implies that for a given visual-based system for which some level of maintenance

modifications are made, a significant amount of procedures and controls can be made OA-dead through typical actions of setting properties and re-sizing controls and forms. Looking across the average of all six case studies, we see that there is a 15% increase in OA-dead procedures from this analysis technique.

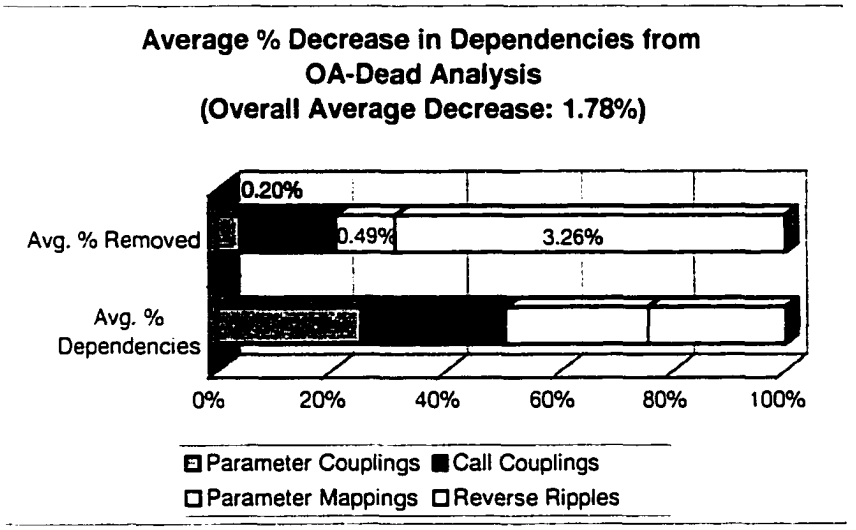

Figure 35. Average dependency decrease from OA-dead analysis

Since procedures are key to the data dependency metrics we used, it is natural to look at any decreases in the parameter coupling, call coupling, parameter mapping, and reverse ripple dependencies as a result of the OA-dead analysis. Figure 35 shows this decrease. The bottom bar represents the breakdown percentages across all six case studies for each of the dependency types. We see that this breakdown is fairly even. The top bar shows the percentage of decreased dependencies by dependency type. We see that the largest decrease is in reverse ripple dependencies. Overall, the average decrease is only 1.78%. This implies that none of the OA-dead procedures that were identified had any calling relationships, parameter usage relationships, or global ref-def relationships for the ripple dependencies. This verifies to some extent that the OA-dead procedure is in fact not being used.

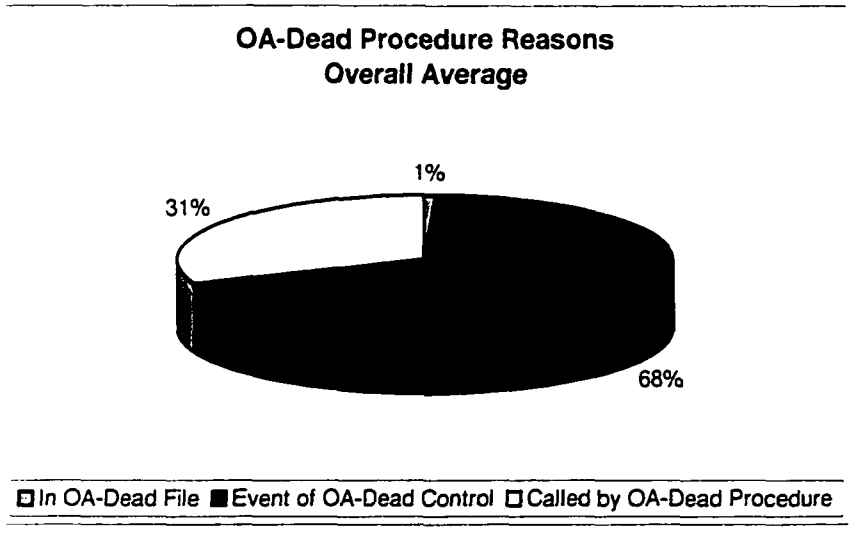

Figure 36. Overall average reasons for OA-dead procedures

Figure 36 shows the average breakdown across all six case studies of the reasons for the OA-dead procedures to be classified as such. A significant portion, 68%, are OA-dead as a result of being the event procedures of an OA-dead control and no other calls are made to that procedure. Event procedures are common for visual-based languages. Another 31% of the procedures were called from other OA-dead procedures and by no other live procedures.

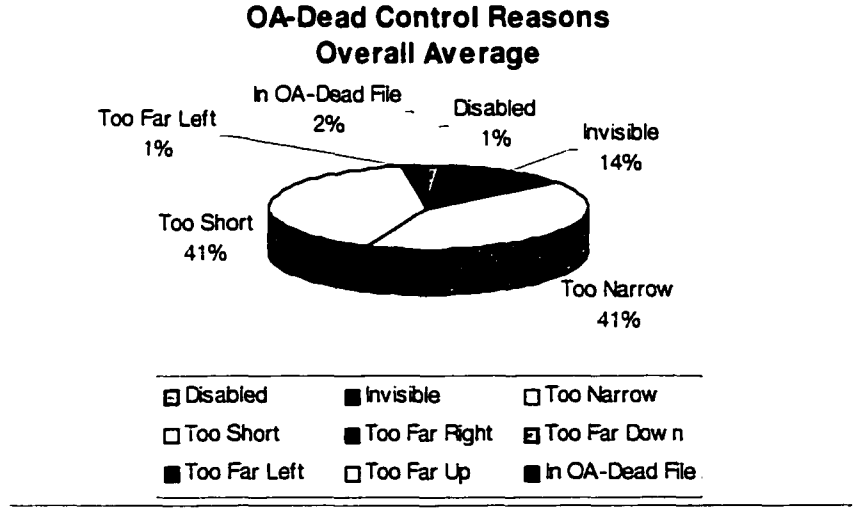

Figure 37. Overall average reasons for OA-dead controls

Figure 37 shows the average breakdown across all six case studies of the reasons for the OA-dead controls to be classified as such. The definitions of these reasons were discussed in Section 6. Over 41% were OA-dead due to the control being either too narrow, too short or both (in most cases) with respect to the form it was on. This implies that a significant amount of form or control resizing is done during maintenance to hide functionality. Over 14% of the OA-dead controls were invisible.

These results show that the OA-dead analysis proved to be an effective technique for discovering additional procedures and controls that were in a state when the analysis was performed that they could not be executed. An average increase of 15% across six real systems implies that the criteria which the analysis used are more common than expected in maintenance activities.

7.2.2. Applying the call graph summary information technique to case studies 7 through 9

To validate our second technique, we applied our summary call graph analysis to case studies 7 through 9 to assess the effect of component developers providing additional information about their components to users. A separate table for each system is used to summarize the results of the analysis on that system. Within each table, we distinguish between the initial or as-is state of the system, the extended state where current IDL information is used to aid the analysis of each component, and the integrated system state which reflects the result of a component user having access to the summary call graphs for each component. For both the initial and extended state, we show the analysis on the main user application containing the 'glue' code to integrate a number of components, as well as each of the components for which we had source code to simulate the component developer. The integrated system represents the merge of all XML summary call graphs for all components for which it was provided along with the graph of the main user application. To assess the results in the following tables, it is important to consider the role of the component user and compare the analysis of the main user application as the amount of available information for components is increased.

Metric	Case Study 7				
	Initial Client App.	Initial Server Comp.	Extended Client App.	Extended Server Comp.	Int. System
PROGRAM SUMMARY					
Total SLOC:	521	400	528	400	921
Code Size in kB:	42	16	43	16	59
Time for Initial Parse and Analysis:	:02	:01	:03	:01	:03
Components:	13	7	14	7	15
Total Files:	17	12	19	12	23
Total Procedures:	27	17	34	17	44
Dead Procedures:	2	1	4	1	4
Total Controls:	74	8	74	8	82
CALL GRAPH METRICS					
Total SLOC of Call Graph:	1804	1276	2195	1276	3201
File Size in kB:	34	28	40	28	67
Time for Global Analysis and Generation:	:04	:02	:05	:02	:08
Total # of ModuleCollections:	4	5	6	5	9
Total # of CallGraphNodes:	27	17	34	17	44
Total # of CalledModules:	11	9	16	9	25
Total # of CallSites:	14	10	19	10	29
Total # of CallSites with parameter mappings:	14	10	19	10	29
Total # of GlobalRefs:	54	49	54	49	134
Total # of GlobalDefs:	28	19	28	19	69
DATA DEPENDENCE METRICS					
Total # of Parameter Coupling Interfaces:	0	12	0	12	18
Total # of Parameter Coupling Paths:	0	6	0	6	9
Total # of Call Coupling Interfaces:	28	20	38	20	58
Total # of Call Coupling Paths:	14	10	19	10	29
Total # of Parameter Mapping Dependencies:	0	6	0	6	9
Total # of Reverse Ripple Dependencies:	9	20	9	20	43

Table 8. Applying the call graph summary information technique to case study 7

The results of applying the summary call graph analysis to case study 7 is depicted in Table 8. The initial system has one main user application that integrates 13 components, for which the server component listed is the primary. This component in turn integrates with 7 other components. In the initial system, the component user has no additional information about the components being integrated and there is no summary call graph for the server component. Constructing the summary call graph for the initial main user application shows 4 module collections, 27 call graph nodes, 14 call sites, 54

global referenced variables and 28 global defined variables. For data dependencies, we see 14 call couplings and 9 reverse ripples.

In the extended system, the component user uses the type library documentation to add the appropriate declares to the main application, resulting in more insight about the components. There is still no summary call graph for the server component. Constructing the summary call graph for the extended main user application shows 6 module collections, 34 call graph nodes, 19 call sites, 54 global referenced variables and 28 global defined variables. For data dependencies, we see 19 call couplings and 9 reverse ripples. It is important to note that using the type library information appropriately allowed the component user to extend some basic call graph information. However, the lack of global variable analysis in the type library information causes no improvement in the global referenced and defined variables or any of the data dependence analyses. The summary call graphs for the component are required to show such improvements.

In the integrated system, the component user obtains the summary call graph for the server component from the component developer, and integrates it with the information for the main application. Constructing the summary call graph for the integrated system shows 9 module collections, 44 call graph nodes, 29 call sites, 134 global referenced variables and 69 global defined variables. For data dependencies, we see 9 parameter couplings, 29 call couplings, 9 parameter mapping dependencies and 43 reverse ripples. This represents a significant improvement in data flow information from the initial system.

Metric	Case Study 8				
	Initial Client App.	Initial Server Comp.	Extended Client App.	Extended Server Comp.	Int. System
PROGRAM SUMMARY					
Total SLOC:	125	380	151	408	554
Code Size in kB:	5	12	7	14	21
Time for Initial Parse and Analysis:	:01	:01	:03	:03	:05
Components:	6	5	10	8	11
Total Files:	7	6	12	10	14
Total Procedures:	4	11	21	24	37
Dead Procedures:	0	0	2	2	5
Total Controls:	8	0	8	0	8

Metric	Case Study 8				
	Initial Client App.	Initial Server Comp.	Extended Client App.	Extended Server Comp.	Int. System
CALL GRAPH METRICS					
Total SLOC of Call Graph:	257	735	1304	1668	2629
File Size in kB:	6	16	26	32	50
Time for Global Analysis and Generation:	:01	:02	:03	:04	:08
Total # of ModuleCollections:	1	1	6	5	7
Total # of CallGraphNodes:	4	11	21	24	37
Total # of CalledModules:	0	2	15	20	37
Total # of CallSites:	0	2	20	30	52
Total # of CallSites with parameter mappings:		2	20	30	52
Total # of GlobalRefs:	3	25	3	22	29
Total # of GlobalDefs:	2	17	2	4	6
DATA DEPENDENCE METRICS					
Total # of Parameter Coupling Interfaces:	0	0	0	0	6
Total # of Parameter Coupling Paths:	0	0	0	0	3
Total # of Call Coupling Interfaces:	0	6	40	60	104
Total # of Call Coupling Paths:	0	3	20	30	52
Total # of Parameter Mapping Dependencies:	0	0	0	0	3
Total # of Reverse Ripple Dependencies:	0	2	0	20	43

Table 9. Applying the call graph summary information technique to case study 8

The results of applying the summary call graph analysis to case study 8 is depicted in Table 9. The initial system has one main user application that integrates 6 components, for which the server component listed is the primary. This component in turn integrates with 5 other components. Constructing the summary call graph for the initial main user application shows 4 module collections, 27 call graph nodes, 14 call sites, 3 global referenced variables and 2 global defined variables. There are no data dependencies.

Constructing the summary call graph for the extended main user application shows 6 module collections, 21 call graph nodes, 20 call sites, 3 global referenced variables and 2 global defined variables. For data dependencies, we see 20 call couplings.

Constructing the summary call graph for the integrated system shows 7 module collections, 37 call graph nodes, 52 call sites, 29 global referenced variables and 6 global

defined variables. For data dependencies, we see 3 parameter couplings, 52 call couplings, 3 parameter mapping dependencies and 43 reverse ripples.

Case Study 9											
Metric	Initial System					Extended System					IS
	A	C1	C2	C3	C4	A	C1	C2	C3	C4	
PROGRAM SUMMARY											
Total SLOC:	495	190	1416	794	199	577	218	1416	872	227	2867
Code Size in kB:	19	7	43	34	7	23	10	43	40	10	107
Time for Initial Parse and Analysis:	:03	:01	:07	:04	:01	:07	:04	:07	:10	:04	:23
Components:	11	5	8	7	5	15	6	8	9	6	12
Total Files:	16	6	11	8	6	21	8	11	11	8	22
Total Procedures:	30	8	73	29	9	72	33	73	95	34	155
Dead Procedures:	6	0	1	1	1	32	17	1	44	18	51
Total Controls:	4	0	2	0	0	4	0	2	0	0	4
CALL GRAPH METRICS											
Total SLOC of Call Graph:	2357	619	6113	2987	685	4656	1978	6113	6878	2070	13953
File Size in kB:	55	15	145	84	17	90	36	145	146	38	355
Time for Global Analysis and Generation:	:07	:01	:17	:06	:01	:13	:05	:17	:16	:05	:43
Total # of ModuleCollections:	8	1	6	2	1	13	3	6	5	3	14
Total # of CallGraphNodes:	30	8	73	29	9	72	33	73	95	34	155
Total # of CalledModules:	28	3	69	30	3	45	16	69	67	18	191
Total # of CallSites:	29	4	99	54	4	46	20	99	95	22	251
Total # of CallSites with parameter mappings:	29	4	99	54	4	46	20	99	95	22	251
Total # of GlobalRefs:	115	19	236	105	20	115	19	236	105	20	680
Total # of GlobalDefs:	32	12	75	59	12	32	12	75	59	12	355
DATA DEPENDENCE METRICS											
Total # of Parameter Coupling Interfaces:	26	22	114	196	22	26	22	114	196	22	382
Total # of Parameter Coupling Paths:	13	11	57	98	11	13	11	57	98	11	191
Total # of Call Coupling Interfaces:	58	8	198	108	8	92	40	198	190	44	502
Total # of Call Coupling Paths:	29	4	99	54	4	46	20	99	95	22	251
Total # of Parameter Mapping Dependencies:	13	11	61	104	11	19	11	61	147	11	244
Total # of Reverse Ripple Dependencies:	12	5	20	23	5	12	5	20	23	5	52

Table 10. Applying the call graph summary information technique to case study 9

The results of applying the summary call graph analysis to case study 9 is depicted in Table 10. With respect to the systems listed in the above table, the following abbreviations are used to distinguish the individual systems and components used:

- A: Dispatcher application
- C1: GCSS component
- C2: SendMail component
- C3: MTF component
- C4: WCOP component
- IS: Integrated system

The initial system has one main user application that integrates 11 components, for which there are 4 primary. Constructing the summary call graph for the initial main user application shows 8 module collections, 30 call graph nodes, 29 call sites, 115 global referenced variables and 32 global defined variables. For data dependencies, we see 13 parameter couplings, 29 call couplings, 13 parameter mapping dependencies and 12 reverse ripples.

Constructing the summary call graph for the extended main user application shows 13 module collections, 72 call graph nodes, 46 call sites, 115 global referenced variables and 32 global defined variables. For data dependencies, we see 13 parameter couplings, 46 call couplings, 19 parameter mapping dependencies and 12 reverse ripples.

Constructing the summary call graph for the integrated system shows 14 module collections, 155 call graph nodes, 251 call sites, 680 global referenced variables and 355 global defined variables. For data dependencies, we see 191 parameter couplings, 251 call couplings, 244 parameter mapping dependencies and 52 reverse ripples.

To summarize the results, we examine the average increases in both call graph information and data dependencies across case studies 7 through 9. Figure 38 shows the increase in call graph information in comparison to the initial, extended and integrated systems. The call graph information for this chart represents the total of modules, call graph nodes, call sites, global referenced variables, and global defined variables. For case study 7, there is a 10% increase in information from the initial to the extended states, and an additional increase of 51% from the extended to the integrated states. This represents a 55% increase from the initial to the integrated state. Case study 8 shows an overall 92% increase, and case study 9 shows an 85% increase. On the average, this represents a 77% improvement in the call graph information available to the component user for analysis.

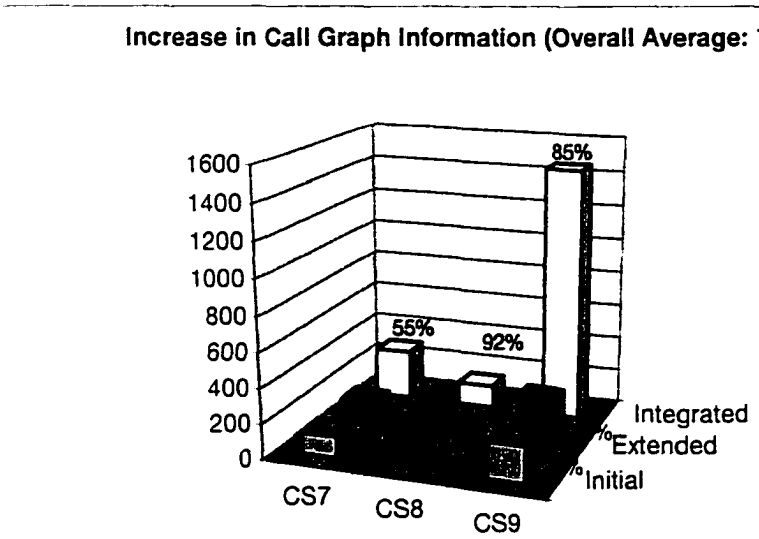

Figure 38. Increase in summary call graph information

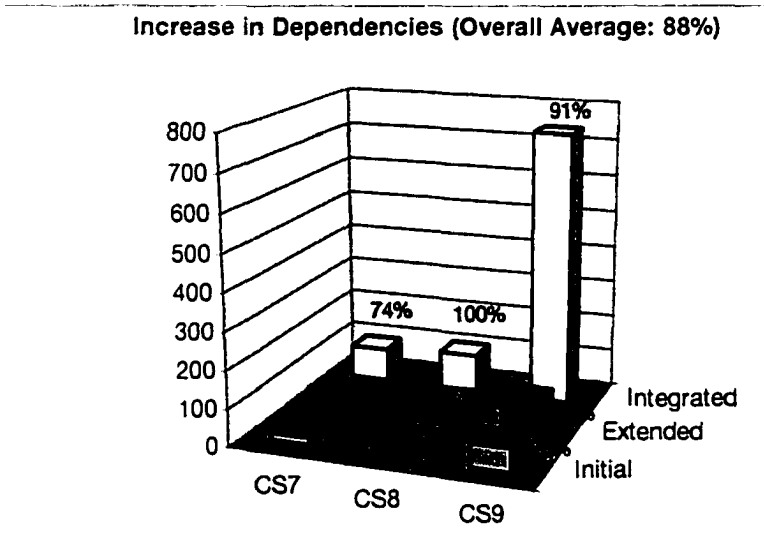

Figure 39. Increase in dependencies

Figure 39 shows the increase in data dependence information in comparison to the initial, extended and integrated systems. The data dependence information for this chart represents the total of parameter couplings, call couplings, parameter mapping

dependencies, and reverse ripple dependencies. For case study 7, there is an 18% increase in information from the initial to the extended states, and an additional increase of 69% from the extended to the integrated states. This represents a 74% increase from the initial to the integrated state. Case study 8 shows an overall 100% increase, and case study 9 shows a 91% increase. On the average, this represents an 88% improvement in the data dependence information that was computed by performing these static analysis techniques on the integrated system.

These results show that the summary call graph technique proved to be very effective for providing component users with the ability to apply some useful static analysis techniques, ultimately gaining more insight into their integrated system. In some cases, such as case study 8 for example, the use of the summary call graphs allowed the static analysis techniques to be performed where previously they were not.

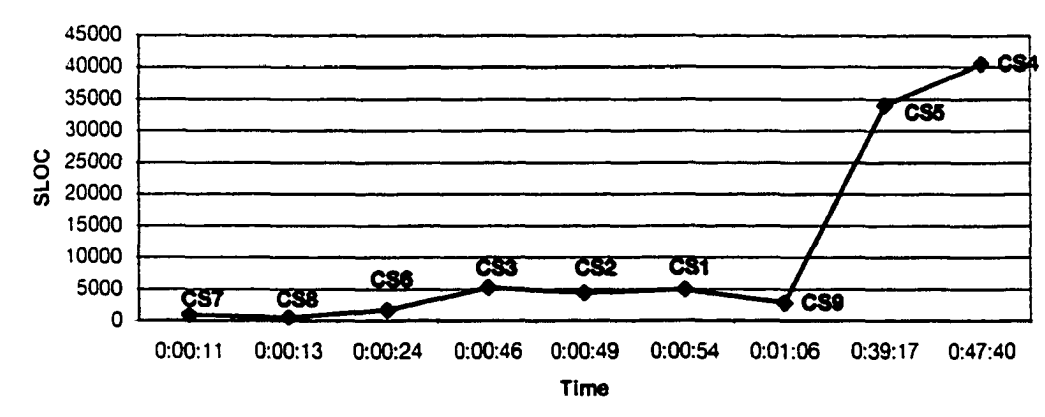

Figure 40. Timing analysis: SLOC vs. analysis time

7.3. Analysis of efficiency

In section 1, we stated that efficiency, reality and usability were three underlying design concerns for this research. With respect to efficiency, performance and storage are important. Using several real systems to obtain the experimental results discussed in this

section have help to address the reality and usability concerns. Here, we will examine efficiency by looking at timing, sizing and algorithm run-time complexity.

Figure 40 depicts the timing analysis for each of the 9 case studies in terms of code SLOC for the integrated systems, and the total time to analyze each system. As can be expected, the total analysis time is a function of the size of the program being analyzed. However, it is also a function of its data complexity. For example, consider case study 9 in the chart. Its code size is slightly less than case study 1, but the time to analyze was greater. This is due to the fact that there are more data dependencies and usage (e.g., global ref-defs) in case study 9, so the global analysis phase would take more time. The SLOC sizes for the 9 systems are all under 45000, and we see from the chart that even for the largest of the systems studied, the total analysis time was under an hour. This time is not unreasonable for the static analysis of a medium-sized system. The majority of the systems averaged around the 4000 SLOC size with an analysis time average of under a minute. We feel that these program sizes will be typical of most component-based systems where the main applications comprise often of smaller portions of 'glue' code. If that is the case, or if the system reaches the 45000 SLOC size, the results here show that the analysis time are practical for real use.

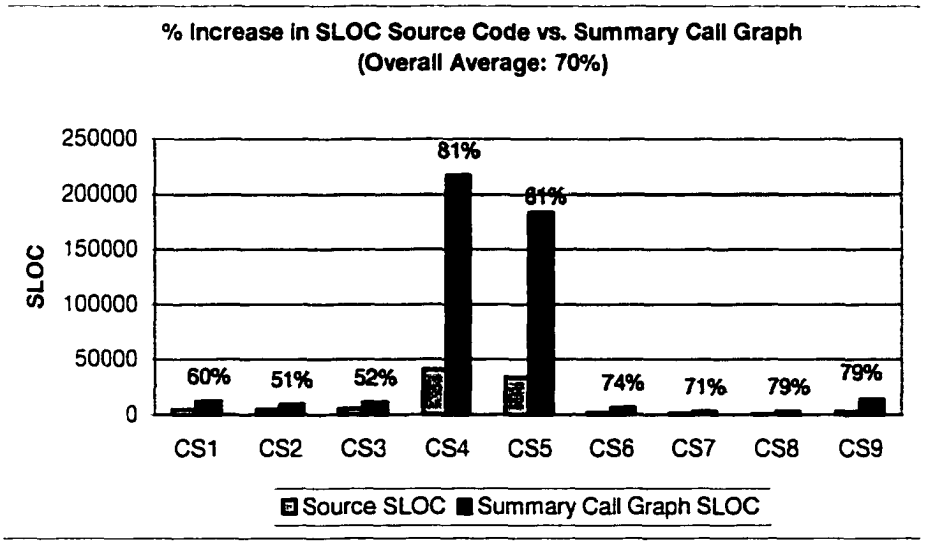

Figure 41. Sizing analysis showing source versus call graph SLOC

To consider storage requirements, Figure 41 shows a sizing analysis of the 9 case studies in terms of program SLOC and call graph SLOC for the integrated systems. It shows the average percentage increase between both SLOC sizes. In all cases, the call graph size is larger than the original program size. The overall average increase is 70%. One reason for the large call graph size is the fact that we use XML. XML files typically average around 10 times the size of its information source due to the strict tagging requirements of the standard [6]. Even so, for the 9 case studies analyzed, the resulting call graph size was manageable.

Another consideration of efficiency is run-time complexity of the algorithms used in the techniques discussed in this research. Three primary algorithms need to be considered closely. The first is the original Project Analyzer analysis. The second is the OA-dead analysis, and finally the global variable analysis and summary call graph generation. The analysis and summary call graph generation do most of the work. Once the information has been computed and stored, the remaining other two analyses, parameter mapping dependence and reverse ripple analysis are simply a series of queries and array traversals to the information stored in memory. In this case, the traversal routines are done in linear time $O(n)$.

The Project Analyzer analysis is done in two phases plus a dead code detection phase. Phase 1 collects the object names. It reads lines of code and examines each word in the line to determine key objects (e.g., variables) and puts them into a table. Phase 2 through all the objects again and compares them to several arrays, one for each object type like procedures, variables, constants, module names an so forth that were constructed in phase 1. If a match is found, it saves cross-reference information. The last phase just traverses through the cross-references and sets a flag every time something is referenced. If we let o be the number of objects (e.g., functions, variables, etc), then both phase 1and phase 2 are $O(o^2)$. The detection phase is $O(o)$.

The OA-Dead analysis is an extension to the dead code detection phase described above. It uses the same two analysis phases as from Project Analyzer to establish the object tables, each phase being $O(o^2)$. An additional object type was established for the

OA-Dead procedures, files and controls and an additional phase was added to inspect each object and compare it to the object properties being examined. This is also $O(o^2)$. The detection and report generation step are $O(o)$ as before.

The extension for the summary call graph technique modified the original Project Analyzer phases 1 & 2 and added an additional phase. As discussed above, during phases 1 & 2 Project Analyzer is gathering data about project structure (e.g., variable and constant declarations, procedure names, identifier usage, etc.). Phase 2 primarily conducts cross-referencing analysis. The phase 2 analysis was modified to capture procedure call statements and mark each identifier serving as an actual parameter as a new identifier reference (IdentRef) object. The new IdentRef object is added to the IdentRef array and further parameter-binding analysis is performed in a new phase 3. As before, phase 1 and 2 are $O(o^2)$. Phase 3 was added to support call graph and data flow analysis. It performs two main tasks. The first task is to construct a system call dependence graph and finalize the parameter-mapping analysis. The parameter-binding relation is determined to be either input (reference) or output (definition). Parameter-binding information is then copied into the ParmBindings table in the CallGraph database. The second task is to copy the IdentRef data into the RefDef table in the database. The summary call graph analysis is based on the system dependence graph defined by [65]. If N_p and E_p denote the number of vertices and edges in a control flow graph of a procedure p, then the construction of the procedure graphs takes time $O(\Sigma p(E_p \times N_p))$.

8. FUTURE WORK

This section identifies several areas in which further work is needed. In particular, we provide recommendations for further study related to static analysis, dynamic analysis and strategic component issues in general.

8.1. Additional work related to static analysis

In Section 5.1, we identified summary information that could be used to support static analysis techniques. In the techniques we developed, we focused on type library information, call-level control flow, global variable ref-def analysis, variable first-use and last-use analysis, and parameter mapping relationships. We defined a way to represent this information using a standards-based XML format, and extended an analysis tool to conduct the supporting analyses for programs and components written in Visual Basic.

Adding support for the other types of summary information we identified in Section 5.1 could allow other types of static analyses to be performed, such as partial evaluation. We feel the following types of summary information and subsequent analysis techniques would be good candidates for follow-on research:

- Statement-level control flow graph information. Statement-level control flow information could support various statement level analysis techniques, such as program slicing, ripple analysis, partial evaluation, and testing.

- Component states. Identification of the various states a component can be in, as well as the dependence information between input, output and state variables could be useful in supporting techniques like: usage patterns, component wrappers, interface slicing and integration testing [38].

- Exception information. Exception information and its dependence to input and state variables could support techniques like: usage patterns, component wrappers, interface slicing and integration testing.

One particular technique that we feel is interesting that could benefit from the above summary information is the usage analysis of a component-based system. Usage analysis is based on identifying usage patterns for a component [118], and can be useful for gaining a better understanding of a component, especially one that was built for a generic purpose with many capabilities that is being used for a specific purpose requiring a limited set of those capabilities. For example, a component user could decide which capabilities of the generic component are necessary, and using techniques like those developed in this research, could gain more confidence in the data dependencies and impacts associated with a particular usage; particularly important to testing. Also, the component user may decide to construct a wrapper around the component, similar to the one shown in case study 8, which effectively hides all the capabilities of the generic component not associated with the intended usage [99].

Support for usage analysis can be provided in a number of forms. In a basic sense, the first technique in our research and its subsequent application to OA-dead analysis can be thought of as identifying a particular usage of a system when the analysis was performed. The additional summary information concerning component states and exceptions can be used to provide a more precise usage analysis. As an example, a state representation of a component can be defined with the transitions between states determined by the method calls on the component. Usage coverage metrics that could be used may include, all states visited, all non-repeating paths through the state graph, etc. The paths through the state graph define sequences of component usage. Below, we discuss usage analysis based on dynamic techniques.

With respect to the analysis tools, we focused on the analysis of Visual Basic programs. It would be useful to develop similar tools for other languages that are commonly used to construct components and component-based systems, such as C++, and Java.

8.2. Additional work related to dynamic analysis

In our research, we focused on static analysis techniques. In contrast, dynamic analysis techniques are very precise as they monitor and analyze a program as it executes.

The component provider summary information concept could be extended to support dynamic analyses, and assessing their effectiveness and utility would be of value. For example, in lieu of a component provider distributing static summary call graphs with their component, they could provide some sort of dynamic interface for this information that can be queried by the user during a specific analysis. This may be considered a form of augmenting components, and some possible techniques for doing this may be:

- Abstract state machine (ASM). Partitions component state into several abstract states that characterize distinct regions of the state space – a form of domain analysis. Dynamic analysis can use this as a coverage metric or monitor the execution of a component: to validate usage states match expected states.

- Usage analysis. Characterization of the way a component is being used. One possible way to describe usage patterns is as regular expressions over the ASM. Dynamic analysis can use this as a coverage metric to test against anticipated usage patterns, or monitor the execution of a component: to validate usage states match expected states.

- Gray-box interface. Provides access to components internal state, either in the form of direct access to private data variables or to state of ASM. Dynamic analysis allows pseudo white-box testing of components that can be used for state or usage based coverage metrics. Also can support monitoring of the component to validate valid states and usage.

- Exception Triggering Interface. Forces component to raise its exceptions. May be only practical way to test exception-handling code. Dynamic Analysis can use this for coverage of exception handling code to support integration testing.

8.3. Additional work related to strategic component issues

In addition to future research in both static and dynamic analysis techniques for component-based systems, we feel that there are several general areas of further work to enhance the development, analysis, maintenance and testing of components and the systems that use them. Some of these include:

- Obtaining buy-in from component vendors. We have demonstrated the effectiveness of the summary call graph. In order for such an approach to be effective in the large, component developers must be willing and able to analyze

their components with minimal risk and effort, and make the graphs available with their components. Effective ways to get component developers to do this need to be identified.

- Modifying existing component interface standards. Another way to ensure that the component summary information is available to users is to modify existing component object model standards, such as COM or CORBA, to include the specification of this information. This seems like a reasonable activity to investigate.

9. CONCLUSION

This research studied ways to extend traditional static analysis techniques to support the development, maintenance and testing of a component-based system. Throughout this work, four primary goals were achieved and the results of each documented in this thesis. These were to:

- Understand and document characteristics and potential issues associated with component-based applications which can make software analysis, development, maintenance and testing more difficult.
- Develop new or extend traditional static analysis techniques for improved analysis of a component-based software system.
- Demonstrate that the use of additional information, such as semantic information about component properties, can be used to improve the quality of analyses. Some of this additional information can only be provided by experienced developers, and some can be extracted automatically.
- Validate the techniques on existing real systems.

We identified situations where standard analysis techniques provide misleading or incomplete information when used on a component-based solution; and showed how traditional static analysis techniques can be extended in a number of ways to analyze component-based systems.

The first technique developed leverages the minimal information that is typically available for components but for which traditional static analysis techniques often do not utilize. This approach aids component users attempting to analyze a component-based system by leveraging the semantic information about component properties contained in the type library or interface definition language (IDL) files associated with a component. We then showed how this information could be used to augment traditional static analysis techniques for analyzing a typical component-based system by enhancing a dead code

detection technique to discover OA-dead code that may be typically found in visual programming languages, such as MS Visual Basic.

While this technique was useful, we discussed additional ways to improve the analysis of component-based systems. Our second technique developed represents one such way. For example, a component provider could provide extended static analysis summary information about their component. This extended interface may include the standard interface information, but also other information that would be useful for gaining insight into the component without having access to the source code. In this approach, the component provider uses analysis techniques to gather summary information that facilitates further analysis and testing of those components by users without requiring access to the source code. The component provider makes the summary information available with the component. The component user then integrates the component summary graphs with the graph for their user application to produce an integrated system graph for analysis. Our technique summarizes global data flow analysis through variable def-use, first use/last use and parameter couplings. This technique can be useful for techniques such as interface-level coupling analysis and testing, ripple analysis, and integration testing.

We then demonstrated the effectiveness of these techniques on several case studies. Seven case studies represent real COTS component-based systems developed and maintained by the Department of Defense (DoD). Two additional case studies are used to represent academic examples designed to illustrate some interesting aspects of component-based development.

The results of applying the first technique show that the OA-dead analysis proved to be an effective method for discovering additional procedures and controls that were in a state when the analysis was performed where they could not be executed. For example, an average increase of 15% of OA-dead procedures was discovered. This is important because it may significantly reduce the level of maintenance and testing on these systems.

The results of applying the second technique show that the summary call graph information that a component developer can provide to the component user can greatly increase the user's ability to analyze and understand the integrated system. On the

average, there was a 77% improvement in the call graph information available to the component user for analysis. Using this additional information, the user can now apply static analysis techniques on the integrated system. The results show that on the average, there was an 88% improvement in the data dependence information that was computed by performing these static analysis techniques on the integrated system.

In terms of efficiency, the results show that most of the systems were around 5000 SLOC in size, and the analysis time for these systems was under a minute. Two of the case studies were over 30000 SLOC, with the time of analysis still under an hour. We feel that typical component-based applications will be within this size range, and as such the results here show that the analysis times are practical for real use.

Overall, the results of this experimentation show that these techniques can effectively be used to improve the analysis of a component-based system. This was the case for the 9 case studies examined. Based on our work in this area, we can make several observations:

- The traditional maintenance problems are still present in component-based systems.

- Component-based solutions, and in particular the visual languages used to create them, appear to increase the presence of unreachable code as software maintenance evolves.

- Semantics can be used to supplement traditional static analysis approaches in significant ways to increase the precision and accuracy of the results from analyzing component-based software.

- A subject matter expert on the component-based system being analyzed can help identify several key pieces of semantic information associated with the component-based architecture which would help to promote a better understanding of the overall system or enhance a particular analysis capability.

- The availability of additional static analysis information for a component is necessary if component users want to apply techniques, such as coupling analysis and ripple analysis, effectively across the entire integrated application.

Our research shows that the use of traditional static code analysis techniques can aid in the understanding of unfamiliar code and in monitoring potential side effects that can be caused by modifications to source code. It has application for development, analysis, debugging, testing, and maintenance. We feel that the techniques reported here are promising and can be used to help narrow the gap between the information available today for black box components and better ways to provide more useful information to component users to help analyze and test component-based systems. The work reported here is especially important since component-based solutions are becoming a widely used development technique in software engineering.

REFERENCES

[1] Active X Documenter Tool, vbaccelerator.com, http://vbaccelerator.com/, 2000.

[2] LotusXSL Tool, IBM Alphaworks, http://www.alphaworks.ibm.com, 2000.

[3] Microsoft Visual Studio Book Sale Component Example, Microsoft, http://www.microsoft.com, 2000.

[4] MS Visual Studio, Microsoft, http://www.microsoft.com, 2000.

[5] OLE Viewer, Microsoft, http://www.microsoft.com, 2000.

[6] Worldwide Web Consortium, http://www.w3.org/TR/, 2000.

[7] V. K. Agarwal and R. Nadhani, Displaying Charts on Thin Clients with MS Excel on the Web Server, ASP Today, http://www.asptoday.com/articles/20000303.htm, 2000.

[8] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing," *ACM SIGPLAN Notices*, vol. 25, pp. 246-256, 1990.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques and Tools*. Reading, Mass: Addison-Wesley, 1986.

[10] P. Aitkin, *Visual Basic 5 Programming Explorer*: Coriolis Group Books, 1997.

[11] T. Albertson, Best practices in distributed object application development: RMI, CORBA and DCOM, developer.com, http://www.developer.com/journal/techfocus/022398_dist1.html, 1998.

[12] F. E. Allen, "Interprocedural Data Flow Analysis," in *Proceedings 1974 IFIPS Congress*: North Holland, Amsterdam, 1974, pp. 398-402.

[13] S. Ambler, *Process Patterns: Delivering Large-Scale Systems Using Object Technology*: SIGS Books: Cambridge University Press, 1998.

[14] L. O. Anderson, "Program Analysis and Specialization for the C Programming Language," in *DIKU*. Denmark: University of Copenhagen, 1994.

[15] R. S. Arnold, "Software Impact Analysis Seminar Notes," *Conference on Software Maintenance*, 1993.

[16] R. D. Banker, M. D. Srikant, C. F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Cost," *Communications of the ACM*, vol. 36, 1993.

[17] S. a. S. H. Bates, "Incremental program testing using program dependence graphs," *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, 1993.

[18] P. Benedusi, A. Cimitile, and U. De Carlini, "Post-maintenance testing based on path change analysis," *Proceedings of the Conference on Software Maintenance*, pp. 352-61, 1988.

[19] R. V. Binder, "The FREE Approach to Testing Object-Oriented Software: An Overview.," RBSC Corporation 1995.

[20] R. Bodik and R. Gupta, "Partial Dead Code Elimination using Slicing Transformations," *Proceedings of the ACM SIGPLAN Conference on Programming Language and Design Implementation*, vol. 32, pp. 159-170, 1997.

[21] A. Brown, *Component-Based Software Engineering*: IEEE Press, 1997.
[22] M. Buchi and W. Weck, "A Plea for Grey-Box Components," Turku Centre for Computer Science, Lemminkaisenkatu, FIN TUCS No 122, August 1997.
[23] R. W. Butler and G. B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Transactions on Software Engineering*, vol. 19, pp. 3-12, 1993.
[24] U. Buy, C. Ghezzi, A. Orso, M. Pezze, and M. Valsasna, "A Framework for Testing Object-Oriented Components," presented at First International ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, 1999.
[25] D. Callahan, "The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Anaylsis," *SIGPLAN 1988 Conference on Programming Language Design and Implementation Atlanta, GA (June 22-24, 1988)*, vol. 23, pp. 47-56, 1988.
[26] D. Chapel, *Understanding ActiveX and OLE: A Guide for Developers & Managers*: Microsoft Press, 1996.
[27] R. Cherinka, "Maintaining a COTS Component-based Solution-Can Traditional Static Analysis Techniques be useful for this new programming methodology?," presented at International Workshop on Component Programming, 1998.
[28] R. Cherinka, "Maintaining a COTS Integrated Solution- Are traditional static analysis techniques sufficient for this new programming methodology?," presented at IEEE International Conference on Software Maintenance (ICSM98), 1998.
[29] R. Cherinka, "Ripple Analysis Using Data Flow Techniques," Old Dominion University, Norfolk, VA 23529-0162 Master's Project Report, May 1991.
[30] R. Cherinka, M. Cokus, G. Cox, and J. Ricci, "Air Force JINTACCS Configuration Management Environment (AFJCME) Strawman Architecture and Automation Requirements," *MITRE Technical Report*, 1997.
[31] R. Cherinka and C. M. Overstreet, "Using Data Flow Analysis to Determine Bi-Directional Ripple Effects During Software Maintenance," Old Dominion University, Technical Report, 1993.
[32] R. Cherinka and J. Ricci, "Preparing and Organziation for Process Automation: A Baseline Assessment of an Integrated Software Engineering Environment (ISEE) Initiative at NSWC PHD/ECO," *Proceedings of the 1996 MITRE Conference on Software Engineering and Economics*, 1996.
[33] R. Cherinka, J. Ricci, and D. Finney, "Reducing The Cost of Regression Testing and Maintenance for a Component-based System," The MITRE Corporation, Hampton WN980000109, September 1998.
[34] R. Cherinka, C. Wild, J. Ricci, and C. M. Overstreet, "Issues and Approaches in Testing Mission-Critical Cystems COTS Systems," presented at International Conference on Software Testing, 2000.
[35] K. D. Cooper and K. Kennedy, "Efficient Computation of Flow-Insensitive Interprocedural Summary Information - A Correction," *ACM SIGPLAN Notices*, vol. 23, pp. 35-42, 1988.

[36] B. Cottman, A. O'Toole, M. Ryland, and R. Soley, "Web, Distributed Object Management and Component Software," *Information and Knowledge Management*, pp. 341, 1996.

[37] O. Danvy, R. Gluck, and P. Thiemann, "Partial Evaluation International Seminar, Dagstuhl Castle, Germany, February 1996: selected papers," *Lecture Notes in Computer Science*, vol. 1110, 1996.

[38] B. P. Douglass, "Components, States and Interfaces, Oh My!," *Software Development*, vol. 8, pp. 56, 2000.

[39] G. Eddon and H. Eddon, "Inside Distributed COM: Type Libraries and Language Integration," 1998.

[40] J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319-349, 1987.

[41] J. Field, G. Ramalingham, and F. Tip, "Parametric Program Slicing," *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pp. 379-392, 1995.

[42] I. Forgacs and T. Gyimothy, "An efficient interprocedural slicing method for large programs," Informatics Lab., MTA SZTAKI TR97-7, June, 1997.

[43] R. Forgacs and A. Bertolino, "Feasible Test Path Selection by Principal Slicing," *Lecture Notes in Computer Science*, vol. 1301, pp. 378, 1997.

[44] B. Francis, "Component Programming 1," presented at Wrox Web Developer's Conference, Washington, USA, 1999.

[45] B. Francis, "Component Programming 2," presented at Wrox Web Developer's Conference, Washington, USA, 1999.

[46] P. G. Frankl and R.-K. Doong, "Case studies on testing object-oriented programs," *Proceedings of the 4th Symposium on testing, Analysis, and Verification*, 1991.

[47] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, pp. 751-761, 1990.

[48] W. M. Gentleman, "Effective use of COTS Software Components in long lived systems," *Proceedings of the 1997 International Conference on Software Engineering*, pp. 635-636, 1997.

[49] D. George, "Building Automation Service Components in Visual Basic and Visual C++," October, 1995.

[50] S. Ghosh and A. P. Mathur, "Issues in Testing Distributed Component-Based Systems," Purdue University, West Lafayette April, 1999.

[51] C. Goswell, "The COM Programmer's Cookbook," September, 1995.

[52] M. Grossman, "Component Testing," *Proceedings of the International Workshop on Component-Oriented Programming*, 1998.

[53] R. Gupta, M. Soffa, and J. Howard, "Hybrid slicing: integrating dynamic information with static analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 370-397, 1997.

[54] R. Hall, "Automatic Extraction of Executable program subsets by simultaneous dynamic program slicing," *Automated Software Engineering*, vol. 2, pp. 33-53, 1995.

[55] M. Harrold, D. Liang, and S.Sinha, "An Approach To Analyzing and Testing Component-Based Systems," presented at IEEE International Conference on Software Engineering, Los Angelos, CA, 1999.

[56] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object-oriented class structures.," *Proceedings of the 14th ICSE*, 1992.

[57] M. J. Harrold, J. McGregor, and K. Fitzpatrick, "Incremental Testing of Object-Oriented Class Inheritance Structures," presented at 14th International Conference on Software Engineering, 1992.

[58] M. J. Harrold and G. Rothermel, "Performing dataflow testing on classes," Clemson University, Technical Report 94-108, June, 1994.

[59] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Transactions on Programming Languages and Systems*, 1994.

[60] M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance.," *Proceedings of the Conference on Software Maintenance*, 1988.

[61] J. Hartmann and D. J. Robson, "Techniques for Selective Revalidation," *IEEE Computer*, vol. 7, pp. 31-36, 1990.

[62] D. M. Hoffman and P. A. Strooper, "Graph-based class testing," *Proceedings of the 7th Australian Software Engineering Conference*, 1993.

[63] T. Hoffner, M. Kamkar, and P. Fritzson, "Evaluation of Program Slicing tools," *2nd International Workshop on Automated and Algorithmic Debugging*, pp. 51-69, 1995.

[64] S. Horwitz, J. Prins, and T. Reps, "On the Adequacy of Program Dependency Graphs Representing Programs," presented at Proceedings of the 15th ACM Annual Symposium on Principles of Programming Languages, 1988.

[65] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs.," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26-60, 1990.

[66] W. E. Howden and E. Miller, "A Survey of Static Analysis Methods," presented at Software Testing and Validation Techniques, 1981.

[67] B. Jeng and I. Forgacs, "On the automatic generation of domain test data," Informatics Lab., MTA SZTAKI, Technical Report TR98-1, April , 1998.

[68] Z. Jin and A. J. Offutt, "Coupling-based Criteria for Integration Testing," George Mason University July, 1998.

[69] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*: Prentice Hall, 1993.

[70] C. Kindel, "Designing COM Interfaces," October, 1995.

[71] C. Kindel, "OLE Component Design Issues," October, 1994.

[72] J. Knight, A. Cass, A. Fernandez, and K. Wika, "Testing A Safety-Critical Application," University of Virginia CS-94-08, Feb., 1994.

[73] J. Knight, R. Lubinsky, J. McHugh, and K. Sullivan, "Architectural Approaches to Information Survivability," CS-97-25, Sept., 1997.

[74] D. Kung, J. Gao, Y. Toyoshima, C. Chen, Y. Kim, and Y. Song, "Developing an Object-oriented Software Testing and Maintenance Environment," *Communications of the ACM*, vol. 38, pp. 75-87, 1995.

[75] W. A. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," *Proceedings of SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[76] L. Larsen and M. Harrold, "Slicing object-oriented software," *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pp. 495-505, 1996.

[77] J. Laski and W. Szermer, "Identification of program modifications and its application in software maintenance," *Proceedings of the Conference on Software Maintenance*, 1992.

[78] H. K. N. Leung and L. J. White, "A cost model to compare regression test strategies," *Proceedings of the Conference on Software Maintenance*, 1991.

[79] H. K. N. Leung and L. J. White, "A study of integration testing and software regression at the integration level," *Proceedings of the Conference on Software Maintenance*, 1990.

[80] D. Liang and M. J. Harrold, "Slicing Objects Using System Dependence Graph," presented at International Conference on Software Maintenance, Washington, D.C., 1998.

[81] J. R. Lyle, "Evaluating Variations on Programming Slicing for Debugging," University of Maryland, Maryland 1984.

[82] A. Major, *COM IDL & Interface Design Effective Object modeling for efficient COM applications*. Birmingham, UK: Wrox Press, 1999.

[83] B. Marick, "How Producers of Reusable Software Can Help Their Users Test," 1999.

[84] B. Marick, "*Notes on Object-Oriented Testing Part 1: Faults Based Test Design*," 1999.

[85] B. Marick, "*Notes on Object-Oriented Testing Part 2: Scenario-Based Test Design*," 1999.

[86] R. McDaniel and J. McGregor, "Testing the Polymorhphic Interactions between Classes," Clemson University, Clemson, SC.

[87] J. D. McGregor, "Constructing Functional Test Cases Using Incrementally Derived State Machines," presented at Proceedings of Eleventh International Conference on Testing Computer Software, 1994.

[88] F. Mercier, P. L. Gall, and A. Bertolino, "Formalizing integration test strategies for distributed systems," presented at First International ICSE Workshop onTesting Distributed Component-Based Systems, Los Angeles, CA, 1999.

[89] T. Niemann, "Nuts to OOP!," in *EMbedded Systems Programming*, 1999, pp. 16-28.

[90] T. J. Ostrand and E. J. Weyuker, "Using dataflow analysis for regression testing," *Proceedings of the 6th Annual Pacific Northwest Software Quality Conference*, 1988.

[91] K. Ottenstein and L. Ottenstein, "High Level Debugging Assitance via Optimizing Compiler Technology," presented at ACM SIGPLAN Notices, 1983.

[92] D. L. Parnas, "Software Aspects of Strategic Defense Systems," *CACM*, vol. 28, pp. 1326-1335, 1985.

[93] H. Patil, "Efficient Program Monitoring Techniques," University of Wisconsin, Madison, Technical Report CS-TR-96-1320, July, 1996.

[94] T. Pattison, "COM+ Overview for Microsoft Visual Basic Programmers," February, 2000.

[95] D. E. Perry and G. E. Kaiser, "Adequate testing and object-oriented programming," *Journal of Object-Oriented Programming*, vol. 2, pp. 13-19, 1990.

[96] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," *Principles of Programming Languages*, pp. 49, 1995.

[97] T. W. Reps, "Demand Interprocedural Program Analysis Using Logic Databases," University of Wisconsin-Madison, Madison.

[98] S. Robinson and A. Krassel, "COMponents," August, 1997.

[99] D. Rogerson, "Calling COM Objects with Interface Wrappers," October, 1995.

[100] G. Rothermel and M. J. Harrold, "A framework for evaluating regression test selection," *Proceedings of the 16th ICSE*, 1994.

[101] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," *Proceedings of the Conference on Software Maintenance*, 1993.

[102] G. Rothermel and M. J. Harrold, "Selecting regression tests for object-oriented software," Clemson University, Technical Report 94-104 1994.

[103] G. Rothermel and M. J. Harrold, "Selecting regression tests for object-oriented software," *Proceedings of the International Conference on Software Maintenance*, 1994.

[104] G. Rothermel and M. J. Harrold, "Selecting tests and identifying test coverage requirements for modified software," *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, 1994.

[105] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression Test Selection for C++ Software," Oregon State University, 99-60-01, January, 1999.

[106] J. A. Rubin, "Component-Based Testing," The MITRE Corporation, Technical Report WN97B0000059 1997.

[107] E. Ruf, "Partitioning dataflow analysis using types," *Principles of programming languages*, 1997.

[108] I. Salmre, "Building COM Components That Take Full Advantage of Visual Basic and Scripting," February, 1998.

[109] T. Salste, Project Analyzer, Aivosto Oy Software Company, http://www.aivosto.com/vb.html, 1998.

[110] A. Sloane and J. Holdsworth, "Beyond traditional program slicing," *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pp. 180-186, 1996.
[111] M. D. Smith and D. J. Robson, "A framework for testing object-oriented programs," *Journal of Object-oriented Programming*, vol. 5, pp. 45-53, 1992.
[112] C. Solomon, *Microsoft Office 97 Developer's Handbook*: Microsoft Press, 1997.
[113] R. Sparks, "Semi-Dead Code Analysis of Visual Basic," Old Dominion University, Norfolk Master's Project Report, July, 2000.
[114] K. a. K. M. Spencer, *Client/Server Programming with Microsoft Visual Basic*: Microsoft Press, 1996.
[115] D. Stearns, "The Basics of Programming Model Design," June, 1998.
[116] C. Steindl, "Intermodular Slicing of Object-Oriented Programs," Johannes Kepler University Linz, Austria, Technical Report CS-SSW-P98-03, March, 1998.
[117] C. Steindl, "Program Slicing Data Structures and Computation of Control Flow Information," Johannes Kepler University Linz, Linz Report 11, March, 1998.
[118] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*: Addison-Wesley, 1998.
[119] A. B. Taha, S. M. Thebaut, and S. S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," *Proceedings of the 13th Annual International Computer Software and Applications Conference*, 1989.
[120] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121-189, 1995.
[121] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, "Flow insensitive C++ pointers and polymorphism analysis and its application to slicing," *Software Engineering*, pp. 433-443, 1997.
[122] C. D. a. D. J. R. Turner, "The state-based testing of object-oriented programs," *Proceedings of the Conference on Software Maintenance*, 1993.
[123] G. Venkatesh, "Experimental results from dynamic slicing of C programs," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 197-216, 1995.
[124] J. Voas, "Certifying Off-the-Shelf Software Components," *IEEE Computer*, 1998.
[125] J. Voas, "A Defensive Approach to Certifying COTS Software," Reliable Software Technologies Corporation, RSTR-002-92-002.01, August, 1997.
[126] J. Voas, "Defensive Approaches to Testing Systems that Contain COTS and Third Party Functionality," presented at 15th International Conference and Exposition on Testing Computer Software, 1998.
[127] J. Voas, G. McGraw and A. Ghosh, "Gluing together software components: How good is your glue?," *Proceedings of the Pacific Northwest Software Quality Conference*, 1996.
[128] J. Voas, "This Decade's Eight Greatest Myths About Software Quality," *IEEE Software*, 1999.
[129] J. Voas, "Upgrading Software Maintenance for Components," *IEEE Software*, 1998.

[130] J. Voas, F. Charron, and K. Miller, "Tolerant Software Interfaces: Can COTS-based Systems be Trusted Without Them?," presented at 15th International Conference on Computer Safety, Reliability & Security (SAFECOMP'96), Vienna, Austria, 1996.

[131] J. Voas and L. Kassab, "Using Assertions to Make Untestable Software More Testable," , 1998.

[132] J. Voas and J. Payne, "Dependability Certification of Software Components," RST, Technical Report, 1999.

[133] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 352-357, 1984.

[134] E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Transactions on Software Engineering*, vol. 12, pp. 1128-38, 1986.

[135] L. J. White and H. K. N. Leung, "A firewall concept for both control-flow and data-flow in regression integration testing," *Proceedings of the Conference on Software Maintenance*, 1992.

[136] S. Williams and C. Kindel, "The Component Object Model: A Technical Overview," October 1994.

[137] T. Williams, "*Component Based Applications*," presented at International Conference on Software Reuse (ICSR '98), 1998.

[138] M. Winter, "Managing Object-Oriented Integration and Regression Testing," presented at euroSTAR 98, 1998.

APPENDIX A. A COMPONENT TAXONOMY

This table represents a more comprehensive laundry list of specific issues that are associated with component-based analysis, testing and component-based systems in general. It is intended that this list be a quick reference to understanding the issues. Each issue is discussed by examining a number of viewpoints:

- **Class** – This classifies the issue with respect to the categories we have defined in this paper.
- **Lifecycle** – This describes which software engineering lifecycle an issue impacts.
- **Issue** – This is a description of the issue.
- **Impact** – This describes the potential impact the issue may have if not addressed.
- **Example** –This is an example of an occurrence of the issue.
- **Mitigation** – This attempts to provide a general solution to addressing the issue.
- **Tools** – This identifies specific tools and/or techniques which may be used to address the issue.
- **Solution Risks** – This describes any potential risks associated with using the solutions describe in the Mitigation and Tools section.

In addition, the following categories are used in the below table:

CLASS (CL):
1. Basic Object Testing
2. Components and Larger Collections of Objects & Patterns
3. COTS Testing
4. Integration Testing
5. Distributed Systems
6. Design Principles in Component-based Development/Maintenance
7. Regression Testing
8. Event-based Programs
9. Non-Technical Issues

LIFECYCLE (LC):
1. Requirements Management
2. Project Management
3. Configuration Management
4. Design
5. Development
6. Testing
7. Quality Assurance
8. All

OBJECT MODEL (OM):
1. COM/DCOM
2. CORBA
3. Java
4. All

CL	LC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
1	6	4	Objects have state.	Existence of Object state complicates testing. Many traditional testing methodologies are stateless and are not adequate to test an internal and possible unknown internal object state.	Initialized vs. un-initialized objects. Reset of navigational components – how to test for reasonable initialization if we don't have access to the state?	Define instrumentation interface to access encapsulated state to ease testing burden.	Gray box testing: *component developer* provides access to state through gray box.	Combinatorial explosion *in number of test cases.* Increased effort and need for access to proprietary information may preclude cooperation from developers. No such information available for legacy components
1	6	4	Polymorphism	Polymorphism allows dynamic selection of functions. Function need not have a semantic relationship. *Even if not* overriding inherited function - may need to retest because it calls an overridden function - static analysis can help here	Polymorphism in Object-oriented systems complicates testing because *actual function* called is determined dynamically	Disallow virtual functions, inherit test cases, and reason about subsumption of test *results. Trace class* hierarchy to test all possible instantiations of a polymorphic function. Alert tester whenever a new subclass with virtual functions is defined. Reuse base class testing specifications.	Reference McDaniel & McGregor [86, 87]	Undeveloped/untried solutions
1	6	4	OO style of programming favors smaller functions but more integration issues	Unit testing is better understood than integration testing	Proliferation of Modifier and Selector functions to give access to state variables. *Get and* Set Property functions	Object style lessens integration issues. New integration testing strategies are being *developed*	Static Analysis with extended component interfaces	Undeveloped/untried solutions
1	6	4	Object style favors objects as arguments, the state of an object	Test coverage will be less rigorous	Passing a spreadsheet object for inclusion in a compound document	Analyze object to define critical states. Use Pattern languages to characterize states	Simplify the analysis of state using assertions	No well defined way to define the optimal level of abstraction in a given context

CL	LC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
			is arbitrarily complex. This complicates testing techniques based on domain analysis.					
1	6	4	Objects support encapsulation favoring black box approaches to testing	Black box approaches have had poor coverage results.	Use of MS Office products where there is no access to the source code, but object interface specifications are published.	Gray box, developer provided testing specifications, formal object specification provided.	Gray box testing: component developer provides access to state through gray box.	Increased effort and need for access to proprietary information may preclude cooperation from developers.
1	6	4	Object style favors structured exception handling. Developers of reusable objects will likely provide exception classes.	Code handling exception cases usually undertested underexperienced – could contain fatal flaws.	Current Java specification makes extensive use of exception classes. Example: Socket exceptions for inability to connect, etc. There is a trend in COM toward structured exception handling.	Require testing specifications to cover exceptional cases. Simulate hard to provide errors (e.g., out of memory) – possibly by forcing code down error path, calling exception.	Gray box testing: component developer provides testing interface for exceptions or test requirements for forcing individual exceptions..	Developers are not motivated to provide these facilities. Legacy systems don't have them.
7	6	4	Regression Testing of Component Updates	Full regression testing for all component updates may be required but is too expensive and time consuming to be practical	New version of a component gets injected into a mission-critical system – calls for revalidation of the entire system	During design, ensure that components are isolated to the maximum extent possible so that regression testing can be focused. Coordinate early and often with the operational test community to build confidence in risk based	Static Analysis with extended component interfaces	There is a danger of making invalid simplifying assumptions, e.g., overlooking a critical test because there is no obvious interaction

CI	LC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
						component testing.		
9	6	4	Government/Contractor Test Team Disconnects	Discontinuities between the government and contractor test teams, particularly with respect to the scope of early operational testing		Ensure early, regular coordination – nurture a strong IPT at the working level through frequent meetings/telecons. Get procedures/process documented.	Organizational structures and procedures	Requires coordinated support from both organization and contractor. Constant oversight and buy-in by contractors.
9	6	4	Testing Process Disconnects	Disconnects between UPC and integration contractor, particularly with respect to interface definition and testing and integration test responsibilities		Ensure early and regular coordination meetings between the parties, build and maintain clear documentation supporting third party developers, and reinforce the roles and responsibilities of the integration contractor (and be prepared to fund them). Get procedures/process documented.	Organizational structures and procedures	Requires coordinated support from both organization and contractor. Constant oversight and buy-in by contractors.
9	6	4	Feedback Impediments	Lack of free flow of information (e.g., requirements) due to technical impediments, cost, data ownership issues, etc		Finalize procedures and responsibilities early. Emphasize accountability and feedback loops. Make the system scalable/adaptable to increasing demands.	Organizational structures and procedures	Requires coordinated support from both organization and contractor. Constant oversight and buy-in by contractors.
9	1	4	Testing Requirements Traceability	Traceability of requirements can be lost, particularly with respect to operational		Build and maintain good historic records of requirements signoff/modifications	DOORS requirements management system. Organizational structures and	Ensuring currency of information. Failure to maintain the databases as systems change.

C L	L C	O M	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
				limitations/assumptions that will impact test		and unique test criteria/methods – presumably in the DOORS database.	procedures	
5	6	4	Distributed Operational Environment Testing	Complexity of testing in a distributed environment which mirrors operational environment.	System certification testing requirements	Test on the operational environment or provide functional facsimile at test site	Use of tools that simulate Joint Services in a certification test. Network analyzer tools	Fidelity of the facsimile is an issue; likewise, the logistical complexity and coordination factors associated with live testing. Traceability in a complex testing environment
5	3	4	Distributed Versioning	Version control of software/hardware and platform in a distributed environment to ensure applicability of testing results to operational use.	Components using upgraded versions of Microsoft libraries may not run on other platforms running older releases.	Control pedigree/version, ensure proper version is used. Make operational manager aware of versioning difference in operation.	SMS and other like services. Web-based versioning	Without Administrative control it may not be possible to force other participants to upgrade,
5	6	4	Increased Failure Possibilities	Extra dimensions of failure due to distributed environment (e.g., network failures, protocol faults, timeouts leading to unavailable services, server overload and race conditions)		Include test specification for network related failures.	Requirement planning for degraded service modes Detection of failures (exception handling) Testing for the degraded modes	Proliferation of partial configurations (degraded modes) that should all be tested individually Failure to specify all degraded modes in the requirements
5	8	4	Leasing of Services	Leasing of software services over a network reduces level of operational control and hence adds new modes of failure (due to retracted/ improved services)	Web/E-mail hosting and other middle tier services	Enter into contractual agreements with service provider with respect to version, availability and support.	Contracting vehicle to provide guarantee of services.	Loss of service Lack of control
3	6	4	COTS Testing	Limitations of	Use of MS Office	Require COTS software	Gray box testing:	Increased effort and need

CL	LC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
				specification based testing of COTS (e.g., poor statement coverage)	products where there is no access to the source code, but object interface specifications are published.	to include testing certification (e.g., statement/path coverage). Objects should provide a "instrumentation" interface for exposing internal state for testing purposes Require software developer to provide test assertions based upon structural analysis	component developer provides access to state through gray box.	for access to proprietary information may preclude cooperation from developers.
6	5	4	Program Size Increases ("Bloat"); Increase of Dead Code	Many maintenance changes made to meet deadlines actually can increase code size and the level of dead code. - Maintainers are reluctant to delete apparently unused routines for fear that they might be needed in future. Identifying dead code is important to a maintainer because maintenance is typically a very costly process - There is a reluctance to make changes due to uncertainty about possible side effects. This can result in the presence of	Component-based development exacerbates this problem by making it easy for the developer to add additional controls to forms while suppressing the behavior of other controls by making them inaccessible. All the code (now dead) behind the inaccessible controls remains.	Identifying and possibly removing code that has no effect on functionality can help reduce the costs associated with debugging, making changes, porting and other maintenance activities.	- Static Analysis with extended component interfaces - event-flow analysis tools - Slicing (impact analysis) may help in this area.	User events may be simulated by injection of pseudo user event messages. This is difficult to predict and therefore difficult to test for.

CL	IC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
				procedures having duplicate functionality				
9	8	1	Keeping track of all of the components in a system.	It is often difficult to locate the code responsible for some observed program behavior. A code snippet can be reside in any of several places, such as on a form, a control, or in a module. Code can also be located in any of the applications that comprise an integrated solution, e.g., in a Visual Basic for Applications macro in MS Excel or MS Word.		Observe sound configuration management practices and utilize automated tools as much as possible.	- Visual Development Environment - Versioning (CM) tools	Minimal
6	5	4	Name Shadowing	Name shadowing presents debugging difficulties. Although this problem is not specific to visual and/or object-oriented programming, the number of disparate places in which variable declarations may reside exacerbates it.	Scoping within programming environment	Use automated tools to aid in the tracking of variable names and scoping issues.	Static Analysis with extended component interfaces Use of Name Spaces	Cost of static analysis tool
2	8	4	Component Behavior on Client Machines	The behavior of the individual application being interfaced with is a function of its API and properties. When new versions of an integrated	Loading a spreadsheet in a particular manner depends on the settings of its properties (like virus	Apply techniques to check or control the expected interface with a distributed client component.	Wrappers may be applied.	Security constraints (e.g., wrappers may be prohibited)

CI	LC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
				application are released, the expectant behavior, API and properties may change.	checking which may disable macros from working).			
9	3	4	Infrastructure Management	It may transpire that a compiled version of the application will fail to run because some component is missing on the target computer, or because it has the wrong version of a run-time library. In general, managing this infrastructure is a major problem.	For an integrated application that invokes the facilities of a COTS application (e.g., uses OLE to employ Excel calculation facilities), upgrading the COTS package at the client can cause unforeseen problems	Use a built-in install function to ensure that all the necessary support facilities are in place and current.	Installation management software SMS or SMS-like tools	In general, need to retest whenever release of a new component version may change the client software's configuration
6	5	4	Component Interface Availability	Dynamic Link Libraries (DLLs), vendor-supplied controls and the like may not provide a window into their APIs. If not, we cannot apply slicing for impact analysis. Even if so, lack of access to the source code makes the program understanding problem more difficult.	COM components (e.g., Microsoft Excel) do not provide information needed for static analysis	Component Vendors should supply the necessary interface information.	Assertions about parameter sue and control dependence	No incentive by developer to provide information, possible proprietary information leaks
8	5	4	Event Visibility	Events can be made inaccessible by particular values of object properties. For example, changing the width property of a form may "hide" controls on that form. The code	Changing the width property of a form may "hide" controls on that form. The code associated with those controls is still compiled into the program but events	Extend static analysis techniques with the appropriate semantic information to handle such cases.	- Static Analysis with extended component interfaces - event-flow analysis tools	User events may be simulated by injection of pseudo user event messages. This is difficult to predict and therefore difficult to test for.

CL	LC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
				associated with those controls is still compiled into the program but events (e.g., mouse-click) for the control may be prohibited from running because no way exists for a user to interact with the control. Changing the visibility property of a control can have the same effect.	(e.g., mouse-click) for the control may be prohibited from running because no way exists for a user to interact with the control. Changing the visibility property of a control can have the same effect.			
2	8	4	Availability of Source Code	Software components may be built in-house or used off-the-shelf. The developer of a component has access to its source code. The user of an off-the-shelf component usually does not have access to the source code.	Using the Microsoft Excel component. There is no source code available.	Depending on the availability of code, different testing and analysis techniques need to be used.	Component provided summary information made available to the component users.	Cost of a static analysis tool to generate this summary information. Must convince component providers to provide this by making the task as easy and low cost as possible.
2	8	4	Heterogeneity of Language, Platforms and Architectures	The components of a system may be written in different programming languages and for use on different hardware and software platforms.	With middleware conforming to standards like CORBA and DCOM, components can interact with each other independent of the language and the platforms.	When a system composed of such components is tested or analyzed, the methodology and tool used must be independent of the language and the platforms.	Component provided summary information made available to the component users. This should be documented in a standard format.	The use of multiple languages to develop various components within a component-based system requires special static analysis tools unique for each language.
6	5	4	Deadlocks and Race Conditions	Distributed or concurrent systems occasionally have problems related to race conditions and	Component call backs. Consider the case where a client component calls a	It would be good for testing and analysis tools to catch this. Standards such as COM	Extended static analysis techniques using the component provided summary	Cost of a static analysis tool to generate this summary information. Must convince component

CL	LC	OM	ISSUE	IMPACT	EXAMPLE	MITIGATION	TOOLS	SOLUTION RISKS
				deadlocks. This is true for components.	server component and waits. The server component calls back to the client (say for status notification purposes). This could potentially lead to deadlock.	state that the developer should not do this, however there is no enforcement.	information could help to find such cases and flag them. The extended call graph will support this.	providers to provide this by making the task as easy and low cost as possible.
6	5	4	Complex Language Constructs	Other difficult problems include virtual functions, function pointers, and dynamic object binding.	Deriving a new window object at run time may override a virtual function that has been previously tested (runtime overloading).	Use dynamic and/or hybrid analysis techniques to monitor the execution of a program and collect the necessary information to take these constructs into account.	Possibly, wrappers could be used to identify the use of an untested override of a virtual function.	New objects can be defined that override virtual functions at any time. Thus, testing can never really be complete.

APPENDIX B. CASE STUDY REPORTS

This appendix provides a sample of each of the reports that were generated for the case studies used in this research. Specifically, the detailed reports for case study 7 are provided here. A total of 9 case studies were analyzed. Seven case studies represent real COTS component-based systems developed and maintained by the Department of Defense (DoD). In addition, two other case studies represent academic examples designed to illustrate some interesting aspects of component-based development. Due to the size of many of the reports, they are not included here. Copies of all reports can be obtained by contacting the author.

B.1. Case Study 7 – Book Sale Manager

B.1.1. Integrated system program summary report

```
-------------------------------------------------------------
Project: Book_cli.vbp  Project summary
Project created in Visual Basic 6.0
-------------------------------------------------------------
Files

Total files:                    23
Source files:                    9 (max approx. 400)

File types
Modules:                         2
Forms:                           4
Classes:                         3
Binary Property Files:           3
Referenced Files:                9
Project Workspaces:              1
Project Files:                   1

Oldest source file: 6/5/98 00:00 - frmBookSales
Newest source file: 8/5/00 19:33 - Sales
Average source file age: 15.1 months = 1.2 years
-------------------------------------------------------------
Code size

Lines of code:          606
Lines of comment:        43
Lines of whitespace:    272
Total source lines:     921

Total source bytes: 59 kB

Averages
Source lines per module: 97
Source bytes per module: 6.6 kB

Source lines per procedure: 20
Source bytes per procedure: 1.3 kB

Max and min
Longest source file:  285 lines (max 65534 lines) - Sales
Shortest source file: 2 lines - frmBookSales

Largest source file:  20.0 kB - frmRevenue
Smallest source file: 1.0 kB - ServerMain

Number of identifiers
```

```
Total number of identifiers: 254 (max 32000)
------------------------------------------------------------
Forms and controls

Forms: 4 (max 230)
Controls: 82
------------------------------------------------------------
Procedures

Basic procedures     44
DLL procedures       0

Public procedures    19
Private procedures   25

Subs                 29
Functions            15

Property procedures  0

Dead procedures      4
Live procedures      40

Total                44
------------------------------------------------------------
Variables and constants

                        Variables Constants Total
Global                  28        1         29
Module-level            16        0         16
Procedure-level         37        3         40
Procedure parameters    33                  33

Dead                    34        0         34
Live                    80        4         84
Total                   114       4         118
------------------------------------------------------------
Enums

Total Enums: 1
Private enums: 0
Public enums: 1
------------------------------------------------------------
User defined Types

Total Types: 0
------------------------------------------------------------
Variable types

Type              Count
ADODB Connection      2
ADODB.Recordset      11
Boolean               2
Currency             22
Field                 1
Integer              29
Long                  8
Model                 1
Panel                 1
Sales                 1
Single                5
String               29
Taxes                 1
Variant               1

Project Analyzer 5.0.07 (8/20/00) book_cli.vbp v6.2.8175
```

B.1.2. Integrated system OA-dead report

```
------------------------------------------------------------
Project: Book_cli.vbp  Semi-Dead Code Report
------------------------------------------------------------
List of semi-dead files:
```

```
Number of semi-dead files: 0
********************************************************************************

------------------------------------------------------------
List of semi-dead controls:

txtRevParm                                                      index: 5
   reason(s):
      Disabled
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
txtRevParm                                                      index: 4
   reason(s):
      Disabled
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
txtRevParm                                                      index: 6
   reason(s):
      Disabled
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
Label2                                                          index: 5
   reason(s):
      Disabled
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
Label2                                                          index: 4
   reason(s):
      Disabled
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
Label2                                                          index: 6
   reason(s):
      Disabled
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
dlgFindDB                                                       index: 0
   reason(s):
      Too narrow
      Too short
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
dlgFindDB                                                       index: 0
   reason(s):
      Too narrow
      Too short
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\Book_svr.frm
Number of semi-dead controls: 8
Number of controls semi-dead by reason:
     Disabled: 6 (75%)
     Invisible: 0 (0%)
     Too narrow: 2 (25%)
     Too short: 2 (25%)
     Too far right: 0 (0%)
     Too far down: 0 (0%)
     Too far left: 0 (0%)
     Too far up: 0 (0%)
     In semi-dead file: 0 (0%)
********************************************************************************

------------------------------------------------------------
List of semi-dead procedures:

txtRevParm_GotFocus
   reason(s):
      Event of semi-dead control
   in file:
```

```
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\BOOK_CLI.FRM
Class_Initialize
   reason(s):
      Called only by dead and/or semi-dead procedures
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\Sales.cls
LoadDB
   reason(s):
      Called only by dead and/or semi-dead procedures
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\Sales.cls
GetBooksale
   reason(s):
      Called only by dead and/or semi-dead procedures
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\Sales.cls
Class_Terminate
   reason(s):
      Called only by dead and/or semi-dead procedures
   in file:
      D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 3-
Integrated System\Sales.cls
Number of semi-dead Procedures: 5
     In semi-dead file: 0 (0%)
     Event of semi-dead control: 1 (20%)
     Called by semi-dead procedure: 4 (80%)
********************************************************************************

Project Analyzer 5.0.07 (8/20/00) book_cli.vbp v6.2 8175
```

B.1.3. Interface design language (IDL) reports

B.1.3.1 BookSale component IDL

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: Book_svr.exe
 [
  uuid(B9E01A72-6B04-11D4-8099-00A0CCE27EBB),
  version(1.0),
  helpstring("Author & Publisher Sales Revenue Sample Server"),
  custom(50867B00-BB69-11D0-A8FF-00A0C9110059, 8495)
]
library BookSaleSvr
{
    // TLib :     // TLib : Microsoft ActiveX Data Objects 2.0 Library : {00000200-0000-
0010-8000-00AA006D2EA4}
    importlib("msado20.tlb");
    // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
    importlib("stdole2.tlb");
    // Forward declare all types defined in this typelib
    interface _Sales;
    [
      odl,
      uuid(B9E01A74-6B04-11D4-8099-00A0CCE27EBB),
      version(1.0),
      hidden,
      dual,
      nonextensible,
      oleautomation
    ]
    interface _Sales : IDispatch {
        [id(0x60030000)]
        HRESULT GetAuthors([out, retval] _Recordset** );
        [id(0x60030001)]
        HRESULT GetTitles(
                        [in] BSTR strSQL,
                        [out, retval] _Recordset** );
```

```
        [id(0x60030002)]
        HRESULT GetBookPages(
                        [in] BSTR strSQL,
                        [out, retval] _Recordset** );
        [id(0x60030003)]
        HRESULT GetRsCOGS(
                        [in] BSTR strSQL,
                        [out, retval] _Recordset** );
        [id(0x60030004)]
        HRESULT GetRevenue(
                        [in, out] short* intSalesModel,
                        [in, out] CURRENCY* curCostPerUnit,
                        [in, out] CURRENCY* curAdvCost,
                        [in, out] short* intSalesPeriod,
                        [in, out] long* lngUnitsPerMonth,
                        [in, out] VARIANT_BOOL* bolIsDiscount,
                        [in, out] BSTR* strBookTitle,
                        [out, retval] VARIANT* );
        [id(0x60030005)]
        HRESULT GetAuthorRoyalty([out, retval] VARIANT_BOOL* );
        [id(0x60030006)]
        HRESULT GetPubRevenue(
                        [in, out] BSTR* strTitle,
                        [out, retval] VARIANT* );
    };
    [
      uuid(B9E01A75-6B04-11D4-8099-00A0CCE27EBB),
      version(1.0)
    ]
    coclass Sales {
        [default] interface _Sales;
    };
};
```

B.1.3.2 BookSale type library documenter report

```
BookSaleSvr Interface Definition

General Information
Library:        BookSaleSvr (Author & Publisher Sales Revenue Sample Server)
File:   D:\work\PhD\Test and Validation\PhD_Validation\Technique2\sys7-booksale\Step 1-
Initial\Server\Book_svr.exe
GUID:   {B9E01A72-6B04-11D4-8099-00A0CCE27EBB}
Version:        1.0

Enumerations
This section lists enumerations exposed by BookSaleSvr.


Interfaces
This section lists the Classes exposed by BookSaleSvr.  For each class, the methods and
events are listed.

Sales {B9E01A75-6B04-11D4-8099-00A0CCE27EBB}

Methods
Function GetAuthors() As _Recordset

Function GetTitles(ByVal strSQL As String) As _Recordset

Function GetBookPages(ByVal strSQL As String) As _Recordset

Function GetRsCOGS(ByVal strSQL As String) As _Recordset

Function GetRevenue(ByVal intSalesModel As Integer, ByVal curCostPerUnit As Currency,
ByVal curAdvCost As Currency, ByVal intSalesPeriod As Integer, ByVal lngUnitsPerMonth As
Long, ByVal bolIsDiscount As Boolean, ByVal strBookTitle As String) As Variant

Function GetAuthorRoyalty() As Boolean

Function GetPubRevenue(ByVal strTitle As String) As Variant

Events
None
```

B.1.4. Integrated system XML summary call graph

```
<?xml version="1.0"?>
<!--DOCTYPE CallGraph SYSTEM "file://CallGraph.dtd"-->
<!-- META NAME="Generator" CONTENT="Project Analyzer 5.0.07" -->
<!-- meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" -->
<!-- Project Analyzer Report / sys7_cg_s3.xml -->
<!-- Project: Book_cli.vbp -->
<!-- Call tree -->
<CallGraph>
<ModuleCollection>ClientMain</ModuleCollection>
<CallGraphNode NodeID="2" ModuleName="lGetAuthors" ParentModName="ClientMain">
   <CalledModules>
      <Module ModuleName="GetAuthors" ModuleID="39"> </Module>
   </CalledModules>
   <CallSites>
      <CallSite>
         <Module ModuleName="GetAuthors" ModuleID="39" InModuleCollection="Sales"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>


         </ParameterMapping>
      </CallSite>
   </CallSites>
   <GlobalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="gCN" VarID="60"> </Variable>
      <Variable VarName="rsAuthors" VarID="54"> </Variable>
      <Variable VarName="rsAuthors" VarID="9"> </Variable>
      <Variable VarName="gobjServer" VarID="1"> </Variable>
   </VarRefs>
   </GlobalRefs>
   <GlobalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="rsAuthors" VarID="54"> </Variable>
      <Variable VarName="rsAuthors" VarID="9"> </Variable>
   </VarDefs>
   </GlobalDefs>
   <LocalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
   </VarRefs>
   </LocalRefs>
   <LocalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
   </VarDefs>
   </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="3" ModuleName="lGetTitles" ParentModName="ClientMain">
   <FormalParameters>
      <PassByRef>
         <Parameter VarName="strAuthor" VarID="10" FirstUse="REF" LastUse="REF">
</Parameter>
      </PassByRef>
      <PassByVal>
      </PassByVal>
   </FormalParameters>
   <ConstantDeclarations>
   </ConstantDeclarations>
   <VariableDeclarations>
      <Variable VarName="strSQL" VarID="11"> </Variable>
      <Variable VarName="rsTitles" VarID="12"> </Variable>
   </VariableDeclarations>
```

```
    <CalledModules>
       <Module ModuleName="GetTitles" ModuleID="40"> </Module>
    </CalledModules>
    <CallSites>
       <CallSite>
          <Module ModuleName="GetTitles" ModuleID="40" InModuleCollection="Sales"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>
             <ActualParameter VarName="strSQL" VarID="11"> </ActualParameter>
             <PassByVal>
                <Parameter VarName="strSQL" VarID="88" FirstUse="REF" LastUse="REF">
</Parameter>
             </PassByVal>


          </ParameterMapping>
       </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="gCN" VarID="60"> </Variable>
       <Variable VarName="rsTitles" VarID="55"> </Variable>
       <Variable VarName="strAuthor" VarID="10"> </Variable>
       <Variable VarName="gobjServer" VarID="1"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="rsTitles" VarID="55"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="rsTitles" VarID="12"> </Variable>
       <Variable VarName="strSQL" VarID="11"> </Variable>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="rsTitles" VarID="12"> </Variable>
       <Variable VarName="strSQL" VarID="11"> </Variable>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>

<ModuleCollection>frmChart</ModuleCollection>
<CallGraphNode NodeID="5" ModuleName="cmdClose_Click" ParentModName="frmChart">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
```

```
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="6" ModuleName="Form_Load" ParentModName="frmChart">
    <CalledModules>
       <Module ModuleName="SetGraphData" ModuleID="7"> </Module>
    </CalledModules>
    <CallSites>
       <CallSite>
          <Module ModuleName="SetGraphData" ModuleID="7" InModuleCollection="frmChart"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>


          </ParameterMapping>
       </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
       <Constant ConstName="gRoyalty" ConstID="58"> </Constant>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="gobjServer" VarID="0"> </Variable>
       <Variable VarName="gintSalesModel" VarID="62"> </Variable>
       <Variable VarName="sngBookPrice" VarID="86"> </Variable>
       <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
       <Variable VarName="gCN" VarID="60"> </Variable>
       <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
       <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
       <Variable VarName="goStatusPanel" VarID="2"> </Variable>
       <Variable VarName="gobjServer" VarID="1"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
       <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
       <Variable VarName="glngUnitsPerMonth" VarID="63"> </Variable>
       <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
       <Variable VarName="gcurAdvertisingCost" VarID="73"> </Variable>
       <Variable VarName="gcurCostPerUnit" VarID="64"> </Variable>
       <Variable VarName="gintSalesModel" VarID="62"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="7" ModuleName="SetGraphData" ParentModName="frmChart">
    <FormalParameters>
       <PassByRef>
       </PassByRef>
       <PassByVal>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
```

```
      <Variable VarName="strGraphData" VarID="13"> </Variable>
      <Variable VarName="vGraphData" VarID="14"> </Variable>
      <Variable VarName="strSrchString" VarID="15"> </Variable>
      <Variable VarName="lStart" VarID="16"> </Variable>
      <Variable VarName="lEnd" VarID="17"> </Variable>
      <Variable VarName="lstrLen" VarID="18"> </Variable>
      <Variable VarName="i" VarID="19"> </Variable>
      <Variable VarName="j" VarID="20"> </Variable>
   </VariableDeclarations>
   <CalledModules>
      <Module ModuleName="GetRevenue" ModuleID="43"> </Module>
   </CalledModules>
   <CallSites>
      <CallSite>
         <Module ModuleName="GetRevenue" ModuleID="43" InModuleCollection="Sales"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>


         </ParameterMapping>
      </CallSite>
   </CallSites>
   <GlobalRefs>
   <ConstRefs>
      <Constant ConstName="gRoyalty" ConstID="58"> </Constant>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="gobjServer" VarID="0"> </Variable>
      <Variable VarName="gintSalesModel" VarID="62"> </Variable>
      <Variable VarName="sngBookPrice" VarID="86"> </Variable>
      <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
      <Variable VarName="gCN" VarID="60"> </Variable>
      <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
      <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
      <Variable VarName="goStatusPanel" VarID="2"> </Variable>
      <Variable VarName="gobjServer" VarID="1"> </Variable>
   </VarRefs>
   </GlobalRefs>
   <GlobalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
      <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
      <Variable VarName="glngUnitsPerMonth" VarID="63"> </Variable>
      <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
      <Variable VarName="gcurAdvertisingCost" VarID="73"> </Variable>
      <Variable VarName="gcurCostPerUnit" VarID="64"> </Variable>
      <Variable VarName="gintSalesModel" VarID="62"> </Variable>
   </VarDefs>
   </GlobalDefs>
   <LocalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="j" VarID="20"> </Variable>
      <Variable VarName="i" VarID="19"> </Variable>
      <Variable VarName="vGraphData" VarID="14"> </Variable>
   </VarRefs>
   </LocalRefs>
   <LocalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="i" VarID="19"> </Variable>
      <Variable VarName="j" VarID="20"> </Variable>
      <Variable VarName="vGraphData" VarID="14"> </Variable>
   </VarDefs>
   </LocalDefs>
</CallGraphNode>

<ModuleCollection>frmRevenue</ModuleCollection>
<CallGraphNode NodeID="9" ModuleName="cboAuthors_Click" ParentModName="frmRevenue">
   <CalledModules>
      <Module ModuleName="lGetTitles" ModuleID="3"> </Module>
```

```
    </CalledModules>
    <CallSites>
      <CallSite>
        <Module ModuleName="lGetTitles" ModuleID="3" InModuleCollection="ClientMain"/>
        <StatementLineNumber/>
        <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
        <ParameterMapping>


        </ParameterMapping>
      </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
      <Variable VarName="gCN" VarID="60"> </Variable>
      <Variable VarName="rsTitles" VarID="55"> </Variable>
      <Variable VarName="gobjServer" VarID="1"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
      <Variable VarName="rsTitles" VarID="55"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
  </CallGraphNode>
  <CallGraphNode NodeID="10" ModuleName="chkDiscount_Click" ParentModName="frmRevenue">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
  </CallGraphNode>
  <CallGraphNode NodeID="11" ModuleName="cmdClose_Click" ParentModName="frmRevenue">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
```

```
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="12" ModuleName="cmdCogs_Click" ParentModName="frmRevenue">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="goStatusPanel" VarID="2"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="13" ModuleName="cmdHelp_Click" ParentModName="frmRevenue">
    <FormalParameters>
       <PassByRef>
       </PassByRef>
       <PassByVal>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
       <Variable VarName="sHelpString" VarID="22"> </Variable>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
```

```
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="sHelpString" VarID="22"> </Variable>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="sHelpString" VarID="22"> </Variable>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="14" ModuleName="cmdExecute_Click" ParentModName="frmRevenue">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="goStatusPanel" VarID="2"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="15" ModuleName="GetBooksale" ParentModName="frmRevenue">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="gDBName" VarID="5"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
```

```
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="16" ModuleName="LoadDB" ParentModName="frmRevenue">
    <CalledModules>
        <Module ModuleName="GetBooksale" ModuleID="15"> </Module>
    </CalledModules>
    <CallSites>
        <CallSite>
            <Module ModuleName="GetBooksale" ModuleID="15" InModuleCollection="frmRevenue"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="gDBName" VarID="5"> </Variable>
        <Variable VarName="gCN" VarID="60"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
        <Variable VarName="gCN" VarID="60"> </Variable>
        <Variable VarName="gDBName" VarID="5"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="17" ModuleName="Form_Load" ParentModName="frmRevenue">
    <CalledModules>
        <Module ModuleName="lGetAuthors" ModuleID="2"> </Module>
    </CalledModules>
    <CallSites>
        <CallSite>
            <Module ModuleName="lGetAuthors" ModuleID="2" InModuleCollection="ClientMain"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="gCN" VarID="60"> </Variable>
        <Variable VarName="rsAuthors" VarID="54"> </Variable>
        <Variable VarName="rsAuthors" VarID="9"> </Variable>
        <Variable VarName="gobjServer" VarID="1"> </Variable>
```

```
            <Variable VarName="goStatusPanel" VarID="2"> </Variable>
        </VarRefs>
        </GlobalRefs>
        <GlobalDefs>
        <ConstDefs>
        </ConstDefs>
        <VarDefs>
            <Variable VarName="rsAuthors" VarID="54"> </Variable>
            <Variable VarName="rsAuthors" VarID="9"> </Variable>
            <Variable VarName="gobjServer" VarID="1"> </Variable>
            <Variable VarName="goStatusPanel" VarID="2"> </Variable>
        </VarDefs>
        </GlobalDefs>
        <LocalRefs>
        <ConstRefs>
        </ConstRefs>
        <VarRefs>
        </VarRefs>
        </LocalRefs>
        <LocalDefs>
        <ConstDefs>
        </ConstDefs>
        <VarDefs>
        </VarDefs>
        </LocalDefs>
    </CallGraphNode>
    <CallGraphNode NodeID="18" ModuleName="Form_Unload" ParentModName="frmRevenue">
        <FormalParameters>
            <PassByRef>
                <Parameter VarName="Cancel" VarID="23" FirstUse="" LastUse=""> </Parameter>
            </PassByRef>
            <PassByVal>
            </PassByVal>
        </FormalParameters>
        <ConstantDeclarations>
        </ConstantDeclarations>
        <VariableDeclarations>
        </VariableDeclarations>
        <CalledModules>
        </CalledModules>
        <CallSites>
        </CallSites>
        <GlobalRefs>
        <ConstRefs>
        </ConstRefs>
        <VarRefs>
        </VarRefs>
        </GlobalRefs>
        <GlobalDefs>
        <ConstDefs>
        </ConstDefs>
        <VarDefs>
            <Variable VarName="gobjServer" VarID="1"> </Variable>
            <Variable VarName="gSn" VarID="6"> </Variable>
        </VarDefs>
        </GlobalDefs>
        <LocalRefs>
        <ConstRefs>
        </ConstRefs>
        <VarRefs>
        </VarRefs>
        </LocalRefs>
        <LocalDefs>
        <ConstDefs>
        </ConstDefs>
        <VarDefs>
        </VarDefs>
        </LocalDefs>
    </CallGraphNode>
    <CallGraphNode NodeID="19" ModuleName="optAnalysis_Click" ParentModName="frmRevenue">
        <FormalParameters>
            <PassByRef>
                <Parameter VarName="Index" VarID="24" FirstUse="REF" LastUse="REF"> </Parameter>
            </PassByRef>
            <PassByVal>
            </PassByVal>
```

```
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="Index" VarID="24"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="20" ModuleName="txtRevParm_GotFocus" ParentModName="frmRevenue">
    <FormalParameters>
       <PassByRef>
          <Parameter VarName="Index" VarID="25" FirstUse="REF" LastUse="REF"> </Parameter>
       </PassByRef>
       <PassByVal>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="Index" VarID="25"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
```

```
</CallGraphNode>
<CallGraphNode NodeID="21" ModuleName="udDiscount_DownClick" ParentModName="frmRevenue">
   <FormalParameters>
      <PassByRef>
         <Parameter VarName="Index" VarID="26" FirstUse="REF" LastUse="REF"> </Parameter>
      </PassByRef>
      <PassByVal>
      </PassByVal>
   </FormalParameters>
   <ConstantDeclarations>
   </ConstantDeclarations>
   <VariableDeclarations>
   </VariableDeclarations>
   <CalledModules>
   </CalledModules>
   <CallSites>
   </CallSites>
   <GlobalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="Index" VarID="26"> </Variable>
   </VarRefs>
   </GlobalRefs>
   <GlobalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
   </VarDefs>
   </GlobalDefs>
   <LocalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
   </VarRefs>
   </LocalRefs>
   <LocalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
   </VarDefs>
   </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="22" ModuleName="udDiscount_UpClick" ParentModName="frmRevenue">
   <FormalParameters>
      <PassByRef>
         <Parameter VarName="Index" VarID="27" FirstUse="REF" LastUse="REF"> </Parameter>
      </PassByRef>
      <PassByVal>
      </PassByVal>
   </FormalParameters>
   <ConstantDeclarations>
   </ConstantDeclarations>
   <VariableDeclarations>
   </VariableDeclarations>
   <CalledModules>
   </CalledModules>
   <CallSites>
   </CallSites>
   <GlobalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="Index" VarID="27"> </Variable>
   </VarRefs>
   </GlobalRefs>
   <GlobalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
   </VarDefs>
   </GlobalDefs>
   <LocalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
```

```
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>

<ModuleCollection>frmCogs</ModuleCollection>
<CallGraphNode NodeID="24" ModuleName="optPicColor_Click" ParentModName="frmCogs">
    <FormalParameters>
        <PassByRef>
            <Parameter VarName="Index" VarID="40" FirstUse="REF" LastUse="REF"> </Parameter>
        </PassByRef>
        <PassByVal>
        </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
        <Module ModuleName="CalcUnitCost" ModuleID="26"> </Module>
    </CalledModules>
    <CallSites>
        <CallSite>
            <Module ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="lngNumPages" VarID="32"> </Variable>
        <Variable VarName="Index" VarID="40"> </Variable>
        <Variable VarName="acurCogs" VarID="39"> </Variable>
        <Variable VarName="acurPictureCost" VarID="37"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
        <Variable VarName="acurCogs" VarID="39"> </Variable>
        <Variable VarName="strPicture" VarID="34"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="25" ModuleName="Command1_Click" ParentModName="frmCogs">
    <FormalParameters>
        <PassByRef>
            <Parameter VarName="Index" VarID="41" FirstUse="REF" LastUse="REF"> </Parameter>
        </PassByRef>
        <PassByVal>
        </PassByVal>
    </FormalParameters>
```

```
      <ConstantDeclarations>
      </ConstantDeclarations>
      <VariableDeclarations>
         <Variable VarName="sHelpString" VarID="42"> </Variable>
      </VariableDeclarations>
      <CalledModules>
      </CalledModules>
      <CallSites>
      </CallSites>
      <GlobalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
         <Variable VarName="Index" VarID="41"> </Variable>
         <Variable VarName="strPaper" VarID="35"> </Variable>
         <Variable VarName="strPicture" VarID="34"> </Variable>
         <Variable VarName="strBinding" VarID="33"> </Variable>
      </VarRefs>
      </GlobalRefs>
      <GlobalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
         <Variable VarName="lRetVal" VarID="28"> </Variable>
      </VarDefs>
      </GlobalDefs>
      <LocalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
         <Variable VarName="sHelpString" VarID="42"> </Variable>
      </VarRefs>
      </LocalRefs>
      <LocalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
         <Variable VarName="sHelpString" VarID="42"> </Variable>
      </VarDefs>
      </LocalDefs>
   </CallGraphNode>
   <CallGraphNode NodeID="26" ModuleName="CalcUnitCost" ParentModName="frmCogs">
      <FormalParameters>
         <PassByRef>
         </PassByRef>
         <PassByVal>
         </PassByVal>
      </FormalParameters>
      <ConstantDeclarations>
      </ConstantDeclarations>
      <VariableDeclarations>
         <Variable VarName="curTemp" VarID="43"> </Variable>
      </VariableDeclarations>
      <CalledModules>
      </CalledModules>
      <CallSites>
      </CallSites>
      <GlobalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
         <Variable VarName="lngNumPages" VarID="32"> </Variable>
         <Variable VarName="acurCogs" VarID="39"> </Variable>
      </VarRefs>
      </GlobalRefs>
      <GlobalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
      </VarDefs>
      </GlobalDefs>
      <LocalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
         <Variable VarName="curTemp" VarID="43"> </Variable>
```

```
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="curTemp" VarID="43"> </Variable>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="27" ModuleName="GetCOGS" ParentModName="frmCogs">
    <FormalParameters>
       <PassByRef>
       </PassByRef>
       <PassByVal>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
       <Variable VarName="fld" VarID="44"> </Variable>
       <Variable VarName="strSQL" VarID="45"> </Variable>
       <Variable VarName="rsCOGS" VarID="46"> </Variable>
    </VariableDeclarations>
    <CalledModules>
       <Module ModuleName="GetRsCOGS" ModuleID="42"> </Module>
    </CalledModules>
    <CallSites>
       <CallSite>
          <Module ModuleName="GetRsCOGS" ModuleID="42" InModuleCollection="Sales"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>
             <ActualParameter VarName="strSQL" VarID="45"> </ActualParameter>
             <PassByVal>
                <Parameter VarName="strSQL" VarID="90" FirstUse="REF" LastUse="REF">
</Parameter>
                </PassByVal>


          </ParameterMapping>
       </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="gCN" VarID="60"> </Variable>
       <Variable VarName="rsCOGS" VarID="57"> </Variable>
       <Variable VarName="gSn" VarID="6"> </Variable>
       <Variable VarName="gobjServer" VarID="1"> </Variable>
       <Variable VarName="goStatusPanel" VarID="2"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="rsCOGS" VarID="57"> </Variable>
       <Variable VarName="acurPaperCost" VarID="38"> </Variable>
       <Variable VarName="acurPictureCost" VarID="37"> </Variable>
       <Variable VarName="acurBindingCost" VarID="36"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="rsCOGS" VarID="46"> </Variable>
       <Variable VarName="strSQL" VarID="45"> </Variable>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
```

```
      <Variable VarName="rsCOGS" VarID="46"> </Variable>
      <Variable VarName="strSQL" VarID="45"> </Variable>
   </VarDefs>
   </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="28" ModuleName="lGetBookPages" ParentModName="frmCogs">
   <FormalParameters>
      <PassByRef>
         <Parameter VarName="strTitle" VarID="47" FirstUse="REF" LastUse="REF">
</Parameter>
      </PassByRef>
      <PassByVal>
      </PassByVal>
   </FormalParameters>
   <ConstantDeclarations>
   </ConstantDeclarations>
   <VariableDeclarations>
      <Variable VarName="strSQL" VarID="48"> </Variable>
      <Variable VarName="rsBookPages" VarID="49"> </Variable>
      <Variable VarName="strOldTitle" VarID="50"> </Variable>
      <Variable VarName="lngPages" VarID="51"> </Variable>
   </VariableDeclarations>
   <CalledModules>
      <Module ModuleName="GetBookPages" ModuleID="41"> </Module>
   </CalledModules>
   <CallSites>
      <CallSite>
         <Module ModuleName="GetBookPages" ModuleID="41" InModuleCollection="Sales"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>
            <ActualParameter VarName="strSQL" VarID="48"> </ActualParameter>
            <PassByVal>
               <Parameter VarName="strSQL" VarID="89" FirstUse="REF" LastUse="REF">
</Parameter>
            </PassByVal>


         </ParameterMapping>
      </CallSite>
   </CallSites>
   <GlobalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="gCN" VarID="60"> </Variable>
      <Variable VarName="rsBookPages" VarID="56"> </Variable>
      <Variable VarName="strTitle" VarID="47"> </Variable>
      <Variable VarName="goStatusPanel" VarID="2"> </Variable>
      <Variable VarName="gobjServer" VarID="1"> </Variable>
   </VarRefs>
   </GlobalRefs>
   <GlobalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="rsBookPages" VarID="56"> </Variable>
   </VarDefs>
   </GlobalDefs>
   <LocalRefs>
   <ConstRefs>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="lngPages" VarID="51"> </Variable>
      <Variable VarName="rsBookPages" VarID="49"> </Variable>
      <Variable VarName="strSQL" VarID="48"> </Variable>
      <Variable VarName="strOldTitle" VarID="50"> </Variable>
   </VarRefs>
   </LocalRefs>
   <LocalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="strOldTitle" VarID="50"> </Variable>
      <Variable VarName="lngPages" VarID="51"> </Variable>
      <Variable VarName="rsBookPages" VarID="49"> </Variable>
```

```
        <Variable VarName="strSQL" VarID="48"> </Variable>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="29" ModuleName="Form_Load" ParentModName="frmCogs">
    <CalledModules>
        <Module ModuleName="lGetBookPages" ModuleID="28"> </Module>
        <Module ModuleName="GetCOGS" ModuleID="27"> </Module>
        <Module ModuleName="CalcUnitCost" ModuleID="26"> </Module>
    </CalledModules>
    <CallSites>
        <CallSite>
            <Module ModuleName="lGetBookPages" ModuleID="28" InModuleCollection="frmCogs"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
        <CallSite>
            <Module ModuleName="GetCOGS" ModuleID="27" InModuleCollection="frmCogs"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
        <CallSite>
            <Module ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="acurCogs" VarID="39"> </Variable>
        <Variable VarName="rsCOGS" VarID="57"> </Variable>
        <Variable VarName="gSn" VarID="6"> </Variable>
        <Variable VarName="gCN" VarID="60"> </Variable>
        <Variable VarName="rsBookPages" VarID="56"> </Variable>
        <Variable VarName="goStatusPanel" VarID="2"> </Variable>
        <Variable VarName="gobjServer" VarID="1"> </Variable>
        <Variable VarName="acurBindingCost" VarID="36"> </Variable>
        <Variable VarName="lngNumPages" VarID="32"> </Variable>
        <Variable VarName="acurPaperCost" VarID="38"> </Variable>
        <Variable VarName="acurPictureCost" VarID="37"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
        <Variable VarName="rsCOGS" VarID="57"> </Variable>
        <Variable VarName="acurPaperCost" VarID="38"> </Variable>
        <Variable VarName="acurPictureCost" VarID="37"> </Variable>
        <Variable VarName="acurBindingCost" VarID="36"> </Variable>
        <Variable VarName="rsBookPages" VarID="56"> </Variable>
        <Variable VarName="strPaper" VarID="35"> </Variable>
        <Variable VarName="strPicture" VarID="34"> </Variable>
        <Variable VarName="strBinding" VarID="33"> </Variable>
        <Variable VarName="acurCogs" VarID="39"> </Variable>
        <Variable VarName="lRetVal" VarID="28"> </Variable>
        <Variable VarName="lngNumPages" VarID="32"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
```

```
<VarRefs>
<VarRefs>
</LocalRefs>
<LocalDefs>
<ConstDefs>
</ConstDefs>
<VarDefs>
</VarDefs>
</LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="30" ModuleName="optBinding_Click" ParentModName="frmCogs">
<FormalParameters>
<PassByRef>
<Parameter VarName="Index" VarID="52" FirstUse="REF" LastUse="REF"> </Parameter>
</PassByRef>
<PassByVal>
</PassByVal>
</FormalParameters>
<ConstantDeclarations>
</ConstantDeclarations>
<VariableDeclarations>
</VariableDeclarations>
<CalledModules>
<Module ModuleName="optPaperGrade_Click" ModuleID="31"> </Module>
<Module ModuleName="CalcUnitCost" ModuleID="26"> </Module>
</CalledModules>
<CallSites>
<CallSite>
<Module ModuleName="optPaperGrade_Click" ModuleID="31"
InModuleCollection="frmCogs"/>
<StatementLineNumber/>
<CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
<ParameterMapping>


</ParameterMapping>
</CallSite>
<CallSite>
<Module ModuleName="optPaperGrade_Click" ModuleID="31"
InModuleCollection="frmCogs"/>
<StatementLineNumber/>
<CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
<ParameterMapping>


</ParameterMapping>
</CallSite>
<CallSite>
<Module ModuleName="optPaperGrade_Click" ModuleID="31"
InModuleCollection="frmCogs"/>
<StatementLineNumber/>
<CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
<ParameterMapping>


</ParameterMapping>
</CallSite>
<CallSite>
<Module ModuleName="optPaperGrade_Click" ModuleID="31"
InModuleCollection="frmCogs"/>
<StatementLineNumber/>
<CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
<ParameterMapping>


</ParameterMapping>
</CallSite>
<CallSite>
<Module ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"/>
<StatementLineNumber/>
<CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
<ParameterMapping>


</ParameterMapping>
</CallSite>
```

```
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="lngNumPages" VarID="32"> </Variable>
       <Variable VarName="acurPaperCost" VarID="38"> </Variable>
       <Variable VarName="Index" VarID="52"> </Variable>
       <Variable VarName="acurCogs" VarID="39"> </Variable>
       <Variable VarName="acurBindingCost" VarID="36"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="strPaper" VarID="35"> </Variable>
       <Variable VarName="acurCogs" VarID="39"> </Variable>
       <Variable VarName="strBinding" VarID="33"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="31" ModuleName="optPaperGrade_Click" ParentModName="frmCogs">
    <FormalParameters>
       <PassByRef>
          <Parameter VarName="Index" VarID="53" FirstUse="REF" LastUse="REF"> </Parameter>
       </PassByRef>
       <PassByVal>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
       <Module ModuleName="CalcUnitCost" ModuleID="26"> </Module>
    </CalledModules>
    <CallSites>
       <CallSite>
          <Module ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>


          </ParameterMapping>
       </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="lngNumPages" VarID="32"> </Variable>
       <Variable VarName="Index" VarID="53"> </Variable>
       <Variable VarName="acurCogs" VarID="39"> </Variable>
       <Variable VarName="acurPaperCost" VarID="38"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="acurCogs" VarID="39"> </Variable>
       <Variable VarName="strPaper" VarID="35"> </Variable>
```

```
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>

<ModuleCollection>frmBookSales</ModuleCollection>
<ModuleCollection>ServerMain</ModuleCollection>
<CallGraphNode NodeID="34" ModuleName="Main" ParentModName="ServerMain">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>

    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="35" ModuleName="ServerMsg" ParentModName="ServerMain">
    <FormalParameters>
        <PassByRef>
            <Parameter VarName="rstrMessage" VarID="74" FirstUse="" LastUse=""> </Parameter>
            <Parameter VarName="rintButtons" VarID="75" FirstUse="" LastUse=""> </Parameter>
            <Parameter VarName="rstrTitle" VarID="76" FirstUse="" LastUse=""> </Parameter>
        </PassByRef>
        <PassByVal>
        </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
```

```
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>

<ModuleCollection>Model</ModuleCollection>
<CallGraphNode NodeID="37" ModuleName="intGetMonthSales" ParentModName="Model">
    <FormalParameters>
        <PassByRef>
            <Parameter VarName="intCurMonth" VarID="77" FirstUse="REF" LastUse="REF">
</Parameter>
            <Parameter VarName="intSalesPeriod" VarID="78" FirstUse="REF" LastUse="REF">
</Parameter>
            <Parameter VarName="intModelType" VarID="79" FirstUse="REF" LastUse="REF">
</Parameter>
        </PassByRef>
        <PassByVal>
        </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
        <Constant ConstName="intMAX_SCHOOL_BOOK
" ConstID="81"> </Constant>
        <Constant ConstName="intMAX_POP_NOVEL
" ConstID="82"> </Constant>
        <Constant ConstName="intMAX_CELEBRITY
" ConstID="83"> </Constant>
    </ConstantDeclarations>
    <VariableDeclarations>
        <Variable VarName="intMonthSales" VarID="80"> </Variable>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="intSalesPeriod" VarID="78"> </Variable>
        <Variable VarName="intCurMonth" VarID="77"> </Variable>
        <Variable VarName="intModelType" VarID="79"> </Variable>
        <Variable VarName="gobjServer" VarID="0"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
        <Constant ConstName="intMAX_CELEBRITY" ConstID="83"> </Constant>
        <Constant ConstName="intMAX_POP_NOVEL" ConstID="82"> </Constant>
        <Constant ConstName="intMAX_SCHOOL_BOOK" ConstID="81"> </Constant>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="intMonthSales" VarID="80"> </Variable>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
```

```
            <Variable VarName="intMonthSales" VarID="80"> </Variable>
        </VarDefs>
        </LocalDefs>
    </CallGraphNode>

    <ModuleCollection>Sales</ModuleCollection>
    <CallGraphNode NodeID="39" ModuleName="GetAuthors" ParentModName="Sales">
        <FormalParameters>
            <PassByRef>
            </PassByRef>
            <PassByVal>
            </PassByVal>
        </FormalParameters>
        <ConstantDeclarations>
        </ConstantDeclarations>
        <VariableDeclarations>
            <Variable VarName="strSQL" VarID="87"> </Variable>
        </VariableDeclarations>
        <CalledModules>
        </CalledModules>
        <CallSites>
        </CallSites>
        <GlobalRefs>
        <ConstRefs>
        </ConstRefs>
        <VarRefs>
            <Variable VarName="gCN" VarID="60"> </Variable>
            <Variable VarName="rsAuthors" VarID="54"> </Variable>
        </VarRefs>
        </GlobalRefs>
        <GlobalDefs>
        <ConstDefs>
        </ConstDefs>
        <VarDefs>
            <Variable VarName="rsAuthors" VarID="54"> </Variable>
        </VarDefs>
        </GlobalDefs>
        <LocalRefs>
        <ConstRefs>
        </ConstRefs>
        <VarRefs>
            <Variable VarName="strSQL" VarID="87"> </Variable>
        </VarRefs>
        </LocalRefs>
        <LocalDefs>
        <ConstDefs>
        </ConstDefs>
        <VarDefs>
            <Variable VarName="strSQL" VarID="87"> </Variable>
        </VarDefs>
        </LocalDefs>
    </CallGraphNode>
    <CallGraphNode NodeID="40" ModuleName="GetTitles" ParentModName="Sales">
        <FormalParameters>
            <PassByRef>
            </PassByRef>
            <PassByVal>
                <Variable VarName="strSQL" VarID="88" FirstUse="REF" LastUse="REF"> </Variable>
            </PassByVal>
        </FormalParameters>
        <ConstantDeclarations>
        </ConstantDeclarations>
        <VariableDeclarations>
        </VariableDeclarations>
        <CalledModules>
        </CalledModules>
        <CallSites>
        </CallSites>
        <GlobalRefs>
        <ConstRefs>
        </ConstRefs>
        <VarRefs>
            <Variable VarName="gCN" VarID="60"> </Variable>
            <Variable VarName="rsTitles" VarID="55"> </Variable>
        </VarRefs>
        </GlobalRefs>
```

```
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="rsTitles" VarID="55"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="strSQL" VarID="88"> </Variable>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="41" ModuleName="GetBookPages" ParentModName="Sales">
    <FormalParameters>
       <PassByRef>
       </PassByRef>
       <PassByVal>
          <Variable VarName="strSQL" VarID="89" FirstUse="REF" LastUse="REF"> </Variable>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="gCN" VarID="60"> </Variable>
       <Variable VarName="rsBookPages" VarID="56"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="rsBookPages" VarID="56"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="strSQL" VarID="89"> </Variable>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="42" ModuleName="GetRsCOGS" ParentModName="Sales">
    <FormalParameters>
       <PassByRef>
       </PassByRef>
       <PassByVal>
          <Variable VarName="strSQL" VarID="90" FirstUse="REF" LastUse="REF"> </Variable>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
```

```
      <VariableDeclarations>
      </VariableDeclarations>
      <CalledModules>
      </CalledModules>
      <CallSites>
      </CallSites>
      <GlobalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
         <Variable VarName="gCN" VarID="60"> </Variable>
         <Variable VarName="rsCOGS" VarID="57"> </Variable>
      </VarRefs>
      </GlobalRefs>
      <GlobalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
         <Variable VarName="rsCOGS" VarID="57"> </Variable>
      </VarDefs>
      </GlobalDefs>
      <LocalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
         <Variable VarName="strSQL" VarID="90"> </Variable>
      </VarRefs>
      </LocalRefs>
      <LocalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
      </VarDefs>
      </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="43" ModuleName="GetRevenue" ParentModName="Sales">
   <FormalParameters>
      <PassByRef>
         <Parameter VarName="intSalesModel" VarID="91" FirstUse="REF" LastUse="REF">
</Parameter>
         <Parameter VarName="curCostPerUnit" VarID="92" FirstUse="REF" LastUse="REF">
</Parameter>
         <Parameter VarName="curAdvCost" VarID="93" FirstUse="REF" LastUse="REF">
</Parameter>
         <Parameter VarName="intSalesPeriod" VarID="94" FirstUse="REF" LastUse="REF">
</Parameter>
         <Parameter VarName="lngUnitsPerMonth" VarID="95" FirstUse="REF" LastUse="REF">
</Parameter>
         <Parameter VarName="bolIsDiscount" VarID="96" FirstUse="" LastUse="">
</Parameter>
         <Parameter VarName="strBookTitle" VarID="97" FirstUse="REF" LastUse="REF">
</Parameter>
      </PassByRef>
      <PassByVal>
      </PassByVal>
   </FormalParameters>
   <ConstantDeclarations>
   </ConstantDeclarations>
   <VariableDeclarations>
      <Variable VarName="i" VarID="98"> </Variable>
      <Variable VarName="iOldBound" VarID="99"> </Variable>
      <Variable VarName="iNewBound" VarID="100"> </Variable>
   </VariableDeclarations>
   <CalledModules>
      <Module ModuleName="GetPubRevenue" ModuleID="45"> </Module>
      <Module ModuleName="ServerMsg" ModuleID="35"> </Module>
      <Module ModuleName="GetAuthorRoyalty" ModuleID="44"> </Module>
   </CalledModules>
   <CallSites>
      <CallSite>
         <Module ModuleName="GetPubRevenue" ModuleID="45" InModuleCollection="Sales"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>
            <ActualParameter VarName="strBookTitle" VarID="97"> </ActualParameter>
            <PassByRef>
```

```
                <Parameter VarName="strTitle" VarID="106" FirstUse="REF" LastUse="REF">
</Parameter>
                </PassByRef>


            </ParameterMapping>
        </CallSite>
        <CallSite>
            <Module ModuleName="ServerMsg" ModuleID="35" InModuleCollection="ServerMain"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
        <CallSite>
            <Module ModuleName="GetAuthorRoyalty" ModuleID="44" InModuleCollection="Sales"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
        <CallSite>
            <Module ModuleName="ServerMsg" ModuleID="35" InModuleCollection="ServerMain"/>
            <StatementLineNumber/>
            <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
            <ParameterMapping>


            </ParameterMapping>
        </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
        <Constant ConstName="gRoyalty" ConstID="58"> </Constant>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="gobjServer" VarID="0"> </Variable>
        <Variable VarName="gintSalesModel" VarID="62"> </Variable>
        <Variable VarName="sngBookPrice" VarID="86"> </Variable>
        <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
        <Variable VarName="gCN" VarID="60"> </Variable>
        <Variable VarName="strBookTitle" VarID="97"> </Variable>
        <Variable VarName="lngUnitsPerMonth" VarID="95"> </Variable>
        <Variable VarName="intSalesPeriod" VarID="94"> </Variable>
        <Variable VarName="curAdvCost" VarID="93"> </Variable>
        <Variable VarName="curCostPerUnit" VarID="92"> </Variable>
        <Variable VarName="intSalesModel" VarID="91"> </Variable>
        <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
        <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
        <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
        <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
        <Variable VarName="glngUnitsPerMonth" VarID="63"> </Variable>
        <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
        <Variable VarName="gcurAdvertisingCost" VarID="73"> </Variable>
        <Variable VarName="gcurCostPerUnit" VarID="64"> </Variable>
        <Variable VarName="gintSalesModel" VarID="62"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
        <Variable VarName="i" VarID="98"> </Variable>
        <Variable VarName="iOldBound" VarID="99"> </Variable>
    </VarRefs>
    </LocalRefs>
```

```
   <LocalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="i" VarID="98"> </Variable>
      <Variable VarName="iOldBound" VarID="99"> </Variable>
   </VarDefs>
   </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="44" ModuleName="GetAuthorRoyalty" ParentModName="Sales">
   <FormalParameters>
      <PassByRef>
      </PassByRef>
      <PassByVal>
      </PassByVal>
   </FormalParameters>
   <ConstantDeclarations>
   </ConstantDeclarations>
   <VariableDeclarations>
      <Variable VarName="i" VarID="101"> </Variable>
      <Variable VarName="cGrossMonthlySalary" VarID="102"> </Variable>
      <Variable VarName="cTaxAmount" VarID="103"> </Variable>
      <Variable VarName="cTotalRevenue" VarID="104"> </Variable>
      <Variable VarName="objTax" VarID="105"> </Variable>
   </VariableDeclarations>
   <CalledModules>
      <Module ModuleName="CalcNationalIncomeTax" ModuleID="51"> </Module>
      <Module ModuleName="CalcSalesTax" ModuleID="52"> </Module>
   </CalledModules>
   <CallSites>
      <CallSite>
         <Module ModuleName="CalcNationalIncomeTax" ModuleID="51"
InModuleCollection="Taxes"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>
            <ActualParameter VarName="cGrossMonthlySalary" VarID="102"> </ActualParameter>
            <PassByRef>
               <Parameter VarName="cGrossSalary" VarID="114" FirstUse="REF" LastUse="REF">
</Parameter>
               </PassByRef>


         </ParameterMapping>
      </CallSite>
      <CallSite>
         <Module ModuleName="CalcSalesTax" ModuleID="52" InModuleCollection="Taxes"/>
         <StatementLineNumber/>
         <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
         <ParameterMapping>
            <ActualParameter VarName="cGrossMonthlySalary" VarID="102"> </ActualParameter>
            <PassByRef>
               <Parameter VarName="cGrossSalary" VarID="115" FirstUse="REF" LastUse="REF">
</Parameter>
               </PassByRef>


         </ParameterMapping>
      </CallSite>
   </CallSites>
   <GlobalRefs>
   <ConstRefs>
      <Constant ConstName="gRoyalty" ConstID="58"> </Constant>
   </ConstRefs>
   <VarRefs>
      <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
      <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
      <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
   </VarRefs>
   </GlobalRefs>
   <GlobalDefs>
   <ConstDefs>
   </ConstDefs>
   <VarDefs>
      <Variable VarName="sngAuthorRoyalty" VarID="85"> </Variable>
   </VarDefs>
```

```
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="objTax" VarID="105"> </Variable>
       <Variable VarName="cGrossMonthlySalary" VarID="102"> </Variable>
       <Variable VarName="i" VarID="101"> </Variable>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="cGrossMonthlySalary" VarID="102"> </Variable>
       <Variable VarName="i" VarID="101"> </Variable>
       <Variable VarName="objTax" VarID="105"> </Variable>
    </VarDefs>
    </LocalDefs>
 </CallGraphNode>
 <CallGraphNode NodeID="45" ModuleName="GetPubRevenue" ParentModName="Sales">
    <FormalParameters>
       <PassByRef>
          <Parameter VarName="strTitle" VarID="106" FirstUse="REF" LastUse="REF">
</Parameter>
       </PassByRef>
       <PassByVal>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
       <Variable VarName="sn" VarID="107"> </Variable>
       <Variable VarName="strSQL" VarID="108"> </Variable>
       <Variable VarName="i" VarID="109"> </Variable>
       <Variable VarName="Price" VarID="110"> </Variable>
       <Variable VarName="objModel" VarID="111"> </Variable>
       <Variable VarName="strOldTitle" VarID="112"> </Variable>
       <Variable VarName="cUnitPrice" VarID="113"> </Variable>
    </VariableDeclarations>
    <CalledModules>
       <Module ModuleName="intGetMonthSales" ModuleID="37"> </Module>
    </CalledModules>
    <CallSites>
       <CallSite>
          <Module ModuleName="intGetMonthSales" ModuleID="37" InModuleCollection="Model"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>
             <ActualParameter VarName="i" VarID="109"> </ActualParameter>
             <PassByRef>
                <Parameter VarName="intCurMonth" VarID="77" FirstUse="REF" LastUse="REF">
</Parameter>
             </PassByRef>
             <ActualParameter VarName="gintSalesPeriod" VarID="67"> </ActualParameter>
             <PassByRef>
                <Parameter VarName="intSalesPeriod" VarID="78" FirstUse="REF"
LastUse="REF"> </Parameter>
             </PassByRef>
             <ActualParameter VarName="gintSalesModel" VarID="62"> </ActualParameter>
             <PassByRef>
                <Parameter VarName="intModelType" VarID="79" FirstUse="REF" LastUse="REF">
</Parameter>
             </PassByRef>


          </ParameterMapping>
       </CallSite>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="gobjServer" VarID="0"> </Variable>
       <Variable VarName="strTitle" VarID="106"> </Variable>
       <Variable VarName="gintSalesModel" VarID="62"> </Variable>
```

```
        <Variable VarName="sngBookPrice" VarID="86"> </Variable>
        <Variable VarName="gintSalesPeriod" VarID="67"> </Variable>
        <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
        <Variable VarName="gCN" VarID="60"> </Variable>
      </VarRefs>
      </GlobalRefs>
      <GlobalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
        <Variable VarName="sngPubRevenue" VarID="84"> </Variable>
      </VarDefs>
      </GlobalDefs>
      <LocalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
        <Variable VarName="objModel" VarID="111"> </Variable>
        <Variable VarName="cUnitPrice" VarID="113"> </Variable>
        <Variable VarName="i" VarID="109"> </Variable>
        <Variable VarName="strSQL" VarID="108"> </Variable>
        <Variable VarName="sn" VarID="107"> </Variable>
        <Variable VarName="strOldTitle" VarID="112"> </Variable>
      </VarRefs>
      </LocalRefs>
      <LocalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
        <Variable VarName="strOldTitle" VarID="112"> </Variable>
        <Variable VarName="i" VarID="109"> </Variable>
        <Variable VarName="cUnitPrice" VarID="113"> </Variable>
        <Variable VarName="sn" VarID="107"> </Variable>
        <Variable VarName="strSQL" VarID="108"> </Variable>
        <Variable VarName="objModel" VarID="111"> </Variable>
      </VarDefs>
      </LocalDefs>
    </CallGraphNode>
    <CallGraphNode NodeID="46" ModuleName="Class_Initialize" ParentModName="Sales">
      <CalledModules>
        <Module ModuleName="LoadDB" ModuleID="47"> </Module>
        <Module ModuleName="ServerMsg" ModuleID="35"> </Module>
      </CalledModules>
      <CallSites>
        <CallSite>
          <Module ModuleName="LoadDB" ModuleID="47" InModuleCollection="Sales"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>


          </ParameterMapping>
        </CallSite>
        <CallSite>
          <Module ModuleName="ServerMsg" ModuleID="35" InModuleCollection="ServerMain"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>


          </ParameterMapping>
        </CallSite>
      </CallSites>
      <GlobalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
        <Variable VarName="gCN" VarID="60"> </Variable>
        <Variable VarName="gDBName" VarID="5"> </Variable>
        <Variable VarName="gintInstanceCount" VarID="59"> </Variable>
      </VarRefs>
      </GlobalRefs>
      <GlobalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
```

```
        <Variable VarName="gCN" VarID="60"> </Variable>
        <Variable VarName="gDBName" VarID="5"> </Variable>
        <Variable VarName="gintInstanceCount" VarID="59"> </Variable>
      </VarDefs>
      </GlobalDefs>
      <LocalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
      </VarRefs>
      </LocalRefs>
      <LocalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
      </VarDefs>
      </LocalDefs>
  </CallGraphNode>
  <CallGraphNode NodeID="47" ModuleName="LoadDB" ParentModName="Sales">
      <CalledModules>
        <Module ModuleName="GetBooksale" ModuleID="48"> </Module>
      </CalledModules>
      <CallSites>
        <CallSite>
          <Module ModuleName="GetBooksale" ModuleID="48" InModuleCollection="Sales"/>
          <StatementLineNumber/>
          <CallSiteAnalysisCompleted>1</CallSiteAnalysisCompleted>
          <ParameterMapping>


          </ParameterMapping>
        </CallSite>
      </CallSites>
      <GlobalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
        <Variable VarName="gDBName" VarID="5"> </Variable>
        <Variable VarName="gCN" VarID="60"> </Variable>
      </VarRefs>
      </GlobalRefs>
      <GlobalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
        <Variable VarName="gCN" VarID="60"> </Variable>
        <Variable VarName="gDBName" VarID="5"> </Variable>
      </VarDefs>
      </GlobalDefs>
      <LocalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
      </VarRefs>
      </LocalRefs>
      <LocalDefs>
      <ConstDefs>
      </ConstDefs>
      <VarDefs>
      </VarDefs>
      </LocalDefs>
  </CallGraphNode>
  <CallGraphNode NodeID="48" ModuleName="GetBooksale" ParentModName="Sales">
      <CalledModules>
      </CalledModules>
      <CallSites>
      </CallSites>
      <GlobalRefs>
      <ConstRefs>
      </ConstRefs>
      <VarRefs>
        <Variable VarName="gDBName" VarID="5"> </Variable>
      </VarRefs>
      </GlobalRefs>
      <GlobalDefs>
      <ConstDefs>
```

```
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="49" ModuleName="Class_Terminate" ParentModName="Sales">
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="gintInstanceCount" VarID="59"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
       <Variable VarName="gintInstanceCount" VarID="59"> </Variable>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>

<ModuleCollection>Taxes</ModuleCollection>
<CallGraphNode NodeID="51" ModuleName="CalcNationalIncomeTax" ParentModName="Taxes">
    <FormalParameters>
       <PassByRef>
          <Parameter VarName="cGrossSalary" VarID="114" FirstUse="REF" LastUse="REF">
</Parameter>
       </PassByRef>
       <PassByVal>
       </PassByVal>
    </FormalParameters>
    <ConstantDeclarations>
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="cGrossSalary" VarID="114"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
```

```
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
  </GlobalDefs>
  <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
  </LocalRefs>
  <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
  </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="52" ModuleName="CalcSalesTax" ParentModName="Taxes">
  <FormalParameters>
    <PassByRef>
      <Parameter VarName="cGrossSalary" VarID="115" FirstUse="REF" LastUse="REF">
</Parameter>
      <Parameter VarName="cYearToDate" VarID="116" FirstUse="REF" LastUse="REF">
</Parameter>
    </PassByRef>
    <PassByVal>
    </PassByVal>
  </FormalParameters>
  <ConstantDeclarations>
  </ConstantDeclarations>
  <VariableDeclarations>
  </VariableDeclarations>
  <CalledModules>
  </CalledModules>
  <CallSites>
  </CallSites>
  <GlobalRefs>
  <ConstRefs>
  </ConstRefs>
  <VarRefs>
    <Variable VarName="cGrossSalary" VarID="115"> </Variable>
    <Variable VarName="cYearToDate" VarID="116"> </Variable>
  </VarRefs>
  </GlobalRefs>
  <GlobalDefs>
  <ConstDefs>
  </ConstDefs>
  <VarDefs>
  </VarDefs>
  </GlobalDefs>
  <LocalRefs>
  <ConstRefs>
  </ConstRefs>
  <VarRefs>
  </VarRefs>
  </LocalRefs>
  <LocalDefs>
  <ConstDefs>
  </ConstDefs>
  <VarDefs>
  </VarDefs>
  </LocalDefs>
</CallGraphNode>
<CallGraphNode NodeID="53" ModuleName="CalcRegionalIncomeTax" ParentModName="Taxes">
  <FormalParameters>
    <PassByRef>
      <Parameter VarName="cGrossSalary" VarID="117" FirstUse="REF" LastUse="REF">
</Parameter>
      <Parameter VarName="strState" VarID="118" FirstUse="REF" LastUse="REF">
</Parameter>
    </PassByRef>
    <PassByVal>
    </PassByVal>
  </FormalParameters>
  <ConstantDeclarations>
```

```
    </ConstantDeclarations>
    <VariableDeclarations>
    </VariableDeclarations>
    <CalledModules>
    </CalledModules>
    <CallSites>
    </CallSites>
    <GlobalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
       <Variable VarName="cGrossSalary" VarID="117"> </Variable>
       <Variable VarName="strState" VarID="118"> </Variable>
    </VarRefs>
    </GlobalRefs>
    <GlobalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </GlobalDefs>
    <LocalRefs>
    <ConstRefs>
    </ConstRefs>
    <VarRefs>
    </VarRefs>
    </LocalRefs>
    <LocalDefs>
    <ConstDefs>
    </ConstDefs>
    <VarDefs>
    </VarDefs>
    </LocalDefs>
</CallGraphNode>
</CallGraph>

<!-- Project Analyzer 5.0.07 (8/6/00) book_cli.vbp v6 2 8175 -->
```

B.1.5. Integrated system XML parameter mapping graph

```
<?xml version="1.0"?>
<!--DOCTYPE CallGraph SYSTEM "file://CallGraph.dtd"-->
<!-- META NAME="Generator" CONTENT="Project Analyzer 5.0.07" -->
<!-- meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" -->
<!-- Project Analyzer Report / sys7_pmg_s3.xml -->
<!-- Project: Book_cli.vbp -->
<!-- Data Dependency Report - Parameter Mapping -->
<DataDependenceReport>
   <ModuleName ID="2" Name="lGetAuthors">
      <DataDependence>
      </DataDependence>
   </ModuleName>
   <ModuleName ID="3" Name="lGetTitles">
      <DataDependence>
         <FormalParameter ID="10" Name="strAuthor">
         </FormalParameter>
      </DataDependence>
   </ModuleName>
   <ModuleName ID="5" Name="cmdClose_Click">
      <DataDependence>
      </DataDependence>
   </ModuleName>
   <ModuleName ID="6" Name="Form_Load">
      <DataDependence>
      </DataDependence>
   </ModuleName>
   <ModuleName ID="7" Name="SetGraphData">
      <DataDependence>
      </DataDependence>
   </ModuleName>
   <ModuleName ID="9" Name="cboAuthors_Click">
      <DataDependence>
      </DataDependence>
   </ModuleName>
```

```
<ModuleName ID="10" Name="chkDiscount_Click">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="11" Name="cmdClose_Click">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="12" Name="cmdCogs_Click">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="13" Name="cmdHelp_Click">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="14" Name="cmdExecute_Click">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="15" Name="GetBooksale">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="16" Name="LoadDB">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="17" Name="Form_Load">
   <DataDependence>
   </DataDependence>
</ModuleName>
<ModuleName ID="18" Name="Form_Unload">
   <DataDependence>
      <FormalParameter ID="23" Name="Cancel">
      </FormalParameter>
   </DataDependence>
</ModuleName>
<ModuleName ID="19" Name="optAnalysis_Click">
   <DataDependence>
      <FormalParameter ID="24" Name="Index">
      </FormalParameter>
   </DataDependence>
</ModuleName>
<ModuleName ID="20" Name="txtRevParm_GotFocus">
   <DataDependence>
      <FormalParameter ID="25" Name="Index">
      </FormalParameter>
   </DataDependence>
</ModuleName>
<ModuleName ID="21" Name="udDiscount_DownClick">
   <DataDependence>
      <FormalParameter ID="26" Name="Index">
      </FormalParameter>
   </DataDependence>
</ModuleName>
<ModuleName ID="22" Name="udDiscount_UpClick">
   <DataDependence>
      <FormalParameter ID="27" Name="Index">
      </FormalParameter>
   </DataDependence>
</ModuleName>
<ModuleName ID="24" Name="optPicColor_Click">
   <DataDependence>
      <FormalParameter ID="40" Name="Index">
      </FormalParameter>
   </DataDependence>
</ModuleName>
<ModuleName ID="25" Name="Command1_Click">
   <DataDependence>
      <FormalParameter ID="41" Name="Index">
      </FormalParameter>
   </DataDependence>
</ModuleName>
<ModuleName ID="26" Name="CalcUnitCost">
   <DataDependence>
```

```
    </DataDependence>
  </ModuleName>
  <ModuleName ID="27" Name="GetCOGS">
    <DataDependence>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="28" Name="lGetBookPages">
    <DataDependence>
      <FormalParameter ID="47" Name="strTitle">
      </FormalParameter>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="29" Name="Form_Load">
    <DataDependence>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="30" Name="optBinding_Click">
    <DataDependence>
      <FormalParameter ID="52" Name="Index">
      </FormalParameter>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="31" Name="optPaperGrade_Click">
    <DataDependence>
      <FormalParameter ID="53" Name="Index">
      </FormalParameter>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="34" Name="Main">
    <DataDependence>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="35" Name="ServerMsg">
    <DataDependence>
      <FormalParameter ID="74" Name="rstrMessage">
      </FormalParameter>
      <FormalParameter ID="75" Name="rintButtons">
      </FormalParameter>
      <FormalParameter ID="76" Name="rstrTitle">
      </FormalParameter>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="37" Name="intGetMonthSales">
    <DataDependence>
      <FormalParameter ID="77" Name="intCurMonth">
        <Variables>
          <Variable ID="109" Name="i"></Variable>
          <Procedure ID="45" Name="GetPubRevenue"></Procedure>
        </Variables>
      </FormalParameter>
      <FormalParameter ID="78" Name="intSalesPeriod">
        <Variables>
          <Variable ID="67" Name="gintSalesPeriod"></Variable>
          <Procedure ID="45" Name="GetPubRevenue"></Procedure>
        </Variables>
      </FormalParameter>
      <FormalParameter ID="79" Name="intModelType">
        <Variables>
          <Variable ID="62" Name="gintSalesModel"></Variable>
          <Procedure ID="45" Name="GetPubRevenue"></Procedure>
        </Variables>
      </FormalParameter>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="39" Name="GetAuthors">
    <DataDependence>
    </DataDependence>
  </ModuleName>
  <ModuleName ID="40" Name="GetTitles">
    <DataDependence>
      <FormalParameter ID="88" Name="strSQL">
        <Variables>
          <Variable ID="11" Name="strSQL"></Variable>
          <Procedure ID="3" Name="lGetTitles"></Procedure>
        </Variables>
      </FormalParameter>
```

```
    </DataDependence>
</ModuleName>
<ModuleName ID="41" Name="GetBookPages">
    <DataDependence>
        <FormalParameter ID="89" Name="strSQL">
            <Variables>
                <Variable ID="48" Name="strSQL"></Variable>
                <Procedure ID="28" Name="lGetBookPages"></Procedure>
            </Variables>
        </FormalParameter>
    </DataDependence>
</ModuleName>
<ModuleName ID="42" Name="GetRsCOGS">
    <DataDependence>
        <FormalParameter ID="90" Name="strSQL">
            <Variables>
                <Variable ID="45" Name="strSQL"></Variable>
                <Procedure ID="27" Name="GetCOGS"></Procedure>
            </Variables>
        </FormalParameter>
    </DataDependence>
</ModuleName>
<ModuleName ID="43" Name="GetRevenue">
    <DataDependence>
        <FormalParameter ID="91" Name="intSalesModel">
        </FormalParameter>
        <FormalParameter ID="92" Name="curCostPerUnit">
        </FormalParameter>
        <FormalParameter ID="93" Name="curAdvCost">
        </FormalParameter>
        <FormalParameter ID="94" Name="intSalesPeriod">
        </FormalParameter>
        <FormalParameter ID="95" Name="lngUnitsPerMonth">
        </FormalParameter>
        <FormalParameter ID="96" Name="bolIsDiscount">
        </FormalParameter>
        <FormalParameter ID="97" Name="strBookTitle">
        </FormalParameter>
    </DataDependence>
</ModuleName>
<ModuleName ID="44" Name="GetAuthorRoyalty">
    <DataDependence>
    </DataDependence>
</ModuleName>
<ModuleName ID="45" Name="GetPubRevenue">
    <DataDependence>
        <FormalParameter ID="106" Name="strTitle">
            <Variables>
                <Variable ID="97" Name="strBookTitle"></Variable>
                <Procedure ID="43" Name="GetRevenue"></Procedure>
            </Variables>
        </FormalParameter>
    </DataDependence>
</ModuleName>
<ModuleName ID="46" Name="Class_Initialize">
    <DataDependence>
    </DataDependence>
</ModuleName>
<ModuleName ID="47" Name="LoadDB">
    <DataDependence>
    </DataDependence>
</ModuleName>
<ModuleName ID="48" Name="GetBooksale">
    <DataDependence>
    </DataDependence>
</ModuleName>
<ModuleName ID="49" Name="Class_Terminate">
    <DataDependence>
    </DataDependence>
</ModuleName>
<ModuleName ID="51" Name="CalcNationalIncomeTax">
    <DataDependence>
        <FormalParameter ID="114" Name="cGrossSalary">
            <Variables>
                <Variable ID="102" Name="cGrossMonthlySalary"></Variable>
                <Procedure ID="44" Name="GetAuthorRoyalty"></Procedure>
```

```
            </Variables>
         </FormalParameter>
      </DataDependence>
   </ModuleName>
   <ModuleName ID="52" Name="CalcSalesTax">
      <DataDependence>
         <FormalParameter ID="115" Name="cGrossSalary">
            <Variables>
               <Variable ID="102" Name="cGrossMonthlySalary"></Variable>
               <Procedure ID="44" Name="GetAuthorRoyalty"></Procedure>
            </Variables>
         </FormalParameter>
         <FormalParameter ID="116" Name="cYearToDate">
         </FormalParameter>
      </DataDependence>
   </ModuleName>
   <ModuleName ID="53" Name="CalcRegionalIncomeTax">
      <DataDependence>
         <FormalParameter ID="117" Name="cGrossSalary">
         </FormalParameter>
         <FormalParameter ID="118" Name="strState">
         </FormalParameter>
      </DataDependence>
   </ModuleName>
</DataDependenceReport>

<!-- Project Analyzer 5.0.07 (8/14/00) book_cli.vbp v6.2.8175 -->
```

B.1.6. Integrated system XML reverse ripple graph

```
<?xml version="1.0"?>
<!--DOCTYPE CallGraph SYSTEM "file://CallGraph.dtd"-->
<!-- META NAME="Generator" CONTENT="Project Analyzer 5.0.07" -->
<!-- meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" -->
<!-- Project Analyzer Report / sys7_rrg_s3.xml -->
<!-- Project: Book_cli.vbp -->
<!-- Data Dependency Report - Reverse Ripple Analysis -->
<ReverseRipple>
   <CallGraphNode ID="2" Name="lGetAuthors" ParentModName="ClientMain">
      <GlobalRefVar ID="1" Name="gobjServer">
         <ImpactedBy>
            <Module ID="17" Name="Form_Load"></Module>
            <Variable ID="1" Name="gobjServer"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="9" Name="rsAuthors">
         <ImpactedBy>
            <Module ID="2" Name="lGetAuthors"></Module>
            <Variable ID="9" Name="rsAuthors"></Variable>
         </ImpactedBy>
         <ImpactedBy>
            <Module ID="17" Name="Form_Load"></Module>
            <Variable ID="9" Name="rsAuthors"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="54" Name="rsAuthors">
         <ImpactedBy>
            <Module ID="2" Name="lGetAuthors"></Module>
            <Variable ID="54" Name="rsAuthors"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="60" Name="gCN">
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="3" Name="lGetTitles" ParentModName="ClientMain">
      <GlobalRefVar ID="1" Name="gobjServer">
      </GlobalRefVar>
      <GlobalRefVar ID="10" Name="strAuthor">
      </GlobalRefVar>
      <GlobalRefVar ID="55" Name="rsTitles">
         <ImpactedBy>
            <Module ID="3" Name="lGetTitles"></Module>
            <Variable ID="55" Name="rsTitles"></Variable>
```

```
      </ImpactedBy>
   </GlobalRefVar>
   <GlobalRefVar ID="60" Name="gCN">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="5" Name="cmdClose_Click" ParentModName="frmChart">
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="6" Name="Form_Load" ParentModName="frmChart">
   <GlobalRefVar ID="1" Name="gobjServer">
   </GlobalRefVar>
   <GlobalRefVar ID="2" Name="goStatusPanel">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="7" Name="SetGraphData" ParentModName="frmChart">
   <GlobalRefVar ID="1" Name="gobjServer">
   </GlobalRefVar>
   <GlobalRefVar ID="2" Name="goStatusPanel">
   </GlobalRefVar>
   <GlobalRefVar ID="88" Name="strSQL">
   </GlobalRefVar>
   <GlobalRefVar ID="89" Name="strSQL">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="9" Name="cboAuthors_Click" ParentModName="frmRevenue">
   <GlobalRefVar ID="1" Name="gobjServer">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="10" Name="chkDiscount_Click" ParentModName='frmRevenue">
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="11" Name="cmdClose_Click" ParentModName="frmRevenue">
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="12" Name="cmdCogs_Click" ParentModName="frmRevenue">
   <GlobalRefVar ID="2" Name="goStatusPanel">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="13" Name="cmdHelp_Click" ParentModName="frmRevenue">
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="14" Name="cmdExecute_Click" ParentModName="frmRevenue">
   <GlobalRefVar ID="2" Name="goStatusPanel">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="15" Name="GetBooksale" ParentModName="frmRevenue">
   <GlobalRefVar ID="5" Name="gDBName">
      <ImpactedBy>
         <Module ID="16" Name="LoadDB"></Module>
         <Variable ID="5" Name="gDBName"></Variable>
      </ImpactedBy>
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="16" Name="LoadDB" ParentModName="frmRevenue">
   <GlobalRefVar ID="5" Name="gDBName">
      <ImpactedBy>
         <Module ID="16" Name="LoadDB"></Module>
         <Variable ID="5" Name="gDBName"></Variable>
      </ImpactedBy>
   </GlobalRefVar>
   <GlobalRefVar ID="60" Name="gCN">
      <ImpactedBy>
         <Module ID="16" Name="LoadDB"></Module>
         <Variable ID="60" Name="gCN"></Variable>
      </ImpactedBy>
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="17" Name="Form_Load" ParentModName="frmRevenue">
```

```
    <GlobalRefVar ID="2" Name="goStatusPanel">
      <ImpactedBy>
        <Module ID="17" Name="Form_Load"></Module>
        <Variable ID="2" Name="goStatusPanel"></Variable>
      </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="1" Name="gobjServer">
      <ImpactedBy>
        <Module ID="17" Name="Form_Load"></Module>
        <Variable ID="1" Name="gobjServer"></Variable>
      </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="9" Name="rsAuthors">
      <ImpactedBy>
        <Module ID="17" Name="Form_Load"></Module>
        <Variable ID="9" Name="rsAuthors"></Variable>
      </ImpactedBy>
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="18" Name="Form_Unload" ParentModName="frmRevenue">
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="19" Name="optAnalysis_Click" ParentModName="frmRevenue">
    <GlobalRefVar ID="24" Name="Index">
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="20" Name="txtRevParm_GotFocus" ParentModName="frmRevenue">
    <GlobalRefVar ID="25" Name="Index">
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="21" Name="udDiscount_DownClick" ParentModName="frmRevenue">
    <GlobalRefVar ID="26" Name="Index">
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="22" Name="udDiscount_UpClick" ParentModName="frmRevenue">
    <GlobalRefVar ID="27" Name="Index">
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="24" Name="optPicColor_Click" ParentModName="frmCogs">
    <GlobalRefVar ID="42" Name="sHelpString">
    </GlobalRefVar>
    <GlobalRefVar ID="44" Name="fld">
    </GlobalRefVar>
    <GlobalRefVar ID="40" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="25" Name="Index">
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="25" Name="Command1_Click" ParentModName="frmCogs">
    <GlobalRefVar ID="26" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="27" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="40" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="41" Name="Index">
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="26" Name="CalcUnitCost" ParentModName="frmCogs">
    <GlobalRefVar ID="25" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="44" Name="fld">
    </GlobalRefVar>
  </CallGraphNode>
  <!-- ************************************ -->
  <CallGraphNode ID="27" Name="GetCOGS" ParentModName="frmCogs">
    <GlobalRefVar ID="1" Name="gobjServer">
    </GlobalRefVar>
    <GlobalRefVar ID="2" Name="goStatusPanel">
```

```
    </GlobalRefVar>
    <GlobalRefVar ID="6" Name="gSn">
    </GlobalRefVar>
    <GlobalRefVar ID="57" Name="rsCOGS">
       <ImpactedBy>
          <Module ID="27" Name="GetCOGS"></Module>
          <Variable ID="57" Name="rsCOGS"></Variable>
       </ImpactedBy>
       <ImpactedBy>
          <Module ID="29" Name="Form_Load"></Module>
          <Variable ID="57" Name="rsCOGS"></Variable>
       </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="60" Name="gCN">
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="28" Name="lGetBookPages" ParentModName="frmCogs">
    <GlobalRefVar ID="1" Name="gobjServer">
    </GlobalRefVar>
    <GlobalRefVar ID="2" Name="goStatusPanel">
    </GlobalRefVar>
    <GlobalRefVar ID="47" Name="strTitle">
    </GlobalRefVar>
    <GlobalRefVar ID="56" Name="rsBookPages">
       <ImpactedBy>
          <Module ID="28" Name="lGetBookPages"></Module>
          <Variable ID="56" Name="rsBookPages"></Variable>
       </ImpactedBy>
       <ImpactedBy>
          <Module ID="29" Name="Form_Load"></Module>
          <Variable ID="56" Name="rsBookPages"></Variable>
       </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="60" Name="gCN">
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="29" Name="Form_Load" ParentModName="frmCogs">
    <GlobalRefVar ID="25" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="41" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="42" Name="sHelpString">
    </GlobalRefVar>
    <GlobalRefVar ID="43" Name="curTemp">
    </GlobalRefVar>
    <GlobalRefVar ID="1" Name="gobjServer">
    </GlobalRefVar>
    <GlobalRefVar ID="2" Name="goStatusPanel">
    </GlobalRefVar>
    <GlobalRefVar ID="6" Name="gSn">
    </GlobalRefVar>
    <GlobalRefVar ID="44" Name="fld">
    </GlobalRefVar>
    <GlobalRefVar ID="56" Name="rsBookPages">
       <ImpactedBy>
          <Module ID="29" Name="Form_Load"></Module>
          <Variable ID="56" Name="rsBookPages"></Variable>
       </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="57" Name="rsCOGS">
       <ImpactedBy>
          <Module ID="29" Name="Form_Load"></Module>
          <Variable ID="57" Name="rsCOGS"></Variable>
       </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="60" Name="gCN">
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="30" Name="optBinding_Click" ParentModName="frmCogs">
    <GlobalRefVar ID="41" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="44" Name="fld">
    </GlobalRefVar>
```

```
    <GlobalRefVar ID="52" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="25" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="43" Name="curTemp">
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="31" Name="optPaperGrade_Click" ParentModName="frmCogs">
    <GlobalRefVar ID="43" Name="curTemp">
    </GlobalRefVar>
    <GlobalRefVar ID="44" Name="fld">
    </GlobalRefVar>
    <GlobalRefVar ID="53" Name="Index">
    </GlobalRefVar>
    <GlobalRefVar ID="25" Name="Index">
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="34" Name="Main" ParentModName="ServerMain">
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="35" Name="ServerMsg" ParentModName="ServerMain">
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="37" Name="intGetMonthSales" ParentModName="Model">
    <GlobalRefVar ID="1" Name="gobjServer">
    </GlobalRefVar>
    <GlobalRefVar ID="77" Name="intCurMonth">
    </GlobalRefVar>
    <GlobalRefVar ID="78" Name="intSalesPeriod">
        <ImpactedBy>
            <Module ID="43" Name="GetRevenue"></Module>
            <Variable ID="67" Name="gintSalesPeriod"></Variable>
        </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="79" Name="intModelType">
        <ImpactedBy>
            <Module ID="43" Name="GetRevenue"></Module>
            <Variable ID="62" Name="gintSalesModel"></Variable>
        </ImpactedBy>
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="39" Name="GetAuthors" ParentModName="Sales">
    <GlobalRefVar ID="54" Name="rsAuthors">
        <ImpactedBy>
            <Module ID="2" Name="lGetAuthors"></Module>
            <Variable ID="54" Name="rsAuthors"></Variable>
        </ImpactedBy>
        <ImpactedBy>
            <Module ID="39" Name="GetAuthors"></Module>
            <Variable ID="54" Name="rsAuthors"></Variable>
        </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="60" Name="gCN">
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="40" Name="GetTitles" ParentModName="Sales">
    <GlobalRefVar ID="55" Name="rsTitles">
        <ImpactedBy>
            <Module ID="3" Name="lGetTitles"></Module>
            <Variable ID="55" Name="rsTitles"></Variable>
        </ImpactedBy>
        <ImpactedBy>
            <Module ID="40" Name="GetTitles"></Module>
            <Variable ID="55" Name="rsTitles"></Variable>
        </ImpactedBy>
    </GlobalRefVar>
    <GlobalRefVar ID="60" Name="gCN">
    </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="41" Name="GetBookPages" ParentModName="Sales">
    <GlobalRefVar ID="56" Name="rsBookPages">
```

```
      <ImpactedBy>
         <Module ID="28" Name="lGetBookPages"></Module>
         <Variable ID="56" Name="rsBookPages"></Variable>
      </ImpactedBy>
      <ImpactedBy>
         <Module ID="41" Name="GetBookPages"></Module>
         <Variable ID="56" Name="rsBookPages"></Variable>
      </ImpactedBy>
   </GlobalRefVar>
   <GlobalRefVar ID="60" Name="gCN">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="42" Name="GetRsCOGS" ParentModName="Sales">
   <GlobalRefVar ID="57" Name="rsCOGS">
      <ImpactedBy>
         <Module ID="27" Name="GetCOGS"></Module>
         <Variable ID="57" Name="rsCOGS"></Variable>
      </ImpactedBy>
      <ImpactedBy>
         <Module ID="42" Name="GetRsCOGS"></Module>
         <Variable ID="57" Name="rsCOGS"></Variable>
      </ImpactedBy>
   </GlobalRefVar>
   <GlobalRefVar ID="60" Name="gCN">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="43" Name="GetRevenue" ParentModName="Sales">
   <GlobalRefVar ID="88" Name="strSQL">
   </GlobalRefVar>
   <GlobalRefVar ID="89" Name="strSQL">
   </GlobalRefVar>
   <GlobalRefVar ID="91" Name="intSalesModel">
   </GlobalRefVar>
   <GlobalRefVar ID="92" Name="curCostPerUnit">
   </GlobalRefVar>
   <GlobalRefVar ID="93" Name="curAdvCost">
   </GlobalRefVar>
   <GlobalRefVar ID="94" Name="intSalesPeriod">
   </GlobalRefVar>
   <GlobalRefVar ID="95" Name="lngUnitsPerMonth">
   </GlobalRefVar>
   <GlobalRefVar ID="97" Name="strBookTitle">
   </GlobalRefVar>
   <GlobalRefVar ID="1" Name="gobjServer">
   </GlobalRefVar>
   <GlobalRefVar ID="60" Name="gCN">
   </GlobalRefVar>
   <GlobalRefVar ID="62" Name="gintSalesModel">
      <ImpactedBy>
         <Module ID="7" Name="SetGraphData"></Module>
         <Variable ID="62" Name="gintSalesModel"></Variable>
      </ImpactedBy>
      <ImpactedBy>
         <Module ID="43" Name="GetRevenue"></Module>
         <Variable ID="62" Name="gintSalesModel"></Variable>
      </ImpactedBy>
   </GlobalRefVar>
   <GlobalRefVar ID="67" Name="gintSalesPeriod">
      <ImpactedBy>
         <Module ID="7" Name="SetGraphData"></Module>
         <Variable ID="67" Name="gintSalesPeriod"></Variable>
      </ImpactedBy>
      <ImpactedBy>
         <Module ID="43" Name="GetRevenue"></Module>
         <Variable ID="67" Name="gintSalesPeriod"></Variable>
      </ImpactedBy>
   </GlobalRefVar>
   <GlobalRefVar ID="90" Name="strSQL">
   </GlobalRefVar>
</CallGraphNode>
<!-- ************************************ -->
<CallGraphNode ID="44" Name="GetAuthorRoyalty" ParentModName="Sales">
   <GlobalRefVar ID="67" Name="gintSalesPeriod">
      <ImpactedBy>
```

```
            <Module ID="43" Name="GetRevenue"></Module>
            <Variable ID="67" Name="gintSalesPeriod"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="88" Name="strSQL">
      </GlobalRefVar>
      <GlobalRefVar ID="89" Name="strSQL">
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="45" Name="GetPubRevenue" ParentModName="Sales">
      <GlobalRefVar ID="60" Name="gCN">
      </GlobalRefVar>
      <GlobalRefVar ID="62" Name="gintSalesModel">
         <ImpactedBy>
            <Module ID="43" Name="GetRevenue"></Module>
            <Variable ID="62" Name="gintSalesModel"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="67" Name="gintSalesPeriod">
         <ImpactedBy>
            <Module ID="43" Name="GetRevenue"></Module>
            <Variable ID="67" Name="gintSalesPeriod"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="88" Name="strSQL">
      </GlobalRefVar>
      <GlobalRefVar ID="90" Name="strSQL">
      </GlobalRefVar>
      <GlobalRefVar ID="106" Name="strTitle">
      </GlobalRefVar>
      <GlobalRefVar ID="1" Name="gobjServer">
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="46" Name="Class_Initialize" ParentModName="Sales">
      <GlobalRefVar ID="5" Name="gDBName">
         <ImpactedBy>
            <Module ID="46" Name="Class_Initialize"></Module>
            <Variable ID="5" Name="gDBName"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="59" Name="gintInstanceCount">
         <ImpactedBy>
            <Module ID="46" Name="Class_Initialize"></Module>
            <Variable ID="59" Name="gintInstanceCount"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="60" Name="gCN">
         <ImpactedBy>
            <Module ID="46" Name="Class_Initialize"></Module>
            <Variable ID="60" Name="gCN"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="47" Name="LoadDB" ParentModName="Sales">
      <GlobalRefVar ID="5" Name="gDBName">
         <ImpactedBy>
            <Module ID="46" Name="Class_Initialize"></Module>
            <Variable ID="5" Name="gDBName"></Variable>
         </ImpactedBy>
         <ImpactedBy>
            <Module ID="47" Name="LoadDB"></Module>
            <Variable ID="5" Name="gDBName"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
      <GlobalRefVar ID="60" Name="gCN">
         <ImpactedBy>
            <Module ID="46" Name="Class_Initialize"></Module>
            <Variable ID="60" Name="gCN"></Variable>
         </ImpactedBy>
         <ImpactedBy>
            <Module ID="47" Name="LoadDB"></Module>
            <Variable ID="60" Name="gCN"></Variable>
         </ImpactedBy>
```

```
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="48" Name="GetBooksale" ParentModName="Sales">
      <GlobalRefVar ID="5" Name="gDBName">
         <ImpactedBy>
            <Module ID="47" Name="LoadDB"></Module>
            <Variable ID="5" Name="gDBName"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="49" Name="Class_Terminate" ParentModName="Sales">
      <GlobalRefVar ID="59" Name="gintInstanceCount">
         <ImpactedBy>
            <Module ID="49" Name="Class_Terminate"></Module>
            <Variable ID="59" Name="gintInstanceCount"></Variable>
         </ImpactedBy>
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="51" Name="CalcNationalIncomeTax" ParentModName="Taxes">
      <GlobalRefVar ID="114" Name="cGrossSalary">
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="52" Name="CalcSalesTax" ParentModName="Taxes">
      <GlobalRefVar ID="115" Name="cGrossSalary">
      </GlobalRefVar>
      <GlobalRefVar ID="116" Name="cYearToDate">
      </GlobalRefVar>
   </CallGraphNode>
   <!-- ************************************ -->
   <CallGraphNode ID="53" Name="CalcRegionalIncomeTax" ParentModName="Taxes">
      <GlobalRefVar ID="117" Name="cGrossSalary">
      </GlobalRefVar>
      <GlobalRefVar ID="118" Name="strState">
      </GlobalRefVar>
   </CallGraphNode>
</ReverseRipple>

<!-- Project Analyzer 5.0.07 (8/16/2000) book_cli.vbp v6.2.8175 -->
```

B.1.7. Integrated system call graph metrics view

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
ClientMain	NodeID: 2 ModuleName: lGetAuthors	1	1	1	4	2
ClientMain	NodeID: 3 ModuleName: lGetTitles	1	1	1	4	1
Module Subtotals		2	2	2	8	3

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
frmChart	NodeID: 5 ModuleName: cmdClose_Click	0	0	0	0	0
frmChart	NodeID: 6 ModuleName: Form_Load	1	1	1	10	7
frmChart	NodeID: 7 ModuleName: SetGraphData	1	1	1	10	7
Module Subtotals		2	2	2	20	14

Module	Call Graph Node	Number of	Number of	Number of Call	Number of	Number of

Collection		Called Modules	Call Sites	Sites per PM	Global Refs	Global Defs
frmRevenue	NodeID: 9 ModuleName: cboAuthors_Click	1	1	1	3	1
frmRevenue	NodeID: 10 ModuleName: chkDiscount_Click	0	0	0	0	0
frmRevenue	NodeID: 11 ModuleName: cmdClose_Click	0	0	0	0	0
frmRevenue	NodeID: 12 ModuleName: cmdCogs_Click	0	0	0	1	0
frmRevenue	NodeID: 13 ModuleName: cmdHelp_Click	0	0	0	0	0
frmRevenue	NodeID: 14 ModuleName: cmdExecute_Click	0	0	0	1	0
frmRevenue	NodeID: 15 ModuleName: GetBooksale	0	0	0	1	0
frmRevenue	NodeID: 16 ModuleName: LoadDB	1	1	1	2	2
frmRevenue	NodeID: 17 ModuleName: Form_Load	1	1	1	5	4
frmRevenue	NodeID: 18 ModuleName: Form_Unload	0	0	0	0	2
frmRevenue	NodeID: 19 ModuleName: optAnalysis_Click	0	0	0	1	0
frmRevenue	NodeID: 20 ModuleName: txtRevParm_GotFocus	0	0	0	1	0
frmRevenue	NodeID: 21 ModuleName: udDiscount_DownClick	0	0	0	1	0
frmRevenue	NodeID: 22 ModuleName: udDiscount_UpClick	0	0	0	1	0
Module Subtotals		3	3	3	17	9

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
frmCogs	NodeID: 24 ModuleName: optPicColor_Click	1	1	1	4	2
frmCogs	NodeID: 25 ModuleName: Command1_Click	0	0	0	4	1
frmCogs	NodeID: 26 ModuleName: CalcUnitCost	0	0	0	2	0
frmCogs	NodeID: 27 ModuleName: GetCOGS	1	1	1	5	4
frmCogs	NodeID: 28 ModuleName: lGetBookPages	1	1	1	5	1
frmCogs	NodeID: 29 ModuleName: Form_Load	3	3	3	11	11
frmCogs	NodeID: 30 ModuleName: optBinding_Click	2	5	5	5	3

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
frmCogs	NodeID: 31 ModuleName: optPaperGrade_Click	1	1	1	4	2
Module Subtotals		9	12	12	40	24

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
Module Subtotals		0	0	0	0	0

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
ServerMain	NodeID: 34 ModuleName: Main	0	0	0	0	0
ServerMain	NodeID: 35 ModuleName: ServerMsg	0	0	0	0	0
Module Subtotals		0	0	0	0	0

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
Model	NodeID: 37 ModuleName: intGetMonthSales	0	0	0	4	0
Module Subtotals		0	0	0	4	0

Module Collection	Call Graph Node	Number of Called Modules	Number of Call Sites	Number of Call Sites per PM	Number of Global Refs	Number of Global Defs
Sales	NodeID: 39 ModuleName. GetAuthors	0	0	0	2	1
Sales	NodeID: 40 ModuleName: GetTitles	0	0	0	2	1
Sales	NodeID: 41 ModuleName: GetBookPages	0	0	0	2	1
Sales	NodeID: 42 ModuleName: GetRsCOGS	0	0	0	2	1
Sales	NodeID: 43 ModuleName: GetRevenue	3	4	4	14	7
Sales	NodeID: 44 ModuleName: GetAuthorRoyalty	2	2	2	4	1
Sales	NodeID: 45 ModuleName: GetPubRevenue	1	1	1	7	1
Sales	NodeID: 46 ModuleName: Class_Initialize	2	2	2	3	3
Sales	NodeID: 47 ModuleName: LoadDB	1	1	1	2	2
Sales	NodeID: 48 ModuleName: GetBooksale	0	0	0	1	0
Sales	NodeID: 49 ModuleName: Class_Terminate	0	0	0	1	1
Module Subtotals		9	10	10	40	19

Module	Call Graph Node	Number of	Number of	Number of Call	Number of	Number of

Collection		Called Modules	Call Sites	Sites per PM	Global Refs	Global Defs
Taxes	NodeID: 51 ModuleName: CalcNationalIncomeTax	0	0	0	1	0
Taxes	NodeID: 52 ModuleName: CalcSalesTax	0	0	0	2	0
Taxes	NodeID: 53 ModuleName: CalcRegionalIncomeTax	0	0	0	2	0
Module Subtotals		0	0	0	5	0

Totals	
Total # of ModuleCollections	9
Total # of CallGraphNodes	44
Total # of CalledModules	25
Total # of CallSites	29
Total # of CallSites/PM	29
Total # of GlobalRefs	134
Total # of GlobalDefs	69

B.1.8. Integrated system call coupling analysis view

CallGraphNode	CallSite
NodeID="2" ModuleName="lGetAuthors" ParentModName="ClientMain"	ModuleName="GetAuthors" ModuleID="39" InModuleCollection="Sales"
NodeID="3" ModuleName="lGetTitles" ParentModName="ClientMain"	ModuleName="GetTitles" ModuleID="40" InModuleCollection="Sales"
NodeID="6" ModuleName="Form_Load" ParentModName="frmChart"	ModuleName="SetGraphData" ModuleID="7" InModuleCollection="frmChart"
NodeID="7" ModuleName="SetGraphData" ParentModName="frmChart"	ModuleName="GetRevenue" ModuleID="43" InModuleCollection="Sales"
NodeID="9" ModuleName="cboAuthors_Click" ParentModName="frmRevenue"	ModuleName="lGetTitles" ModuleID="3" InModuleCollection="ClientMain"
NodeID="16" ModuleName="LoadDB" ParentModName="frmRevenue"	ModuleName="GetBooksale" ModuleID="15" InModuleCollection="frmRevenue"
NodeID="17" ModuleName="Form_Load" ParentModName="frmRevenue"	ModuleName="lGetAuthors" ModuleID="2" InModuleCollection="ClientMain"
NodeID="24" ModuleName="optPicColor_Click" ParentModName="frmCogs"	ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"
NodeID="27" ModuleName="GetCOGS" ParentModName="frmCogs"	ModuleName="GetRsCOGS" ModuleID="42" InModuleCollection="Sales"
NodeID="28" ModuleName="lGetBookPages" ParentModName="frmCogs"	ModuleName="GetBookPages" ModuleID="41" InModuleCollection="Sales"
NodeID="29" ModuleName="Form_Load" ParentModName="frmCogs"	ModuleName="lGetBookPages" ModuleID="28" InModuleCollection="frmCogs"
NodeID="29" ModuleName="Form_Load"	ModuleName="GetCOGS" ModuleID="27"

CallGraphNode	CallSite
ParentModName="frmCogs"	InModuleCollection="frmCogs"
NodeID="29" ModuleName="Form_Load" ParentModName="frmCogs"	ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"
NodeID="30" ModuleName="optBinding_Click" ParentModName="frmCogs"	ModuleName="optPaperGrade_Click" ModuleID="31" InModuleCollection="frmCogs"
NodeID="30" ModuleName="optBinding_Click" ParentModName="frmCogs"	ModuleName="optPaperGrade_Click" ModuleID="31" InModuleCollection="frmCogs"
NodeID="30" ModuleName="optBinding_Click" ParentModName="frmCogs"	ModuleName="optPaperGrade_Click" ModuleID="31" InModuleCollection="frmCogs"
NodeID="30" ModuleName="optBinding_Click" ParentModName="frmCogs"	ModuleName="optPaperGrade_Click" ModuleID="31" InModuleCollection="frmCogs"
NodeID="30" ModuleName="optBinding_Click" ParentModName="frmCogs"	ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"
NodeID="31" ModuleName="optPaperGrade_Click" ParentModName="frmCogs"	ModuleName="CalcUnitCost" ModuleID="26" InModuleCollection="frmCogs"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	ModuleName="GetPubRevenue" ModuleID="45" InModuleCollection="Sales"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	ModuleName="ServerMsg" ModuleID="35" InModuleCollection="ServerMain"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	ModuleName="GetAuthorRoyalty" ModuleID="44" InModuleCollection="Sales"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	ModuleName="ServerMsg" ModuleID="35" InModuleCollection="ServerMain"
NodeID="44" ModuleName="GetAuthorRoyalty" ParentModName="Sales"	ModuleName="CalcNationalIncomeTax" ModuleID="51" InModuleCollection="Taxes"
NodeID="44" ModuleName="GetAuthorRoyalty" ParentModName="Sales"	ModuleName="CalcSalesTax" ModuleID="52" InModuleCollection="Taxes"
NodeID="45" ModuleName="GetPubRevenue" ParentModName="Sales"	ModuleName="intGetMonthSales" ModuleID="37" InModuleCollection="Model"
NodeID="46" ModuleName="Class_Initialize" ParentModName="Sales"	ModuleName="LoadDB" ModuleID="47" InModuleCollection="Sales"
NodeID="46" ModuleName="Class_Initialize" ParentModName="Sales"	ModuleName="ServerMsg" ModuleID="35" InModuleCollection="ServerMain"
NodeID="47" ModuleName="LoadDB" ParentModName="Sales"	ModuleName="GetBooksale" ModuleID="48" InModuleCollection="Sales"

Totals	
Total # of Call Couplings	29

B.1.9. Integrated system parameter coupling analysis view

CallGraphNode	CallSite	Actual Parameter	Formal Parameter
NodeID="3" ModuleName="lGetTitles" ParentModName="ClientMain"	ModuleName="GetTitles" ModuleID="40" InModuleCollection="Sales"	VarName="strSQL" VarID="11"	PassByVal VarName="strSQL" VarID="88" FirstUse="REF" LastUse="REF"
NodeID="27" ModuleName="GetCOGS" ParentModName="frmCogs"	ModuleName="GetRsCOGS" ModuleID="42" InModuleCollection="Sales"	VarName="strSQL" VarID="45"	PassByVal VarName="strSQL" VarID="90" FirstUse="REF" LastUse="REF"
NodeID="28" ModuleName="lGetBookPages" ParentModName="frmCogs"	ModuleName="GetBookPages" ModuleID="41" InModuleCollection="Sales"	VarName="strSQL" VarID="48"	PassByVal VarName="strSQL" VarID="89" FirstUse="REF" LastUse="REF"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	ModuleName="GetPubRevenue" ModuleID="45" InModuleCollection="Sales"	VarName="strBookTitle" VarID="97"	PassByRef VarName="strTitle" VarID="106" FirstUse="REF" LastUse="REF"
NodeID="44" ModuleName="GetAuthorRoyalty" ParentModName="Sales"	ModuleName="CalcNationalIncomeTax" ModuleID="51" InModuleCollection="Taxes"	VarName="cGrossMonthlySalary" VarID="102"	PassByRef VarName="cGrossSalary" VarID="114" FirstUse="REF" LastUse="REF"
NodeID="44" ModuleName="GetAuthorRoyalty" ParentModName="Sales"	ModuleName="CalcSalesTax" ModuleID="52" InModuleCollection="Taxes"	VarName="cGrossMonthlySalary" VarID="102"	PassByRef VarName="cGrossSalary" VarID="115" FirstUse="REF" LastUse="REF"
NodeID="45" ModuleName="GetPubRevenue" ParentModName="Sales"	ModuleName="intGetMonthSales" ModuleID="37" InModuleCollection="Model"	VarName="i" VarID="109"	PassByRef VarName="intCurMonth" VarID="77" FirstUse="REF" LastUse="REF"
NodeID="45" ModuleName="GetPubRevenue" ParentModName="Sales"	ModuleName="intGetMonthSales" ModuleID="37" InModuleCollection="Model"	VarName="gintSalesPeriod" VarID="67"	PassByRef VarName="intSalesPeriod" VarID="78" FirstUse="REF" LastUse="REF"
NodeID="45" ModuleName="GetPubRevenue" ParentModName="Sales"	ModuleName="intGetMonthSales" ModuleID="37" InModuleCollection="Model"	VarName="gintSalesModel" VarID="62"	PassByRef VarName="intModelType" VarID="79" FirstUse="REF" LastUse="REF"

B.1.10. Integrated system parameter mapping dependence view

Module Name	Formal Parameter	Dependence
ModuleName="intGetMonthSales" ModuleID="37"	VariableName="intCurMonth" VariableID="77"	VariableName="i" VariableID="109" ModuleName="GetPubRevenue" ModuleID="45"
ModuleName="intGetMonthSales" ModuleID="37"	VariableName="intSalesPeriod" VariableID="78"	VariableName="gintSalesPeriod" VariableID="67" ModuleName="GetPubRevenue" ModuleID="45"
ModuleName="intGetMonthSales" ModuleID="37"	VariableName="intModelType" VariableID="79"	VariableName="gintSalesModel" VariableID="62" ModuleName="GetPubRevenue" ModuleID="45"
ModuleName="GetTitles" ModuleID="40"	VariableName="strSQL" VariableID="88"	VariableName="strSQL" VariableID="11" ModuleName="lGetTitles"

Module Name	Formal Parameter	Dependence
		ModuleID="3"
ModuleName="GetBookPages" ModuleID="41"	VariableName="strSQL" VariableID="89"	VariableName="strSQL" VariableID="48" ModuleName="lGetBookPages" ModuleID="28"
ModuleName="GetRsCOGS" ModuleID="42"	VariableName="strSQL" VariableID="90"	VariableName="strSQL" VariableID="45" ModuleName="GetCOGS" ModuleID="27"
ModuleName="GetPubRevenue" ModuleID="45"	VariableName="strTitle" VariableID="106"	VariableName="strBookTitle" VariableID="97" ModuleName="GetRevenue" ModuleID="43"
ModuleName="CalcNationalIncomeTax" ModuleID="51"	VariableName="cGrossSalary" VariableID="114"	VariableName="cGrossMonthlySalary" VariableID="102" ModuleName="GetAuthorRoyalty" ModuleID="44"
ModuleName="CalcSalesTax" ModuleID="52"	VariableName="cGrossSalary" VariableID="115"	VariableName="cGrossMonthlySalary" VariableID="102" ModuleName="GetAuthorRoyalty" ModuleID="44"

Totals	
Total # of Parameter Mapping Dependencies in Call Graph:	9

B.1.11. Integrated system reverse ripple dependence view

CallGraphNode	GlobalRefVar	Impacted By
NodeID="2" ModuleName="lGetAuthors" ParentModName="ClientMain"	Name="gobjServer" ID="1"	ModuleName="Form_Load" ModuleID="17" VariableName="gobjServer" VariableID="1"
NodeID="2" ModuleName="lGetAuthors" ParentModName="ClientMain"	Name="rsAuthors" ID="9"	ModuleName="lGetAuthors" ModuleID="2" VariableName="rsAuthors" VariableID="9"
NodeID="2" ModuleName="lGetAuthors" ParentModName="ClientMain"	Name="rsAuthors" ID="9"	ModuleName="Form_Load" ModuleID="17" VariableName="rsAuthors" VariableID="9"
NodeID="2" ModuleName="lGetAuthors" ParentModName="ClientMain"	Name="rsAuthors" ID="54"	ModuleName="lGetAuthors" ModuleID="2" VariableName="rsAuthors" VariableID="54"
NodeID="3" ModuleName="lGetTitles" ParentModName="ClientMain"	Name="rsTitles" ID="55"	ModuleName="lGetTitles" ModuleID="3" VariableName="rsTitles" VariableID="55"
NodeID="15" ModuleName="GetBooksale" ParentModName="frmRevenue"	Name="gDBName" ID="5"	ModuleName="LoadDB" ModuleID="16" VariableName="gDBName" VariableID="5"
NodeID="16" ModuleName="LoadDB" ParentModName="frmRevenue"	Name="gDBName" ID="5"	ModuleName="LoadDB" ModuleID="16" VariableName="gDBName" VariableID="5"
NodeID="16" ModuleName="LoadDB" ParentModName="frmRevenue"	Name="gCN" ID="60"	ModuleName="LoadDB" ModuleID="16" VariableName="gCN" VariableID="60"
NodeID="17" ModuleName="Form_Load" ParentModName="frmRevenue"	Name="goStatusPanel" ID="2"	ModuleName="Form_Load" ModuleID="17" VariableName="goStatusPanel" VariableID="2"

CallGraphNode	GlobalRefVar	Impacted By
NodeID="17" ModuleName="Form_Load" ParentModName="frmRevenue"	Name="gobjServer" ID="1"	ModuleName="Form_Load" ModuleID="17" VariableName="gobjServer" VariableID="1"
NodeID="17" ModuleName="Form_Load" ParentModName="frmRevenue"	Name="rsAuthors" ID="9"	ModuleName="Form_Load" ModuleID="17" VariableName="rsAuthors" VariableID="9"
NodeID="27" ModuleName="GetCOGS" ParentModName="frmCogs"	Name="rsCOGS" ID="57"	ModuleName="GetCOGS" ModuleID="27" VariableName="rsCOGS" VariableID="57"
NodeID="27" ModuleName="GetCOGS" ParentModName="frmCogs"	Name="rsCOGS" ID="57"	ModuleName="Form_Load" ModuleID="29" VariableName="rsCOGS" VariableID="57"
NodeID="28" ModuleName="lGetBookPages" ParentModName="frmCogs"	Name="rsBookPages" ID="56"	ModuleName="lGetBookPages" ModuleID="28" VariableName="rsBookPages" VariableID="56"
NodeID="28" ModuleName="lGetBookPages" ParentModName="frmCogs"	Name="rsBookPages" ID="56"	ModuleName="Form_Load" ModuleID="29" VariableName="rsBookPages" VariableID="56"
NodeID="29" ModuleName="Form_Load" ParentModName="frmCogs"	Name="rsBookPages" ID="56"	ModuleName="Form_Load" ModuleID="29" VariableName="rsBookPages" VariableID="56"
NodeID="29" ModuleName="Form_Load" ParentModName="frmCogs"	Name="rsCOGS" ID="57"	ModuleName="Form_Load" ModuleID="29" VariableName="rsCOGS" VariableID="57"
NodeID="37" ModuleName="intGetMonthSales" ParentModName="Model"	Name="intSalesPeriod" ID="78"	ModuleName="GetRevenue" ModuleID="43" VariableName="gintSalesPeriod" VariableID="67"
NodeID="37" ModuleName="intGetMonthSales" ParentModName="Model"	Name="intModelType" ID="79"	ModuleName="GetRevenue" ModuleID="43" VariableName="gintSalesModel" VariableID="62"
NodeID="39" ModuleName="GetAuthors" ParentModName="Sales"	Name="rsAuthors" ID="54"	ModuleName="lGetAuthors" ModuleID="2" VariableName="rsAuthors" VariableID="54"
NodeID="39" ModuleName="GetAuthors" ParentModName="Sales"	Name="rsAuthors" ID="54"	ModuleName="GetAuthors" ModuleID="39" VariableName="rsAuthors" VariableID="54"
NodeID="40" ModuleName="GetTitles" ParentModName="Sales"	Name="rsTitles" ID="55"	ModuleName="lGetTitles" ModuleID="3" VariableName="rsTitles" VariableID="55"
NodeID="40" ModuleName="GetTitles" ParentModName="Sales"	Name="rsTitles" ID="55"	ModuleName="GetTitles" ModuleID="40" VariableName="rsTitles" VariableID="55"
NodeID="41" ModuleName="GetBookPages" ParentModName="Sales"	Name="rsBookPages" ID="56"	ModuleName="lGetBookPages" ModuleID="28" VariableName="rsBookPages" VariableID="56"
NodeID="41"	Name="rsBookPages"	ModuleName="GetBookPages"

CallGraphNode	GlobalRefVar	Impacted By
ModuleName="GetBookPages" ParentModName="Sales"	ID="56"	ModuleID="41" VariableName="rsBookPages" VariableID="56"
NodeID="42" ModuleName="GetRsCOGS" ParentModName="Sales"	Name="rsCOGS" ID="57"	ModuleName="GetCOGS" ModuleID="27" VariableName="rsCOGS" VariableID="57"
NodeID="42" ModuleName="GetRsCOGS" ParentModName="Sales"	Name="rsCOGS" ID="57"	ModuleName="GetRsCOGS" ModuleID="42" VariableName="rsCOGS" VariableID="57"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	Name="gintSalesModel" ID="62"	ModuleName="SetGraphData" ModuleID="7" VariableName="gintSalesModel" VariableID="62"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	Name="gintSalesModel" ID="62"	ModuleName="GetRevenue" ModuleID="43" VariableName="gintSalesModel" VariableID="62"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	Name="gintSalesPeriod" ID="67"	ModuleName="SetGraphData" ModuleID="7" VariableName="gintSalesPeriod" VariableID="67"
NodeID="43" ModuleName="GetRevenue" ParentModName="Sales"	Name="gintSalesPeriod" ID="67"	ModuleName="GetRevenue" ModuleID="43" VariableName="gintSalesPeriod" VariableID="67"
NodeID="44" ModuleName="GetAuthorRoyalty" ParentModName="Sales"	Name="gintSalesPeriod" ID="67"	ModuleName="GetRevenue" ModuleID="43" VariableName="gintSalesPeriod" VariableID="67"
NodeID="45" ModuleName="GetPubRevenue" ParentModName="Sales"	Name="gintSalesModel" ID="62"	ModuleName="GetRevenue" ModuleID="43" VariableName="gintSalesModel" VariableID="62"
NodeID="45" ModuleName="GetPubRevenue" ParentModName="Sales"	Name="gintSalesPeriod" ID="67"	ModuleName="GetRevenue" ModuleID="43" VariableName="gintSalesPeriod" VariableID="67"
NodeID="46" ModuleName="Class_Initialize" ParentModName="Sales"	Name="gDBName" ID="5"	ModuleName="Class_Initialize" ModuleID="46" VariableName="gDBName" VariableID="5"
NodeID="46" ModuleName="Class_Initialize" ParentModName="Sales"	Name="gintInstanceCount" ID="59"	ModuleName="Class_Initialize" ModuleID="46" VariableName="gintInstanceCount" VariableID="59"
NodeID="46" ModuleName="Class_Initialize" ParentModName="Sales"	Name="gCN" ID="60"	ModuleName="Class_Initialize" ModuleID="46" VariableName="gCN" VariableID="60"
NodeID="47" ModuleName="LoadDB" ParentModName="Sales"	Name="gDBName" ID="5"	ModuleName="Class_Initialize" ModuleID="46" VariableName="gDBName" VariableID="5"
NodeID="47" ModuleName="LoadDB" ParentModName="Sales"	Name="gDBName" ID="5"	ModuleName="LoadDB" ModuleID="47" VariableName="gDBName" VariableID="5"
NodeID="47" ModuleName="LoadDB"	Name="gCN" ID="60"	ModuleName="Class_Initialize" ModuleID="46"

CallGraphNode	GlobalRefVar	Impacted By
ParentModName="Sales"		VariableName="gCN" VariableID="60"
NodeID="47" ModuleName="LoadDB" ParentModName="Sales"	Name="gCN" ID="60"	ModuleName="LoadDB" ModuleID="47" VariableName="gCN" VariableID="60"
NodeID="48" ModuleName="GetBooksale" ParentModName="Sales"	Name="gDBName" ID="5"	ModuleName="LoadDB" ModuleID="47" VariableName="gDBName" VariableID="5"
NodeID="49" ModuleName="Class_Terminate" ParentModName="Sales"	Name="gintInstanceCount" ID="59"	ModuleName="Class_Terminate" ModuleID="49" VariableName="gintInstanceCount" VariableID="59"

Totals	
Total # of Reverse Ripple Impacts in Call Graph:	43

APPENDIX C. TOOL EXTENSIONS AND SOURCE LISTINGS

This appendix contains source code listings and other supporting details for the tools and techniques developed or modified for use in this research. The first section describes the modifications made to Project Analyzer for the OA-dead analysis capability. Note that the term semi-dead is used to mean OA-dead in the tool. The next section describes the Project Analyzer modifications made to perform a global variable usage analysis and generate the XML summary call graph, parameter mapping dependence graph, and impact dependence graphs. The XML schemas for each of these graphs are included next, followed by the XSL scripts to render the various views of the graphs.

C.1. Project analyzer extension: OA-dead analysis

C.1.1. OA-dead report modifications to project analyzer

This is a list of all modifications made to the Project Analyzer code (version 5.0.07) to incorporate the OA-Dead Report. These modifications can be quickly located in the source code by searching for the string "sparks."

1. Modified Data.bas
   a. Added init and set attributes to CtrlType (InitDisabled, SetEnabled, InitInvisible, SetVisible, etc.)
   b. Added SemiDead and Checked attribute to ProcType
   c. Added SemiDead and Checked attributes to FileType
   d. Added semidead reason attributes to ProcType and CtrlType (sdFile, sdInvisible, etc.)

2. Modified Analysis.bas
   a. Added subconditionals to ReadFormData30 to gather Visible, Enabled, Width, Height, Left, and Top information for Form Files
   b. Added subconditionals to ReadFormData40 to gather Visible, Enabled, Width, Height, Left, and Top information for Form Files
   c. Added AnalyzeCtrlRef procedure (see Appendices B and/or C for details)
   d. Added call to AnalyzeCtrlRef in Analyze_Word_Scope within "case stModule" code where it checks for a Control
   e. Added line to grab NextIdent for dotted control in IsDotted
   f. Added ParentFileNR assignment to AddCtrl

3. Modified Project.frm
   a. Added "OA-Dead Code Report" to Report menu
   b. Added Process_Phase check to Report_Click procedure

4. Added SemiDeadReportOptions.frm

5. Modified Report.bas
   a. Added procedure ReportSemiDead (see Appendices B and/or C for details)
   b. Added procedure CheckDownFile(filenr) as recursive check of SemiDead File information (see Appendices B and/or C for details)
   c. Added procedure CheckDownProc(procnr) as recursive check of SemiDead procedure information (see Appendices B and/or C for details)

C.1.2. OA-dead report pseudo-code

This pseudo-code is the basic outline for the OA-Dead Report.

1. Parse source code to look for OA-dead attributes. (Analysis.bas)

2. For each attribute found gather initialized and set information. (Analysis.bas)

3. Initialize OA-Dead analysis and count variables. (ReportSemiDead in Report.bas)

4. For each control, if any OA-dead affecting attribute is initialized unavailable and not set then the control is OA-dead. (ReportSemiDead in Report.bas)

5. For each file, if file is OA-dead, check child files for other parents who are not OA-dead. Otherwise child files are OA-dead too. (ReportSemiDead in Report.bas)

6. For each procedure, if parent file is OA-dead, procedure is OA-dead. (ReportSemiDead in Report.bas)

7. For each control, if control is OA-dead, child procedures are OA-dead. (ReportSemiDead in Report.bas)

8. For each procedure, if procedure is OA-dead, check child procedures for other parents who are not OA-dead. Otherwise child procedures are OA-dead too. (ReportSemiDead in Report.bas)

9. Get user input on what to report. (SemiDeadReportOptions.frm)

10. Report. (ReportSemiDead in Report.bas)

C.2. Project analyzer extension: call graph summary information

C.2.1. Summary information modifications to project analyzer

This is a list of all modifications made to the Project Analyzer code to incorporate the global variable usage analysis, summary call graph generation, parameter mapping analysis, and ripple analysis reports.

1. Existing storage mechanisms in project analyzer. The following array structures were used as they exist without modification:

    a. LocalIdent Array. This is an array of all local (ie, procedure-level) variables, constants and parameters.

    b. Ident Array. This is an array of all global variables and constants. Project Analyzer defines global as Module-level and higher in a VB project.

    c. IdentRef Array. This array stores variable and constant usage information. Every time a variable is referenced or defined is a unique IdentRef instance and information about that IdentRef is stored in the IdentRef array.

    d. Proc Array. This is an array of procedure information. Every procedure and function in a VB project will have an information element in the Proc array.

2. Storage mechanism extensions and modifications:

a. IDMap Array. Project Analyzer uses the Ident and LocalIdent arrays to identify variables/constants within the context of their scope. Thus, a local variable and a global variable can share the same ID value. Call graph analysis requires the ability to uniquely identify all variable/constant instances outside of their scope. To ensure unique ID key values the IDMap array was designed to map a unique context-free key value to every variable and constant defined in the project.

b. CallGraph.mdb Database. This is an MS Access database that organizes project analysis data into information tables. The CallGraph database facilitates the use of SQL queries to perform complex information analysis tasks. The alternative to the database approach would be to use repetitive sequential array processing to perform complex information analysis tasks. Following is a description of the tables used in the database:

- RefDef Table. This table replicates much of the information stored in the IdentRef array. Information describing variable/constant attributes and how they are used was copied out of each IdentRef array element into the RefDef table. SQL queries were used to manipulate variable usage information in the RefDef table. SQL queries are more efficient than repetitively traversing the IdentRef array to algorithmically determine the same information.

- ProcedureCalls Table. This table simply records caller and callee information about every procedure call site in the project. Each call site is referred to as a procedure reference or ProcRef. ProcRefs are uniquely identified by a key value, which is also stored in the ProcedureCalls table.

- ParmBindings Table. This table records actual-to-formal parameter binding information at each procedure call site. Each parameter-binding instance is associated with a unique ProcRef number, which relates it back to the ProcedureCalls table.

- Ripple Table. This is a utility table used to temporarily store ripple analysis information prior to printing the Ripple analysis report.

- Visit table. Another utility table.

3. The following software modules were added to Project Analyzer:

a. IDMap.bas This module contains the data structures and methods to manage and use the IDMap array.

b. ProcMod.bas This module contains the software to interface with the CallGraph database and perform much of the data analysis for the call graph related reports.

c. XML.bas This module contains the output procedures that generate and print XML-formatted text.

4. The following software modules were modified in Project Analyzer to support call graph report generation:

a. Analysis.bas Those portions of the analysis module that process procedure callsites were modified to record call site data into the ProcedureCalls table and parameter-binding information into the ParmBindings table in the database.

b. Data.bas, IdentMod.bas, and LocalIdentMod.bas These modules were all modified to accommodate the inclusion of a unique IDKey field in the data structure definitions of IdentType objects in Project Analyzer. Those portions of these modules that add new variables/constants to

the Ident and LocalIdent arrays were modified to register the new variable/constant with the IDMap array.

c. Report.bas The algorithms to perform data analysis necessary to generate call graph reports were implemented in the Report module.

5. Project Analyzer Analysis Phases:

a. Phases 1 & 2. During phases 1 & 2 Project Analyzer is gathering data about project structure, e.g., variable and constant declarations, procedure names, identifier usage, etc. Phase 2 primarily conducts cross-referencing analysis. Phase 2 analysis was modified to capture procedure call statements and mark each identifier serving as an actual parameter as a new identifier reference (IdentRef) object. The new IdentRef object is added to the IdentRef array and further parameter-binding analysis is performed in Phase 3.

b. Phase 3 was added to support call graph and data flow analysis. It performs two main tasks. The first task is to finalize the parameter-mapping analysis. The parameter-binding relation is determined to be either input (reference) or output (definition). Parameter-binding information is then copied into the ParmBindings table in the CallGraph database. The second task is to copy the IdentRef data into the RefDef table in the database.

C.2.2. Summary call graph pseudo-code

For each procedure (P) in the project:

```
Report Formal Parameters
        For each parameter in LocalIdentifier list
                Compute first use/Last use information
                Print "Parameter" tag
        Next parameter

Report Constant declarations
        For each constant in LocalIdentifier list
                Print "Constant" tag
        Next constant

Report Variable declarations
        For each variable in LocalIdentifier list
                Print "Variable" tag
        Next variable

Report Called Modules
        For each calledProcedure in P.ToProcs  list
                Print "Module" tag
        Next calledProcedure

Report CallSite Tags
        Get list of call sites in P
        For each callSite in P
                Print "Module" tag
                Print Parameter Bindings (ordered list of Actuals followed by Formals)
        Next callSite

Report Global References Lists
```

```
                Get list of Global Constants referenced in P
                For each ConstRef in list
                        Print "Constant" tag
                Next ConstRef

        Get list of Global Variables referenced in P
                For each VarRef in list
                        Print "Variable" tag
                Next VarRef

        Report Global Defines Lists
                Get list of Global Variables defined in P
                For each VarDef in list
                        Print "Variable" tag
                Next VarDef

        Report Local References Lists
                Get list of Local Constants referenced in P
                For each ConstRef in list
                        Print "Constant" tag
                Next ConstRef

        Get list of Local Variables referenced in P
                For each VarRef in list
                        Print "Variable" tag
                Next VarRef

        Report Local Defines Lists
                Get list of Local Constants defined in P
                For each ConstDef in list
                        Print "Constant" tag
                Next ConstDef

        Get list of Local Variables defined in P
                For each VarDef in list
                        Print "Variable" tag
                Next VarDef

Next Procedure
```

C.2.3. Global ref-def computation pseudo-code

The purpose of this algorithm is to compute the set of global variables referenced or defined directly in the body of procedure (P) and indirectly through program control flow out of P.

Global reference algorithm:

```
Select the list of Variables (VarList) from RefDef table where:
        Referencing procedure is P
        The variable is global and it is referenced, or
        The variable is a ByRef parameter and referenced
```

Generate list of indirect global variable references:

```
        Get list of procedures comprising the call tree rooted at P (CPList)
        For each called procedure (CP) in CPList
                Select list of Variables from RefDef table where:
                        Referencing procedure is CP
                        The variable is global and it is referenced
                Append selected variables to VarList
        Next called procedure

Global defines algorithm:

Select the list of Variables (VarList) from RefDef table where:
        Referencing procedure is P
        The variable is global and it is defined, or
        The variable is a ByRef parameter and defined

Generate list of indirect global variable defines:
        Get list of procedures comprising the call tree rooted at P (CPList)
        For each called procedure (CP) in CPList
                Select list of Variables from RefDef table where:
                        Referencing procedure is CP
                        The variable is global and it is defined
                Append selected variables to VarList
        Next called procedure
```

C.2.4. Reverse ripple analysis report pseudo-code

```
clear marked entries/initialize
sub main()
        For each callgrahpnode
                CGN = CallGraphNode name/id
                for each GlobalRef variable
                        GRV = GlobalRef Variable name/id
                        If GRV in GlobalDef list of CGN then Mark CGN/GRV pair
                        for each CalledByModule in CGN
                                CBMOD = CalledByModule name/id
                                Visit_Module(CBMOD, GRV)
                                next
                        end for
                        print marked entries
                        clear marked entries
                        next
                end for
                next
        end for
end sub

sub Visit_Module(cgn, v) : boolean
        If cgn has already been visited then Return

        If v is a formalparameter then
                for each callsite in cgn
                        act_v = mapped actual parameter of v
                        Visit_Module(cgn, act_v)
```

```
                    next
                end for
        else
                If v in GlobalDef list of cgn then
                    Mark cgn/v pair
                else
                    for each CalledByModule in cgn
                        cbm = CalledByModule name/id
                        Visit_Module(cbm, v)
                        next
                    end for
                end if
        end if
end sub
```

C.2.5. Parameter mapping dependence report pseudo-code

```
For each procedure (P) in Procedure list
        For each local identifier (LID) defined in P
                If LID is a formal parameter then
                    Print "FormalParameter" tag describing LID
                    Compute List of actual arguments (ActualsList) bound to LID
                    Note: See ComputeDataDependencies Algorithm below
                    For each actual argument in ActualsList
                        Print "Variable" tag describing actual argument information
                        Print "Procedure" tag describing procedure callsite information
                    Next actual argument
                End if
        Next local identifier
Next procedure


ComputeDataDependencies (FormalParameter, ActualsList)

If FormalParameter has been visited then Return
Select list of (Formal, Actual) pairs from ParmBindings table where Formal is equal to FormalParameter
For each pair in (Formal, Actual) list
        Add Actual to ActualsList
        If Actual is also a formal parameter then
            ComputeDataDependencies (Actual, ActualsList)
        End if
Next pair
```

C.3. Summary call graph XML schema

```
<?xml version ="1.0"?>
<!--Generated by XML Authority. Conforms to w3c http://www.w3.org/TR/xmlschema-1/-->
<schema targetNamespace = "CallGraph.xsd"
        xmlns = "http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd">
        <element name = "CallGraph">
                <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "ReportTitle"/>
                        <element ref = "ProjectTitle"/>
                        <group order = "seq" minOccurs = "0" maxOccurs = "*">
```

```
                <element ref = "ModuleCollection"/>
                <element ref = "CallGraphNode" minOccurs = "1" maxOccurs = "*"/>
        </group>
        </type>
</element>
<element name = "ReportTitle" type = "string"/>
<element name = "ProjectTitle" type = "string"/>
<element name = "CallGraphNode">
        <type content = "elementOnly">
        <group order = "seq">
        <element ref = "FormalParameters" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "ConstantDeclarations" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "VariableDeclarations" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "CalledModules" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "CallSites" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "GlobalRefs" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "GlobalDefs" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "LocalRefs" minOccurs = "0" maxOccurs = "*"/>
        <element ref = "LocalDefs" minOccurs = "0" maxOccurs = "*"/>
        </group>
        <attribute name = "NodeID" minOccurs = "1" type = "integer"/>
        <attribute name = "ProcName" minOccurs = "1" type = "string"/>
        <attribute name = "ParentModName" minOccurs = "1" type = "string"/>
        </type>
</element>
<element name = "FormalParameters">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "PassByRef" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "PassByVal" minOccurs = "0" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "ConstantDeclarations">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "Constant" minOccurs = "1" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "VariableDeclarations">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "Variable" minOccurs = "1" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "CalledModules">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "Module" minOccurs = "1" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "PassByRef">
        <type content = "elementOnly">
                <group order = "seq" minOccurs = "1" maxOccurs = "*">
                        <element ref = "Parameter"/>
                </group>
        </type>
</element>
<element name = "PassByVal">
        <type content = "elementOnly">
                <group order = "seq" minOccurs = "1" maxOccurs = "*">
                        <element ref = "Parameter"/>
                </group>
        </type>
</element>
<element name = "Constant">
        <type content = "elementOnly">
                <group order = "seq"/>
                <attribute name = "ConstID" minOccurs = "1" type = "string"/>
                <attribute name = "ConstName" minOccurs = "1" type = "string"/>
        </type>
</element>
<element name = "Variable">
```

```
        <type content = "elementOnly">
                <group order = "seq"/>
                <attribute name = "VarID" minOccurs = "1" type = "string"/>
                <attribute name = "VarName" minOccurs = "1" type = "string"/>
        </type>
</element>
<element name = "Module">
        <type content = "elementOnly">
                <group order = "seq"/>
                <attribute name = "NodeID" minOccurs = "1" type = "integer"/>
                <attribute name = "ModName" minOccurs = "1" type = "string"/>
        </type>
</element>
<element name = "ConstName" type = "string"/>
<element name = "ConstID" type = "string"/>
<element name = "VarName" type = "string"/>
<element name = "VarID" type = "string"/>
<element name = "ModuleCollection" type = "string"/>
<element name = "GlobalRefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "ConstRefs" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "VarRefs" minOccurs = "0" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "GlobalDefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "ConstDefs" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "VarDefs" minOccurs = "0" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "LocalRefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "ConstRefs" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "VarRefs" minOccurs = "0" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "LocalDefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "ConstDefs" minOccurs = "0" maxOccurs = "*"/>
                        <element ref = "VarDefs" minOccurs = "0" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "ConstRefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "Constant" minOccurs = "1" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "VarRefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "Variable" minOccurs = "1" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "ConstDefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "Constant" minOccurs = "1" maxOccurs = "*"/>
                </group>
        </type>
</element>
<element name = "VarDefs">
        <type content = "elementOnly">
                <group order = "seq">
                        <element ref = "Variable" minOccurs = "1" maxOccurs = "*"/>
                </group>
```

```
            </type>
        </element>
        <element name = "FirstUse" type = "string"/>
        <element name = "LastUse" type = "string"/>
        <element name = "Parameter">
            <type content = "textOnly">
                <attribute name = "VarID" minOccurs = "1" type = "string"/>
                <attribute name = "VarName" minOccurs = "1" type = "string"/>
                <attribute name = "FirstUse" minOccurs = "1">
                    <datatype source = "string">
                        <enumeration value = "ref"/>
                        <enumeration value = "def"/>
                    </datatype>
                </attribute>
                <attribute name = "LastUse" minOccurs = "1">
                    <datatype source = "string">
                        <enumeration value = "ref"/>
                        <enumeration value = "def"/>
                    </datatype>
                </attribute>
            </type>
        </element>
        <element name = "CallSites">
            <type content = "elementOnly">
                <group order = "seq">
                    <element ref = "CallSite" minOccurs = "1" maxOccurs = "*"/>
                </group>
            </type>
        </element>
        <element name = "CallSite">
            <type content = "elementOnly">
                <group order = "seq">
                    <element ref = "Module"/>
                    <element ref = "StatementLineNumber"/>
                    <element ref = "CallSiteAnalysisCompleted"/>
                    <element ref = "ParameterMapping"/>
                </group>
            </type>
        </element>
        <element name = "StatementLineNumber" type = "string"/>
        <element name = "CallSiteAnalysisCompleted" type = "string"/>
        <element name = "ParameterMapping">
            <type content = "elementOnly">
                <group order = "seq" minOccurs = "1" maxOccurs = "*">
                    <element ref = "ActualParameter"/>
                    <group order = "choice">
                        <element ref = "PassByRef"/>
                        <element ref = "PassByVal"/>
                    </group>
                </group>
            </type>
        </element>
        <element name = "ActualParameter">
            <type content = "textOnly">
                <attribute name = "VarName" minOccurs = "1" type = "string"/>
                <attribute name = "VarID" minOccurs = "1" type = "string"/>
            </type>
        </element>
</schema>
```

C.4. Parameter mapping dependence graph XML schema

```
<?xml version ="1.0"?>
<!--Generated by XML Authority. Conforms to w3c http://www.w3.org/TR/xmlschema-1/-->
<schema targetNamespace = "ParmMappingRpt.xsd"
        xmlns = "http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd">
        <element name = "DataDependenceReport">
            <type content = "mixed">
                <element ref = "ModuleName"/>
            </type>
        </element>
        <element name = "ModuleName">
            <type content = "elementOnly">
                <group order = "seq">
```

```
                        <element ref = "DataDependence"/>
                    </group>
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
                </type>
        </element>
        <element name = "DataDependence">
                <type content = "elementOnly">
                    <group order = "seq">
                    <element ref = "FormalParameter" minOccurs = "0" maxOccurs = "*"/>
                    </group>
                </type>
        </element>
        <element name = "FormalParameter">
                <type content = "elementOnly">
                    <group order = "seq">
                        <element ref = "Variables" minOccurs = "1" maxOccurs = "*"/>
                        <element ref = "Constants" minOccurs = "1" maxOccurs = "*"/>
                    </group>
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
                </type>
        </element>
        <element name = "Variables">
                <type content = "elementOnly">
                    <group order = "seq">
                        <element ref = "Variable"/>
                        <element ref = "Procedure"/>
                    </group>
                </type>
        </element>
        <element name = "Constants">
                <type content = "elementOnly">
                    <group order = "seq">
                        <element ref = "Constant"/>
                        <element ref = "Procedure"/>
                    </group>
                </type>
        </element>
        <element name = "Variable">
                <type content = "textOnly">
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
                </type>
        </element>
        <element name = "Procedure">
                <type content = "textOnly">
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
                </type>
        </element>
        <element name = "Constant">
                <type content = "textOnly">
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
                </type>
        </element>
        <datatype name = "DataDependence" source = "string"/>
</schema>
```

C.5. Reverse ripple dependence graph XML schema

```
<?xml version ="1.0"?>
<!--Generated by XML Authority. Conforms to w3c http://www.w3.org/TR/xmlschema-1/-->
<schema targetNamespace = "ReverseRipple.xsd"
        xmlns = "http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd">
        <element name = "ReverseRipple">
                <type content = "elementOnly">
                    <group order = "seq">
                    <element ref = "CallGraphNode" minOccurs = "1" maxOccurs = "*"/>
                    </group>
                </type>
        </element>
        <element name = "CallGraphNode">
```

```
            <type content = "elementOnly">
                    <group order = "seq">
                    <element ref = "GlobalRefVar" minOccurs = "0" maxOccurs = "*"/>
                    </group>
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
                    <attribute name = "ParentModName" type = "string"/>
            </type>
        </element>
        <element name = "GlobalRefVar">
            <type content = "elementOnly">
                    <group order = "seq">
                    <element ref = "ImpactedBy" minOccurs = "0" maxOccurs = "*"/>
                    </group>
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
            </type>
        </element>
        <element name = "ImpactedBy">
            <type content = "elementOnly">
                    <group order = "seq">
                            <element ref = "Module"/>
                            <element ref = "Variable"/>
                    </group>
            </type>
        </element>
        <element name = "Module">
            <type content = "textOnly">
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
            </type>
        </element>
        <element name = "Variable">
            <type content = "textOnly">
                    <attribute name = "ID" type = "string"/>
                    <attribute name = "Name" type = "string"/>
            </type>
        </element>
</schema>
```

C.6. Call graph metrics XSL view

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<!--
    Call Graph Metrics View
This view collects some element counts for key elements and displays them.  Basically, the
result can be a table, with rows for each moduleCollection/callgraphnode grouped by
modulecollection.  The columns would be ModuleCollection name, callgraphnode (listing the
nodeid and name), # CalledModules, # CallSites, # CallSite with non-empty
ParameterMappings, # GlobalRefs, and# GlobaDefs.  The # of columns are counts, so the cell
would just have a number.  Then display SubTotals for that modulecollection. # of
CallGraphNodes is the # of nodes in that ModuleCollection.

6 Aug 00
-->
<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
        <!-- Title -->
        <H1 style="text-align:center; background-color:gray; color:white">Call Graph
Metrics View</H1>
        <xsl:apply-templates select="/CallGraph/ModuleCollection"/>
        <P/>
        <!-- grand totals -->
        <TABLE border="1">
                <TR>
                        <TH colspan="2">Totals</TH>
                </TR>
                <TR>
                        <TH align="left">Total # of ModuleCollections</TH>
                        <TD>
                                <xsl:value-of select="count(/CallGraph/ModuleCollection)"/>
                        </TD>
```

```
                </TR>
                <TR>
                        <TH align="left">Total # of CallGraphNodes</TH>
                        <TD>
                                <xsl:value-of select="count(/CallGraph/CallGraphNode)"/>
                        </TD>
                </TR>
                <TR>
                        <TH align="left">Total # of CalledModules</TH>
                        <TD>
                                <xsl:value-of
select="count(/CallGraph/CallGraphNode/CalledModules/Module)"/>
                        </TD>
                </TR>
                <TR>
                        <TH align="left">Total # of CallSites</TH>
                        <TD>
                                <xsl:value-of
select="count(/CallGraph/CallGraphNode/CallSites/CallSite)"/>
                        </TD>
                </TR>
                <TR>
                        <TH align="left">Total # of CallSites/PM</TH>
                        <TD>
                                <xsl:if
test="/CallGraph/CallGraphNode/CallSites/CallSite/ParameterMapping">
                                        <xsl:value-of
select="count(/CallGraph/CallGraphNode/CallSites/CallSite)"/>
                                </xsl:if>
                        </TD>
                </TR>
                <TR>
                        <TH align="left">Total # of GlobalRefs</TH>
                        <TD>
                                <xsl:value-of
select="count(/CallGraph/CallGraphNode/GlobalRefs/ConstRefs/Constant)-count(/CallGraph/Cal
lGraphNode/GlobalRefs/VarRefs/Variable)"/>
                        </TD>
                </TR>
                <TR>
                        <TH align="left">Total # of GlobalDefs</TH>
                        <TD>
                                <xsl:value-of
select="count(/CallGraph/CallGraphNode/GlobalDefs/ConstDefs/Constant)-count(/CallGraph/Cal
lGraphNode/GlobalDefs/VarDefs/Variable)"/>
                        </TD>
                </TR>
        </TABLE>
</xsl:template>
<xsl:template match="ModuleCollection">
        <!-- name of this MC -->
        <xsl:variable name="mc_name">
                <xsl:value-of select="."/>
        </xsl:variable>
        <TABLE border="1">
                <TR>
                        <TH>Module Collection</TH>
                        <TH>Call Graph Node</TH>
                        <TH>Number of Called Modules</TH>
                        <TH>Number of Call Sites</TH>
                        <TH>Number of Call Sites per PM</TH>
                        <TH>Number of Global Refs</TH>
                        <TH>Number of Global Defs</TH>
                </TR>
        <xsl:apply-templates select="../CallGraphNode[@ParentModName=$mc_name]">
                <!--<xsl:with-param name="mc_name">
                        <xsl:value-of select="$mc_name"/>
                </xsl:with-param>-->
        </xsl:apply-templates>
                <!-- subtotals -->
                <TR>
                        <TH colspan="2">Module Subtotals</TH>
                        <TD>
                                <xsl:value-of
select="count(../CallGraphNode[@ParentModName=$mc_name]/CalledModules/Module)"/>
```

```
                        </TD>
                        <TD>
                                <xsl:value-of
select="count(../CallGraphNode[:ParentModName=$mc_name]/CallSites/CallSite)"/>
                        </TD>
                        <TD>
                                <xsl:choose>
                                        <xsl:when
test="../CallGraphNode[:ParentModName=$mc_name]/CallSites/CallSite/ParameterMapping">
                                                <xsl:value-of
select="count(../CallGraphNode[:ParentModName=$mc_name]/CallSites/CallSite)"/>
                                        </xsl:when>
                                        <xsl:otherwise>
                                                <xsl:value-of select="'0'"/>
                                        </xsl:otherwise>
                                </xsl:choose>
                        </TD>
                        <TD>
                                <xsl value-of
select="count(../CallGraphNode[:ParentModName=$mc_name]/GlobalRefs/ConstRefs/Constant)+cou
nt(../CallGraphNode[:ParentModName=$mc_name]/GlobalRefs/VarRefs/Variable)"/>
                        </TD>
                        <TD>
                                <xsl:value-of
select="count(../CallGraphNode[:ParentModName=$mc_name]/GlobalDefs/ConstDefs/Constant)+cou
nt(../CallGraphNode[:ParentModName=$mc_name]/GlobalDefs/VarDefs/Variable)"/>
                        </TD>
                </TR>
        </TABLE>
        <P/>
</xsl:template>
<xsl:template match="CallGraphNode">
        <!--<xsl:param name="mc_name"/>
        <xsl:if test="./:ParentModName=$mc_name">-->
                <TR>
                        <!-- Module Collection -->
                        <TD>
                                <xsl:apply-templates select="./:ParentModName"/>
                        </TD>
                        <!-- Call Graph Node -->
                        <TD>
                                NodeID: <xsl:value-of select="./:NodeID"/><BR/>
                                ModuleName: <xsl:value-of select="./:ModuleName"/>
                        </TD>
                        <!-- Number of Called modules -->
                        <TD>
                                <xsl:value-of select="count(./CalledModules/Module)"/>
                        </TD>
                        <!-- Number of Call Sites -->
                        <TD>
                                <xsl:value-of select="count(./CallSites/CallSite)"/>
                        </TD>
                        <!--  # CallSite with non-empty ParameterMappings -->
                        <TD>
                                <xsl:choose>
                                        <xsl:when
test="./CallSites/CallSite/ParameterMapping">
                                                <xsl:value-of
select="count(CallSites/CallSite)"/>
                                        </xsl:when>
                                        <xsl:otherwise>
                                                <xsl:value-of select="'0'"/>
                                        </xsl:otherwise>
                                </xsl:choose>
                        </TD>
                        <!--  Number of Global Refs -->
                        <TD>
                                <xsl:value-of
select="count(./GlobalRefs/ConstRefs/Constant)+count(./GlobalRefs/VarRefs/Variable)"/>
                        </TD>
                        <!--  Number of Global Defs -->
                        <TD>
                                <xsl:value-of
select="count( /GlobalDefs/ConstDefs/Constant)+count(./GlobalDefs/VarDefs/Variable)"/>
                        </TD>
                        <!-- Number of Couplings -->
```

```
            </TR>
<!--    </xsl:if>-->
</xsl:template>
</xsl:stylesheet>
```

C.7. Call graph table XSL view

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="html" indent="no"/>
<xsl:template match="/">
        <HTML>
        <BODY>
                <xsl:variable name="root_name" select="name(*)"/>
                <!-- title bar -->
                <xsl:element name="H1">
                        <xsl:attribute name="style">
                                <xsl:text>text-align:center; background-color:gray;
color:white</xsl:text>
                        </xsl:attribute>
                        <xsl:value-of select="$root_name"/>
                </xsl:element>
                <!-- note to explain color coding -->
                <xsl:element name="P">
                        <xsl:text>Parameter names are in </xsl:text>
                        <xsl:element name="SPAN">
                                <xsl:attribute name="style">
                                        <xsl:text>color:blue</xsl:text>
                                </xsl:attribute>
                                <xsl:text>blue</xsl:text>
                        </xsl:element>
                        <xsl:text> and attribute names are in </xsl:text>
                        <xsl:element name="SPAN">
                                <xsl:attribute name="style">
                                        <xsl:text>color:red</xsl:text>
                                </xsl:attribute>
                                <xsl:text>red</xsl:text>
                        </xsl:element>
                        <xsl:text>.</xsl:text>
                </xsl:element>
                <xsl:apply-templates select="*/*"/>
        </BODY>
        </HTML>
</xsl:template>
<!-- match on any root -->
<xsl:template match="node()">
        <xsl:element name="TABLE">
        <xsl:attribute name="border">1</xsl:attribute>
                <xsl:element name="TR">
                        <xsl:element name="TD">
                                <xsl:element name="SPAN">
                                        <xsl:attribute name="style">
                                                <xsl:text>color:blue</xsl:text>
                                        </xsl:attribute>
                                        <xsl:value-of select="name(.)"/>
                                </xsl:element>
                                <xsl:text>: </xsl:text>
                                <!-- display name and value of all attributes -->
                                <xsl:for-each select="@*">
                                        <xsl:element name="BR"/>
                                        <xsl:element name="SPAN">
                                                <xsl:attribute name="style">
                                                        <xsl:text>color:red</xsl:text>
                                                </xsl:attribute>
                                                <xsl:value-of select="name()"/>
                                        </xsl:element>
                                        <xsl:text>: </xsl:text>
                                        <xsl:value-of select="."/>
                                </xsl:for-each>
                                <!-- process children after listing attributes -->
                                <xsl:apply-templates/>
                        </xsl:element>
                </xsl:element>
```

```
        </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

C.8. Parameter coupling analysis XSL view

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<!--
    This stylesheet searches for all CallSites that have ActualParameters that are paired
with either a PassByRef or PassByVal whose FirstUse is equal to REF.  Once found, selected
data associated with its CallGraphNode, CallSite, Actual Parameter, and Formal Parameter
are output to a table.
-->
<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
        <!-- Title -->
        <H1 style="text-align:center; background-color:gray; color:white">Coupling
Analysis View</H1>
        <TABLE border="1" width="100%">
                <TR>
                        <TH>CallGraphNode</TH>
                        <TH>CallSite</TH>
                        <TH>Actual Parameter</TH>
                        <TH>Formal Parameter</TH>
                </TR>
        <xsl:apply-templates select="/CallGraph/CallGraphNode"/>
        </TABLE>
</xsl:template>
<xsl:template match="CallGraphNode">
        <xsl:apply-templates select="CallSites"/>
</xsl:template>
<xsl:template match="CallSites">
        <xsl:apply-templates select="CallSite"/>
</xsl:template>
<xsl:template match="CallSite">
        <xsl:apply-templates select="ParameterMapping"/>
</xsl:template>
<!-- search for the ActualParameters that meet all Criteria -->
<xsl:template match="ParameterMapping">
        <!-- look at each child element of ParameterMapping -->
        <xsl:for-each select="*">
                <xsl:variable name="parameter_mapping_element_name">
                        <xsl:value-of select="name()"/>
                </xsl:variable>
                <xsl:variable name="parameter_mapping_element_position">
                        <xsl:value-of select="position()"/>
                </xsl:variable>
                <xsl:variable name="position_of_next_element">
                        <xsl:value-of
select="number($parameter_mapping_element_position+1)"/>
                </xsl:variable>
                <xsl:variable name="name_of_next_element">
                        <xsl:value-of
select="name(../*[position()=$position_of_next_element])"/>
                </xsl:variable>
                <!-- test to see if we've found an ActualParameter -->
                <xsl:if test="$parameter_mapping_element_name='ActualParameter'">
                        <!-- when we find one, test to see if the next (sibling) element is
PassByRef or PassByVal -->
                        <xsl:if test="$name_of_next_element='PassByRef'">
                                <!-- if it is, go to child element iff it is a Parameter
element and has an attribute called FirstUse = 'REF' -->
                                <xsl:apply-templates
select="../*[position()=$position_of_next_element]/Parameter[@FirstUse='REF']">
                                        <!-- pass the position of the ActualParameter, so we
can get to its attributes later on -->
                                        <xsl:with-param name="position_of_actual_parameter">
                                                <xsl:value-of
select="$parameter_mapping_element_position"/>
                                        </xsl:with-param>
                                </xsl:apply-templates>
                        </xsl:if>
```

```
                        <xsl:if test="$name_of_next_element='PassByVal'">
                                <!-- if it is, go to child element iff it is a Parameter
element and has an attribute called FirstUse = 'REF' -->
                                <xsl:apply-templates
select="../*[position()=$position_of_next_element]/Parameter[@FirstUse='REF']">
                                        <!-- pass the position of the ActualParameter, so we
can get to its attributes later on -->
                                        <xsl:with-param name="position_of_actual_parameter">
                                                <xsl:value-of
select="$parameter_mapping_element_position"/>
                                        </xsl:with-param>
                                </xsl:apply-templates>
                        </xsl:if>
                </xsl:if>
        </xsl:for-each>
</xsl:template>
<!-- having found the desired ActualParameter, spit out all the data to the table -->
<xsl:template match="Parameter">
        <xsl:param name="position_of_actual_parameter"/>
        <TR>
                <!-- CallGraphNode -->
                <TD>
                        NodeID="<xsl:value-of select="../../../../../@NodeID"/>"<BR/>
                        ModuleName="<xsl:value-of
select="../../../../../@ModuleName"/>"<BR/>
                        ParentModName="<xsl:value-of
select="../../../../../@ParentModName"/>"
                </TD>
                <!-- CallSite -->
                <TD>
                        ModuleName="<xsl:value-of
select="../../../Module/@ModuleName"/>"<BR/>
                        ModuleID="<xsl:value-of select="../../../Module/@ModuleID"/>"<BR/>
                        InModuleCollection="<xsl:value-of
select="../../../Module/@InModuleCollection"/>"
                </TD>
                <!-- Actual Parameter -->
                <TD>
                        VarName="<xsl:value-of
select="../../*[position()=$position_of_actual_parameter]/@VarName"/>"<BR/>
                        VarID="<xsl:value-of
select="../../*[position()=$position_of_actual_parameter]/@VarID"/>"
                </TD>
                <!-- Formal Parameter -->
                <TD>
                        <xsl:value-of select="name(..)"/><BR/>
                        VarName="<xsl:value-of select="@VarName"/>"<BR/>
                        VarID="<xsl:value-of select="@VarID"/>"<BR/>
                        FirstUse="<xsl:value-of select="@FirstUse"/>"<BR/>
                        LastUse="<xsl:value-of select="@LastUse"/>"
                </TD>
        </TR>
</xsl:template>
</xsl:stylesheet>
```

C.9. Call coupling analysis XSL view

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<!--
    This stylesheet searches for all CallSites that have ActualParameters that are paired
with either a PassByRef or PassByVal whose FirstUse is equal to REF.  Once found, selected
data associated with its CallGraphNode, CallSite, Actual Parameter, and Formal Parameter
are output to a table.
-->
<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
        <!-- Title -->
        <H1 style="text-align:center; background-color:gray; color:white">Call Coupling
Analysis View</H1>
        <TABLE border="1" width="100%">
                <TR>
                        <TH>CallGraphNode</TH>
```

```
                        <TH>CallSite</TH>
                </TR>
        <xsl:apply-templates select="/CallGraph/CallGraphNode"/>
        </TABLE>
        <P/>
        <!-- grand totals -->
        <TABLE border="1">
                <TR>
                        <TH colspan="2">Totals</TH>
                </TR>
                <TR>
                        <TH align="left">Total # of Call Couplings</TH>
                        <TD>
                                <xsl:value-of
select="count(/CallGraph/CallGraphNode/CallSites/CallSite)"/>
                        </TD>
                </TR>
        </TABLE>
</xsl:template>
<xsl:template match="CallGraphNode">
        <xsl:apply-templates select="CallSites"/>
</xsl:template>
<xsl:template match="CallSites">
        <xsl:apply-templates select="CallSite"/>
</xsl:template>
<xsl:template match="CallSite">
        <TR>
                <!-- CallGraphNode -->
                <TD>
                        <!--  NodeID="<xsl:value-of select="../../../../../:NodeID"/>"<BR/>
                        ModuleName="<xsl:value-of
select="../../../../../:ModuleName"/>"<BR/>
                        ParentModName="<xsl:value-of
select="../../../../../:ParentModName"/>"  -->
                        NodeID="<xsl:value-of select="../../:NodeID"/>"<BR/>
                        ModuleName="<xsl:value-of select="../../:ModuleName"/>"<BR/>
                        ParentModName="<xsl:value-of select="../../:ParentModName"/>"
                </TD>
                <!-- CallSite -->
                <TD>
                        <!--  ModuleName="<xsl:value-of
select="../../../Module/:ModuleName"/>"<BR/>
                        ModuleID="<xsl:value-of select="../../../Module/:ModuleID"/>"<BR/>
                        InModuleCollection="<xsl:value-of
select="../../../Module/:InModuleCollection"/>"  -->
                        ModuleName="<xsl:value-of select="./Module/:ModuleName"/>"<BR/>
                        ModuleID="<xsl:value-of select="./Module/:ModuleID"/>"<BR/>
                        InModuleCollection="<xsl:value-of
select="./Module/:InModuleCollection"/>"
                </TD>
        </TR>
</xsl:template>
</xsl:stylesheet>
```

C.10. Parameter mapping dependence XSL view

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<!--
        3 Aug 00
-->
<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
        <!-- Title -->
<H1 style="text-align:center; background-color:gray; color:white">Parameter Mapping
Dependence View</H1>
        <TABLE border="1" width="100%">
                <TR>
                        <TH>Module Name</TH>
                        <TH>Formal Parameter</TH>
                        <TH>Dependence</TH>
                </TR>
        <xsl:apply-templates select="/DataDependenceReport/ModuleName"/>
```

```
        </TABLE>
        <P/>
        <!-- sub totals -->
        <H1 style="text-align:center; background-color:gray; color:white">Metrics</H1>
        <TABLE border="1">
                <TR>
                        <TH>Module Name</TH>
                        <TH>Formal Parameter</TH>
                        <TH>Subtotal per Formal Parameter:</TH>
                </TR>
                <xsl:for-each select="/DataDependenceReport/ModuleName">
                        <xsl:if test="count(./DataDependence/FormalParameter)=0">
                                <TR>
                                <!-- CallGraphNode -->
                                <TD>
                                        ModuleName="<xsl:value-of select="./@Name"/>"<BR/>
                                                ID="<xsl:value-of select="./@ID"/>"<BR/>
                                </TD>
                                <!-- Formal Parameter -->
                                <TD>
                                ---
                                </TD>
                                <TD>
                                0
                                </TD>
                                </TR>
                        </xsl:if>
                        <xsl:for-each select="./DataDependence/FormalParameter">
                        <TR>
                        <!-- CallGraphNode -->
                        <TD>
                                ModuleName="<xsl:value-of select="../../@Name"/>"<BR/>
                                ID="<xsl:value-of select="../../@ID"/>"<BR/>
                        </TD>
                        <!-- Formal Parameter -->
                        <TD>
                                Name="<xsl:value-of select="./@Name"/>"<BR/>
                                ID="<xsl:value-of select="./@ID"/>"
                        </TD>
                        <TD>
                                <xsl:value-of
select="count(./Variables)+count(./Constants)"/>
                        </TD>
                        </TR>
                        </xsl:for-each>
                        <TR>
                                <TH align="left">Subtotal per CallGraphNode: </TH>
                                <TD>
                                        ---
                                </TD>
                                <TD>
                                        <xsl:value-of
select="count( /DataDependence/FormalParameter/Variables)+count(./DataDependence/FormalPar
ameter/Constants)"/>
                                </TD>
                        </TR>
                </xsl:for-each>
        </TABLE>
        <P/>
        <!-- grand totals -->
        <TABLE border="1">
                <TR>
                        <TH colspan="2">Totals</TH>
                </TR>
                <TR>
                        <TH align="left">Total # of Parameter Mapping Dependencies in Call
Graph:</TH>
                        <TD>
                                <xsl:value-of
select="count(/DataDependenceReport/ModuleName/DataDependence/FormalParameter/Variables)+c
ount(/DataDependenceReport/ModuleName/DataDependence/FormalParameter/Constants)"/>
                        </TD>
                </TR>
        </TABLE>
</xsl:template>
<xsl:template match="ModuleName">
```

```
        <xsl:apply-templates select="DataDependence"/>
</xsl:template>
<xsl:template match="DataDependence">
        <xsl:apply-templates select="FormalParameter"/>
</xsl:template>
<xsl:template match="FormalParameter">
        <xsl:apply-templates select="Variables"/>
        <xsl:apply-templates select="Constants"/>
</xsl:template>
<xsl:template match="Variables">
        <TR>
                <!-- Module -->
                <TD>
                        Name="<xsl:value-of select="../../../@Name"/>"<BR/>
                        ID="<xsl:value-of select="../../../@ID"/>"<BR/>
                </TD>
                <!-- Formal Parameter -->
                <TD>
                        Name="<xsl:value-of select="../@Name"/>"<BR/>
                        ID="<xsl:value-of select="../@ID"/>"
                </TD>
                <TD>
                        Variable Name="<xsl:value-of select="./Variable/@Name"/>"<BR/>
                        Variable ID="<xsl:value-of select="./Variable/@ID"/>"
                        Procedure Name="<xsl:value-of select="./Procedure/@Name"/>"<BR/>
                        Procedure ID="<xsl:value-of select="./Procedure/@ID"/>"<BR/>
                </TD>
        </TR>
</xsl:template>
<xsl:template match="Constants">
        <TR>
                <!-- Module -->
                <TD>
                        Name="<xsl:value-of select="../../../@Name"/>"<BR/>
                        ID="<xsl:value-of select="../../../@ID"/>"<BR/>
                </TD>
                <!-- Formal Parameter -->
                <TD>
                        Name="<xsl:value-of select="../@Name"/>"<BR/>
                        ID="<xsl:value-of select="../@ID"/>"
                </TD>
                <TD>
                        Constant Name="<xsl:value-of select="./Constant/@Name"/>"<BR/>
                        Constant ID="<xsl:value-of select="./Constant/@ID"/>"
                        Procedure Name="<xsl:value-of select="./Procedure/@Name"/>"<BR/>
                        Procedure ID="<xsl:value-of select="./Procedure/@ID"/>"<BR/>
                </TD>
        </TR>
</xsl:template>
</xsl:stylesheet>
```

C.11. Reverse ripple analysis XSL view

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<!--
        3 Aug 00
-->
<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
        <!-- Title -->
        <H1 style="text-align:center; background-color:gray; color:white">Reverse Ripple
Analysis View</H1>
        <TABLE border="1" width="100%">
                <TR>
                        <TH>CallGraphNode</TH>
                        <TH>GlobalRefVar</TH>
                        <TH>Impacted By</TH>
                </TR>
        <xsl:apply-templates select="/ReverseRipple/CallGraphNode"/>
        </TABLE>
        <P/>
        <!-- sub totals -->
```

```
        <H1 style="text-align:center; background-color:gray; color:white">Metrics</H1>
        <TABLE border="1">
                <TR>
                        <TH>CallGraphNode</TH>
                        <TH>GlobalRefVar</TH>
                        <TH>Subtotal per GlobalRefVar:</TH>
                </TR>
                <xsl:for-each select="/ReverseRipple/CallGraphNode">
                        <xsl:for-each select="./GlobalRefVar">
                                <TR>
                                        <!-- CallGraphNode -->
                                        <TD>
                                                NodeID="<xsl:value-of select="../@ID"/>"<BR/>
                                                ModuleName="<xsl:value-of
select="../@Name"/>"<BR/>
                                        </TD>
                                        <!-- Formal Parameter -->
                                        <TD>
                                                Name="<xsl:value-of select="./@Name"/>"<BR/>
                                                ID="<xsl:value-of select="./@ID"/>"
                                        </TD>
                                        <TD>
                                                <xsl:value-of select="count(./ImpactedBy)"/>
                                        </TD>
                                </TR>
                        </xsl:for-each>
                        <TR>
                                <TH align="left">Subtotal per CallGraphNode: </TH>
                                <TD>
                                        ---
                                </TD>
                                <TD>
                                        <xsl:value-of
select="count( /GlobalRefVar/ImpactedBy)"/>
                                </TD>
                        </TR>
                </xsl:for-each>
        </TABLE>
        <P/>
        <!-- grand totals -->
        <TABLE border="1">
                <TR>
                        <TH colspan="2">Totals</TH>
                </TR>
                <TR>
                        <TH align="left">Total # of Reverse Ripple Impacts in Call
Graph:</TH>
                        <TD>
                                <xsl:value-of
select="count(/ReverseRipple/CallGraphNode/GlobalRefVar/ImpactedBy)"/>
                        </TD>
                </TR>
        </TABLE>
</xsl:template>
<xsl:template match="CallGraphNode">
        <xsl:apply-templates select="GlobalRefVar"/>
</xsl:template>
<xsl:template match="GlobalRefVar">
        <xsl:apply-templates select="ImpactedBy"/>
</xsl:template>
<xsl:template match="ImpactedBy">
        <TR>
                <!-- CallGraphNode -->
                <TD>
                        <!--  NodeID="<xsl:value-of select="../../../../../@NodeID"/>"<BR/>
                        ModuleName="<xsl:value-of
select="../../../../../@ModuleName"/>"<BR/>
                        ParentModName="<xsl:value-of
select="../../../../../@ParentModName"/>"  -->
                        NodeID="<xsl:value-of select="../../@ID"/>"<BR/>
                        ModuleName="<xsl:value-of select="../../@Name"/>"<BR/>
                        ParentModName="<xsl:value-of select="../../@ParentModName"/>"
                </TD>
                <!-- Formal Parameter -->
                <TD>
```

```
                        <!--  ModuleName="<xsl:value-of
select="  /../../Module/@ModuleName"/>"<BR/>
                        ModuleID="<xsl:value-of select="../../../Module/@ModuleID"/>"<BR/>
                        InModuleCollection="<xsl:value-of
select="../../../Module/@InModuleCollection"/>"  -->
                        Name="<xsl:value-of select="../@Name"/>"<BR/>
                        ID="<xsl:value-of select="../@ID"/>"
                </TD>
                <TD>
                        <!--  ModuleName="<xsl:value-of
select="../../../Module/@ModuleName"/>"<BR/>
                        ModuleID="<xsl:value-of select="../../../Module/@ModuleID"/>"<BR/>
                        InModuleCollection="<xsl:value-of
select="../../../Module/@InModuleCollection"/>"  -->
                        ModuleName="<xsl:value-of select="./Module/@Name"/>"<BR/>
                        ModuleID="<xsl:value-of select="./Module/@ID"/>"<BR/>
                        VariableName="<xsl:value-of select="./Variable/@Name"/>"<BR/>
                        VariableID="<xsl:value-of select="./Variable/@ID"/>"
                </TD>
        </TR>
</xsl:template>
</xsl:stylesheet>
```

VITA

ROBERT D. CHERINKA
111 Kicotan Turn
Yorktown, VA 23693
(757) 867-7348
rcherinka@home.com

Robert D. Cherinka is a Lead Information Systems Engineer for the MITRE Corporation, located in Hampton, Virginia. His expertise is in software and process engineering technology. Prior to joining MITRE in 1993, Mr. Cherinka was an Officer performing software engineering duties in the United States Air Force at the Air Combat Command Computer Support Squadron, Langley Air Force Base, Hampton, Virginia. Prior to that, he worked as a software engineer for Computer Techniques in Clarks Summit, Pennsylvania and the Department of Surgery at the University of Pittsburgh, Pennsylvania.

Mr. Cherinka is currently working on a Ph.D. in computer science from Old Dominion University, Norfolk, Virginia, with an expected date of graduation of December 2000. He earned a M.S. in computer science in 1991 from Old Dominion University. He earned a B.S. in computer science in 1987 from the University of Pittsburgh. He has published numerous papers and conference presentations in the area of software and process engineering. He is a member of the IEEE, IEEE Computer Society, and ACM.

RECENT PUBLICATIONS

R. Cherinka, C. Wild, J. Ricci and C.M. Overstreet, *Issues and Approaches in Testing Mission-Critical COTS Systems*, 2000 International Conference on Software Testing (TCS2000), June 2000;

R. Cherinka, C. Wild, J. Ricci and C.M. Overstreet, *Testing and Validating Mission-Critical COTS Systems*, 2000 European Conference on Object-oriented Programming: International Workshop on Component-based Programming (WCOP '00), June 2000;

R. Cherinka, *Maintaining a COTS Integrated Solution – Are traditional static analysis techniques sufficient for this new programming methodology?*, 1998 IEEE International Conference on Software Maintenance (ICSM98), October 1998;

R. Cherinka, *Maintaining a COTS Component-based Solution - Can Traditional Static Analysis Techniques be useful for this new programming methodology?*, 1998 European Conference on Object-oriented Programming: International Workshop on Component-based Programming (WCOP '98);