1999

A Transformational Reengineering System That Supports Software Maintenance Using a Graph Representation for the Identification of an Object-Oriented Software Architecture.

Ramachenga Reddy Valasareddi
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A TRANSFORMATIONAL REENGINEERING SYSTEM THAT SUPPORTS SOFTWARE MAINTENANCE USING A GRAPH REPRESENTATION FOR THE IDENTIFICATION OF AN OBJECT-ORIENTED SOFTWARE ARCHITECTURE

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Ramachenga Reddy Valasareddi
B.Tech., Sri Venkateswara University, India, 1986
M.Tech., Hyderabad Central University, India, 1988
M.S., University of Southwestern Louisiana, 1991
May 1999

To
Saumya
Leelavathi
Siddarami Reddy

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Doris Carver, for her support and guidance during my stay at LSU. I particularly appreciate her patience. I also thank her for providing the facilities of the Software Engineering Laboratory.

I owe special thanks to Dr. Iyengar for his kind words and assistance during early stages of my graduate studies. I would like to thank all the members of my doctoral committee, Dr. Sitarama Iyengar, Dr. Donald Kraft, Dr. Debashish Gosh, and Dr. David Mukai for taking the time to review my dissertation and for their suggestions.

I would like to thank Dr. Lalit Verma of Agriculture Engineering and Dr. Kevin Robbins of Southern Regional Climatic Center for the financial assistance. Special thanks to Joyce Robertson of River City medical, Inc. for the job and for allowing me to pursue my studies.

Thanks to Samuel Harries, Tim Ottinger, and Michael Siff for their inputs to my research.

I appreciate the support and encouragement of the Indian families in Baton Rouge, especially those from my home state Andhra Pradesh. Because of them I moved to Baton Rouge and was subsequently introduced to LSU. They truly made my stay here worthwhile.

Special thanks to Rama, Subba Rao, Venkat, Siva Ram, Ravi, Prasad, Raju, Niranjan, and Carlos for being good friends. I thank those many others not mentioned here who provided encouragement and assistance during my graduate studies.

I thank my parents and my brothers for their guidance and support. Without their encouragement I couldn't have made it through the program.

I am especially thankful to my daughter Saumya and wife Lalasa for their patient cooperation and unconditional support.

Finally, I am thankful to God for granting me the skills and opportunities that made this possible.

Table of Contents

List of Tables

List of Figures

Abstract

The process of maintenance and enhancement of legacy software systems is a laborious and unavoidable task. Often these systems lack structure or modularity, as they were developed using programming languages and paradigms that do not incorporate object-oriented features and sound design principles. The software engineer's task can be simplified if tools are available to identify object like features in the code. These tools can help transform the non-object-oriented code to object oriented code. This research describes a comprehensive and systematic process for transformational reengineering of legacy systems.

Research in reengineering is mainly focused on clustering techniques that group procedures present in the legacy system into candidate objects. These clustering approaches are limited to systems with well-defined data structures and procedures. Several of these approaches are either not comprehensive, limited to certain types of systems, or depend extensively on engineer knowledge of the system. Unlike these approaches that analyze legacy systems at the procedural level, the reengineering process we present analyzes systems at the statement level.

This process results in the identification of object operations. These operations, along with the state variables and the user defined data structures, are arranged in a hierarchy that represents the object structure of the reengineered variant of the legacy system. From this system hierarchy, objects are identified and encapsulated by streamlining the interfaces. The reengineering process is incorporated in a tool, ReArchitect.

Programs are statically analyzed and represented as a statement dependence graph (StDG) for further processing. The StDG is a fine-grained representation with modular representation for functions and program slices. It can adapt to program changes, unlike other representations. The StDG is restructured by merging cohesive components in the graph. The restructured graph is used to build the object structure, which is used to identify the objects.

The StDG is a theoretically sound framework that provides support for many problems found in the reengineering domain. We show the value of the StDG in two such domains: program slicing and maintenance. The StDG is restructured differently for different requirements (space/time), and for different types of applications.

Chapter 1

Introduction

1.1 Introduction

Industrial-strength software is complex. This complexity makes it difficult to comprehend all the subtleties of its design. Regardless, complexity is an essential property of the software which we need to master. Causes for this complexity include: complexity of the problem domain, difficulty of managing the development process, the flexibility possible through the software, and the evolutionary nature of the software [Booc 94].

A software system should evolve if it is to be useful over time. Continual satisfaction demands continuing change. During the life of a system, its environment changes, user needs change, and developing concepts and technologies advance. Systems failing to adopt to these changes become increasingly less useful in the new environment [Lehm 85]. Evolution of software is also known as software maintenance.

The understanding and adaptation of systems to advanced technologies is the topic of this research. Reengineering is the process of examination, understanding, and altering a system with the intent to implement it in a new form to make it more maintainable. In this research, we present a methodology for reengineering procedural systems to object-oriented systems. The remainder of the chapter presents an overview of the maintenance problem, the objectives of this research, and finally a description of the organization of this dissertation.

1.2 Background

1.2.1 Maintenance

The life span of a system consists of specification, design, implementation and maintenance phases. The maintenance phase requires the greatest effort and resources. On an average, maintenance costs constitute 70-90% of the total system costs [Thom 84]. The American National Standards Institute defines *software maintenance* as the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adopt the product to changed environments" [Chik 90].

One reason for high maintenance costs is that the structure of the systems which have to be modified may not be obvious to the maintainer. As systems evolve, their structure degrades and the complexity of the structure increases, unless specific complexity control effort is applied [Lehm 85]. Complexity control actions are rarely carried out in the real world due to lack of time and/or cost considerations. Further, system structure is also corrupted when changes are made without regard to the overall architecture of the system. As changes are made, these changes introduce new system faults which then require more changes to correct them. Gradually the system approaches a limit where it is no longer cost-efficient or even technically motivating to continue the maintenance.

1.2.2 Program understanding

A system and its components need to be understood before changes can be made. Program understanding is the task of extracting information about a program's

behavior (what the program does and how it does it) from the source code. The understanding process occupies 50-90% of maintenance cost [Thom 84]. Program understanding is a conceptually complex process of making sense of a complex system. One reason for this complexity is lack of system documentation. Often changes are made only to the source code of the system because of time and high costs involved in updating the documentation, resulting in inconsistencies between the source code and its documentation. Due to inaccurate and/or incomplete documentation, the source code becomes the only dependable source of information for maintenance.

Maintenance costs are also governed by the approaches adopted for system development. Systems developed using information hiding principles are better suited for maintenance [Booc 94]. If support from the software development methodology, documentation, and the structure of the system are absent, maintenance becomes a technically challenging process.

1.2.3 Current status of legacy software systems

Many systems in use today have evolved for several years, with modification after modification layered upon the original implementation by several generations of programmers. These systems were developed using a procedural paradigm, which uses algorithmic decomposition methods and imperative languages such as C, for implementation.

Maintenance of software developed with a procedural paradigm presents many problems. In procedural programs, it is difficult to define well-delimited components and their relationships because often they are not visible. Algorithmic decomposition highlights the ordering of events, and each module in the system denotes a major step in

the overall process. Further, imperative languages do not generally provide facilities for expressing system structure information.

Additionally, many of the modifications that have been made were not anticipated in the original design of these systems, resulting in global modifications to incorporate the changes. A global change distributes design information over the entire system, corrupting the existing system structure. It is advantageous to hide these modifications in one module and still achieve the same functionality. Furthermore, the development documentation of these systems is often out of date or nonexistent. The result is that understanding and maintaining these systems is difficult and time consuming.

1.2.4 Object-oriented paradigm

The object-oriented paradigm has entered the mainstream of computing and has matured over the past decade [Booc 94]. This paradigm has proven to be advantageous over the procedural paradigm in all phases of software development, from analysis and design to implementation and maintenance. The expressive power of object-based and object-oriented programming languages is one reason for their rise in popularity.

An object-oriented system is decomposed according to the key abstractions in the problem domain. It emphasizes the agents that either cause action or are the subjects upon which these operations act. It directly addresses the inherent complexity of the software by facilitating intelligent decisions regarding the separation of concerns and information hiding. Object-oriented decomposition has a number of advantages over algorithmic decomposition. It yields smaller systems through the reuse of common

mechanisms [Booc 94]. Also, these systems are more resilient to change and thus evolve more easily over time.

The ability to comprehend different kinds of information at once is limited. The object-oriented concepts of abstraction, encapsulation, modularity and hierarchy, help overcome this limitation. An *abstraction* denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer [Booc 94]. *Encapsulation* is the technique of combining data and operations needed to process the data under one entity. *Modularity* is the property of a system that has been decomposed into a set of cohesive, loosely coupled modules. *Hierarchy* is the ordering of abstractions.

1.2.5 Reengineering

Software developers, both new system developers and system maintainers, are slowly moving to the object-oriented paradigm. Maintainers of procedural systems face many problems in the process of moving the systems. Technological advances have provided little help with the process.

When a procedural system needs to be migrated to an object-oriented architecture, two basic options are available. One option is to discard the old system and build a new one. The other option is to encapsulate or wrap the old system and then communicate with it via a standard application program interface. The former option is unlikely to happen due to cost considerations. In some cases, it is impractical. The latter option results in a system that will continue to have all the drawbacks of the old system. Current research is focusing on a third option, reengineering.

The field of reengineering is growing rapidly in response to the need to support legacy system maintenance. Reengineering of procedural systems to object-oriented systems requires a radical restructuring of the systems. So far, this restructuring has been an impractical, highly risky, and costly undertaking because of the lack of a reliable and automated translation process [Wood 98].

We use the following reengineering model proposed by [Jaco 91] for this work:

Reengineering = Reverse engineering + Δ + Forward Engineering

Reverse engineering is the process of analyzing a subject system to understand and represent the system at a higher level of abstraction. The "Δ" represents changes to be made to the system. Change can be in the functionality, in the implementation technique, or in the design of the system. Forward engineering is the re-implementation of the system.

1.3 Overview of the Research

1.3.1 Objective

The goal of this research is to define a reengineering methodology for transforming procedural systems into an object-oriented architecture. In particular, this research extracts a procedural design from an existing system (reverse engineering), makes changes to the design (Δ), and identifies and extracts objects in the procedural code for use in implementing the system with the object-oriented paradigm (forward engineering). The methodology is supported by a tool to assist the reverse engineering process. The tool is called ReArchitect.

Reengineering reflects a design decision that is inherently subjective. Hence, it is unlikely that the reengineering can ever be fully automated. The assistance of the software engineer is crucial to any tool [Wood 98]. The tool should be able to help the engineer understand the program and the process, and also be able to incorporate the knowledge of the engineer into the process. In particular, the tool should be capable of program slicing (for understanding) and restructuring (for understanding program and improving the quality of results obtained), in addition to reengineering.

1.3.2 Reengineering process

We propose the reengineering methodology presented in figure 1.1. The reengineering process along with the roles of the ReArchitect and the software engineer are indicated in the figure. Four important steps are involved in the reengineering process.

i. Reverse engineering. The program code is analyzed statically. The design of the program is extracted and represented in the form of a graph known as a Statement Dependence Graph (StDG).

ii. Design change. The StDG is restructured, which results in identification of cohesive components in the program. The restructured graph is known as a Restructured StDG (VRG).

iii. Design optimization. The VRG is optimized for object identification.

iv. Object identification. A potential objects chart known as a State Reference graph (SRG) is built from the VRG and call graph. The code and the SRG are used for object identification and extraction from the code.

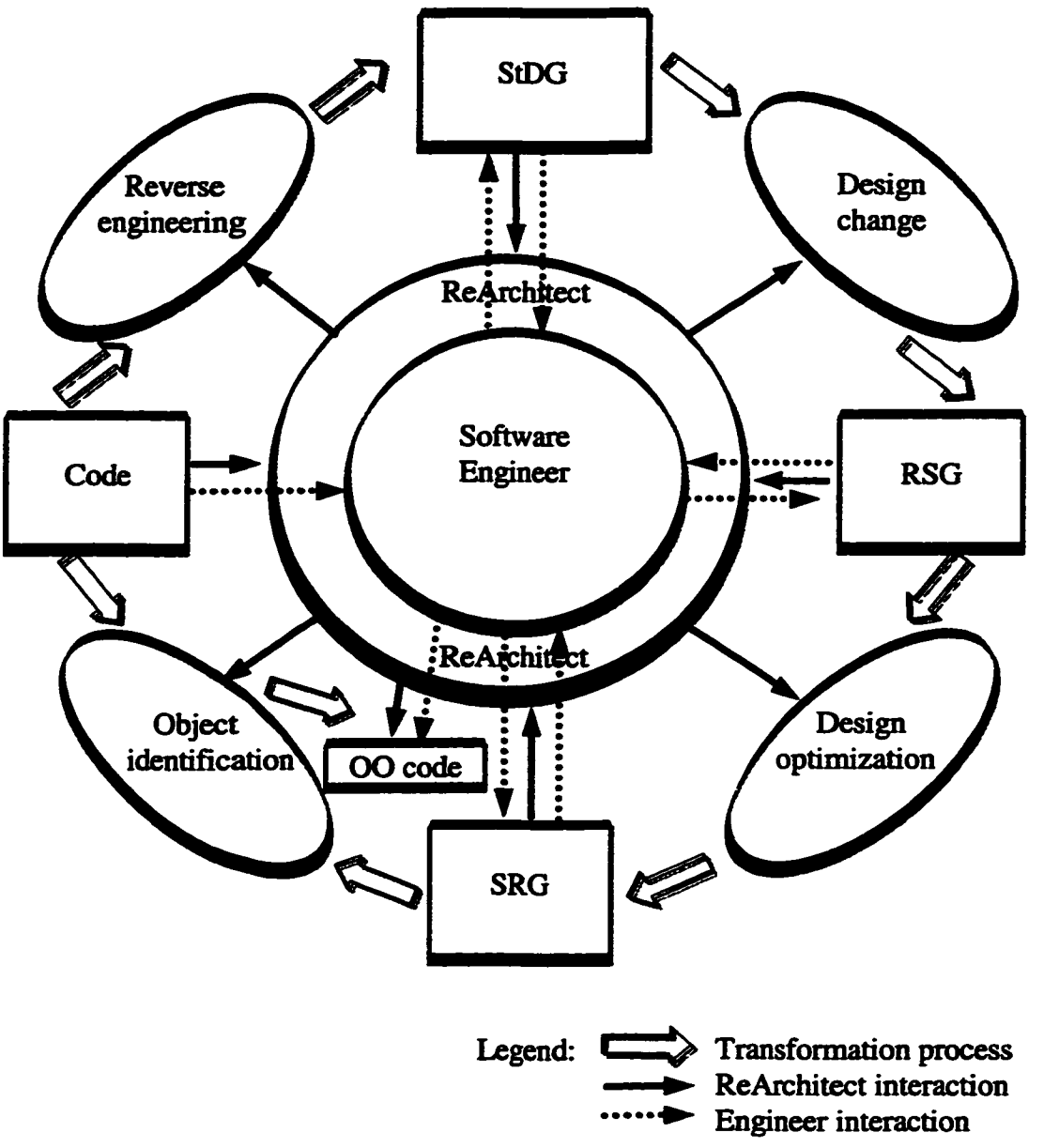


Figure 1.1. Reengineering process

Role of the software engineer. The software engineer develops new software specific knowledge with the help of the ReArchitect. This knowledge is used in the reengineering process for further processing of the program and for obtaining optimal results.

Role of the ReArchitect. ReArchitect helps in extraction of information, model creation, visualization and slicing for program understanding, and program restructuring transformations. It incorporates the input of the software engineer in the process. Each step in the process is automatic. The results of each step are presented to the software engineer for understanding and for feedback.

1.3.3 Implementation considerations

A prototype of the ReArchitect is implemented in Java. It has three key phases:

i. Extraction. Extraction and representation of the program dependences in the form of Statement Dependence Graph (StDG).

ii. Transformation. Transformation of the StDG to VRG.

iii. Application. Use of the VRG for finding slices of the program and for applying modifications to the program (maintenance activities).

1.4 Outline of the Dissertation

The dissertation is organized as follows:

Chapter 1 discusses the maintenance problem and the objectives of this research. The new reengineering process model is briefly introduced.

Chapter 2 presents the motivation for the research and reviews reengineering approaches available in the literature.

Chapter 3 discusses the issues in program analysis and representation, along with approaches available in the literature. The chapter summarizes the requirements for a program representation, and describes a new program representation, known as the

Statement Dependence Graph (StDG). A representation of alias information and construction of the StDG are also presented in the chapter.

Chapter 4 describes techniques to restructure the StDG. The restructured StDG is known as the Restructured StDG (VRG). The VRG can be used for program slicing, program maintenance, and for reengineering.

Chapter 5 presents the application of the VRG to program slicing and program maintenance. Program slices are used to understand programs. A new slicing technique, known as modular slicing, is introduced in the chapter. Program restructuring is used for understanding and for improving the quality of reengineering processing. The chapter also describes how the VRG can be used for program maintenance.

Chapter 6 describes the reverse engineering process. The language independent format (LIF) is an intermediate representation for programs. The chapter describes a process for the translation of LIF to the StDG. The StDG and other design views of the program are represented graphically.

Chapter 7 discusses various approaches available for object identification and describes a new approach using the VRG. It also presents how the new design of the program is represented.

Chapter 8 describes a forward engineering process that involves identification and extraction of objects. Examples designed to demonstrate the process and a case study are also included.

Chapter 9 presents the conclusions and contributions of the dissertation. Implementation details of the ReArchitect are also described in the chapter.

Chapter 2

Motivation and Related Research

2.1 Motivation for the Research

Roughly 75 billion dollars were spent on maintenance in 1990 alone [Wood 98]. Reliable help to partially automate the maintenance activities or to reduce the complexity of the underlying software system could have a staggering economic impact. Reengineering is the ultimate step in the maintenance process. Moreover, it is the only recourse available for migrating systems to different architectures or environments. Reengineering software is not normally effective unless some automated tool support can be deployed to support the process [Wood 98][Some 96].

Unfortunately software engineers have little or no tool support available to them [Wood 98]. Existing tools are limited to automatically extracting and analyzing program structure. They extract features such as relationships between functions or relationships between functions and variables. Unfortunately, structural information alone is not sufficient for performing meaningful operations on programs [Some 96]. The "semantic" or "conceptual" understanding required can only be provided through human assistance. Human interactions are crucial because of the technical complexities and the lack of validated methodologies for reengineering. Moreover, reengineering from a procedural paradigm to an object-oriented paradigm reflects a design decision that is inherently subjective, and thus human interaction is unavoidable [Siff 99]. For instance, static analysis of programs is one area where human interaction is known to be beneficial.

Static analysis is the process of extracting semantic information about a program at compile time. Static analysis of programs is hard - hence several simplifying assumptions are made [Land 92]. The results thus obtained are approximate, therefore, input from the software engineer can improve the analysis. The input is also crucial in other stages of the reengineering process. As a result, complete automation of the reengineering process is not feasible [Gall 95][Snee 98]. A reengineering tool should be capable of not only helping the software engineer but also incorporating the engineer's knowledge into the reengineering process.

For an engineer to provide help, understanding of the program is crucial. One way to help the human better understand the code is by simplifying the program structure. Ongoing maintenance of software systems tends to destroy program structure, complicating efforts to gain a deeper understanding of the code [Some 96]. When code is restructured, it is often easier to understand. Program slicing is known to help in understanding code [Weis 84][Gall 91].

Other reasons for reengineering procedural systems to object-oriented architectures are:

- Object-oriented development methods can be used to gradually modernize an old system [Newc 95].
- There is a need to integrate object-oriented programming into an existing system that is not implemented using modern programming techniques. A technique such as information hiding, for example, can help reduce the complexity of the system.

- Object-oriented programming in general, and inheritance in particular makes it possible to define and use modules that are functionally incomplete. It then allows extension of the modules without upsetting the operation of other modules or their clients. This feature makes the system more flexible, more easily extensible, and simpler to maintain.
- Finding objects within procedural programs has become a promising approach to reduce the effort in program understanding and maintenance cost. Object identification constitutes a basis for a complete re-architecting of the system from the conventional procedural program to an object-oriented program.
- Identification of object like features in a program, including the support for testing and debugging, helps avoid the degradation of the original design during maintenance and facilitates reuse [Liva 92].
- Even though it is generally unrealistic to replace an old system by a completely new system, there is a need for methods to gradually replace older system parts.

2.2 Related Research

The reverse engineering of a program involves program understanding and design abstraction. Program understanding is a process of relating program artifacts to the conceptual model of the human observer. [Bigg 94] discusses issues that relate program artifacts to domain knowledge. A generic reverse engineering tool for program understanding is described in [Jarz 95]. [Chen 90][Plat 91] describe a means for C program representation and a toolkit for extracting different views of the program. A process of automatically exacting design knowledge from the source code for program

understanding is presented in [Quil 94][Snee 95]. [Snee 95] proposes seven types of objects such as user interface objects, information objects, and FILE objects. Programs are searched for these specific objects and are used in new object-oriented software development.

One purpose of reengineering is to extract system parts for reuse. Reuse may be at the level of artifacts, sub-system, or the entire system. Program artifacts are in the form of components, abstract data types (ADT), or objects. Various researchers use terms such as functions [Lanu 93], segments [Ning 94], slices [Weis 84], plans [John 85], and concepts [Quil 94] to refer to program components.

A practical approach to components recovery is described in [Ning 94][Lanu 93]. A reengineering process for extracting ADTs is presented in [Conf 93]. Automatic extraction of objects from existing code may result in spurious or misleading groupings. To prevent the undesirable groupings, [Gall 95] discusses a mapping process from extracted objects to domain knowledge. An interactive approach to recovering ADTs and object instances is given in [Yeh 95]. Strategies for gradually transforming a system composed of procedural programs to object-oriented system are described in [Newc 95][Jaco 91]. Reengineering of legacy systems using Fusion/RE and Draco-puc transformation system is presented in [Pent 98]. Reengineering techniques have significant market value for vendors, thus much of the literature provides more details about the results of applying these techniques than to the techniques themselves.

To date, research in object identification has been focused mainly on developing techniques for extracting objects from data that has already been aggregated in programmer-defined data structures [Ruga 95]. Concept analysis is applied in [Siff 99]

[Lind 97][Sahr 97] to identify potential modules. The concept analysis approach generates a variety of possible partitions from which a user can select an appropriate decomposition. The optimality of this approach depends on how well the attributes of functions are formulated for the concept analysis. This method requires that the user have a good knowledge of the system, as the process requires manual selection of appropriate attributes for the system under consideration and a suitable partition. Moreover, the number of partitions obtained is too large for manual inspection (one of the examples given in the paper has over 4000 partitions), and the process provides little help to the software engineer.

A clustering technique is used in [Wigg 97][Yeh 95][Canf 96][Liu 90][Dunn 93] to identify objects. Cohesion-based object identification approaches are presented in [Ache 95][Chu 92]. Gall and Klosch [Gall 95] generate data flow diagrams (DFDs) from the source code and use the data structures used in the data stores as potential object-candidates. Other user-defined data structures that are either declared in or related to the data structures that are already identified as potential object-candidates are also considered as potential object-candidates. This approach is suitable only to programs with user-defined data structures.

In most of the approaches mentioned above, two types of undesired links are identified among sub-graphs - coincidental and spurious connections. Coincidental connections are due to routines that implement more than one function, each function logically belonging to a different object. Spurious connections are related to routines that access more than one data structure. [Canf 96] proposes slicing to separate routines contributing to coincidental connections, and routines that introduce spurious

connections are discarded from the graph. Several techniques do not mention how to identify the spurious or coincidental connections and others specify that it is the role of the software engineer [Liu 90][Dunn 93].

[Canf 96] presents a statistical approach to identify these unwanted connections. [Canf 96] computes an index value for each procedure in the system that measures the internal connectivity of the sub-graph identified through the procedure. If the index value of a procedure is less than a step value, the connections due to the procedure are assumed to be either spurious or coincidental; these procedures are either discarded or eliminated with the help of a human expert. This approach seems to work fine for certain type of systems with a limited number of unwanted connections. However, computation of the step value is subjective and also changes from iteration to iteration for the same system.

Non-code sources, such as documentation and manuals, are also used to find candidate variables for objects [Jaco 91][Gal 95]. This set of variables is further extended based on their relationship with other variables in the system. Processing elements that access these variables are extracted and grouped along with the variables to form object instances. These object instances are normalized to correct the inaccuracies, typically through human interaction, and their classes are abstracted by merging object instances.

The approach we use in this research avoids undesired links by separating the uses from definitions and by replacing direct uses of state variables by selector operations. Automatic identification [Canf 96][Siff 99] of these connections invariably requires human participation. Moreover, the results thus obtained through a laborious

process are not always reliable. We solve this problem by first separating a procedure which defines several state variables, which reduces the number of unwanted connections. The remaining unwanted connections are taken care of with the help of the software engineer by providing the engineer with supporting information. The engineer's task is simplified by the definition of heuristics and general guidelines.

Several of the research initiatives discussed above represent the system as a graph, with functions, global variables, or function attributes as nodes, and references by the procedures to the variables as edges. Each isolated sub-graph contained in the graph is a candidate for an object. The approach we present follows a similar approach, but it extends the graph nodes by including key local variables and program slices.

Chapter 3

Static Analysis of Source Code

3.1 Introduction

Software is difficult and costly to modify. Automating tiresome mechanical tasks, such as program restructuring, reduce the burden of software maintenance. Several restructuring tools have been proposed, all centered on the concept of meaning-preserving transformations that are similar in spirit to compiler optimizations. Like optimizing compilers, these tools rely on static analysis to reason about the correctness of program changes.

Static analysis of programs is indispensable to any software tool, environment, or system that requires compile-time information about the semantics of programs. Static analysis has been used successfully in program maintenance [Gall 91], program integration [Horw 88], transformation [Opdy 92][Gris 95], reverse engineering [Erra 96], and slicing [Jack 94]. Over time, static analysis techniques have been improved. Still, they can only conservatively approximate the control and data flow dependences between different program components. Fortunately, these approximate dependences are generally sufficient for the purpose of program understanding and restructuring; however, the quality of results depends on the quality of the analysis.

Static analysis is also termed dependence analysis or data flow analysis. In section 3.2, we further elaborate on this concept. Previous approaches to static analysis are strongly related to the concept of an intermediate program representation. These representations are the data structures of choice for many types of tools, as their use in

static analysis is well understood. Various types of representations used in the literature are discussed in section 3.3. Section 3.4 discusses meaning preserving transformations. This research defines an analysis technique based on a new program representation that can be used in a wide variety of activities including program slicing, maintenance, and transformation. We present this representation, known as Statement Dependence Graph (StDG) in section 3.5 and provide a brief summary of this chapter in section 3.6.

3.2 Dependence Analysis

A dependence between two program statements is a conflict that prevents the statements from executing concurrently. The use of dependence analysis originated in compiler design for the purposes of optimization and parallelization. The same principles are now being applied to programs for the purposes of debugging, maintenance, and restructuring. Dependences among program statements can be broadly categorized as data dependences, control dependences, and flow dependences.

Data dependences

A data dependence between two program statements indicates that changing the statement's order of execution may change the program's computation. Consider the following sequence of statements:

S1: A = B + C
S2: D = A – E

The value of the variable *A* is defined in *S1* and used in *S2*. Clearly reversing the order of execution of *S1* and *S2* changes the semantic nature of the code. The data dependence that exists between statements *S1* and *S2* is known as a *Definition-Use* (def-use)

dependence. Another type of dependence is a *Use-Definition* [Horw 88] (use-def) dependence, as illustrated in the following sequence of statements:

```
S1:   D = F + G
S2:   F = H – I
```

The value of variable *F* is used in *S1* and defined in *S2*. Again, reversing the order of execution of *S1* and *S2* changes the semantic nature of the code fragment. This type of dependence is also known as *anti-dependence*. Yet another type of data dependence is *Definition-Order* [Horw 88] (def-order) dependence, as illustrated using the following statements.

```
S1:   if (J) K = 0;
S2:   if (L) K = 1;
S3:   M = K;
```

The value of the variable *M* in *S3* clearly depends on the order of execution of statements *S1* and *S2*. Data dependences are typically viewed as a dependence graph. Where the nodes represent statements and the directed edges between nodes represent data dependences.

Control dependences

A control dependence [Horw 88][Liva 92] from statement S_i to statement S_j exists when statement S_j should be executed only if statement S_i produces a certain value. For example, consider the following sequence of statements:

```
S1:   if (N)
S2:         P = Q + R;
```

Execution of *S2* depends on the value of predicate *N*. In the dependence graph, directed edges from the statement containing the predicate to the statements depending on the predicate represent control dependences.

Flow Dependences

A flow dependence exists between statements S_i and S_j, if S_i must be executed before S_j can be executed. The order of execution of output statements, for instance, is flow dependent if the order of the output is important. *Def-order* and *use-def* dependences can be considered as flow dependences.

3.2.1 Issues in static analysis

Granularity

Dependences are analyzed at the expression level [Jack 94][Erns 94][Kinl 94] or at the statement level [Horw 90][Weis 84]. Dependences exists among the variables that are used or defined in a program statement and not among the statements themselves. In a single statement, more than one variable may be defined or used. A course-grained analyses, which considers dependences at the statement level, provides no indication among which variables the dependence exists. A fine-grained analysis considers the dependences among the variables (or expressions). The loss of information in coarse-grained analyses fails to answer important questions about the dependences.

Pointers, aliasing and arrays

Statically finding aliases is a fundamental problem of static analysis. An alias occurs at some point during execution of a program when two or more names exist for the same storage location. This situation can occur due to procedure calls, pointer variables, and array references. For example, the C statement "$p = \&v$" creates an alias between $*p$ and v. Aliases are associated with program points, indicating not only that $*p$ and v refer to the same location during execution but also the location in the program

at which they refer to the same location. Static analysis of languages with pointers, dynamic storage, and recursive data structures is hard (probably NP complete [Land 92]). Hence, the static-analysis community has resorted to simplifying assumptions and approximate solutions, resulting in less precise data flow information which can adversely affect the effectiveness of analysis that depends on this information.

Each element of an array can be considered as an independent variable, but determining the subscript of the array statically is nontrivial. Knowledge of when an array is completely defined is helpful, but this knowledge is not always easy to obtain; hence, the arrays are usually considered as a scalar unit.

Interprocedural analysis

The analysis of the effects of a call is known as interprocedural analysis. Analyzing a program or a procedure (function) determines the sets of variables used and defined by the procedure. Unless these sets can be determined, worst case assumptions must be made. For example, if the procedure includes a call to another procedure, in the absence of information about the called procedure it must be assumed that all variables visible to the called procedure will be used and defined. This assumption, while safe, prevents many structural manipulations. Better results occur if the effects of a call are more carefully analyzed.

Programming language issues

Dependences depend on how a language represents computation and the language constructs used to control the computation. We chose ANSI C to demonstrate the concepts presented in this research. In some ways, C is an ideal choice as it presents

most of the difficulties inherent in existing imperative languages. C is also widely used in industry and academia.

Constituents in a C program can be broadly identified as expressions and statements. Statements may contain expressions; expressions cannot contain statements. Braces, { and }, are used to group statements into a block. A semicolon is a statement terminator. In this research, the following symbols are also considered as statement terminators: '{', '}' (an exception is a `do . . . while` loop, '`}while(expression);`' is placed in one line) , and ')', only if ')' is not followed by a '{'. One statement per line is allowed in programs. Statements express most of the control flow semantics of C. Expressions are syntactic constructs that actually represent program computation, but may also contain embedded control flow. Control in a C program can be thought of as moving from expression to expression [Kern 90]. C, like most languages, does not specify the order in which the operands of an operator are evaluated (including function arguments). Function calls, nested assignment statements, increment operators and decrement operators cause side effects. In expressions involving side effects, there can be subtle dependencies on the order in which variables taking part in the expressions are updated. C allows local jumps (using `continue` and `break`) within a block and non-local jumps (using `goto`), adding to complexity of the analysis. The support of pointers and aliases poses additional challenges during analysis.

3.2.2 Approaches to static analysis

Exhaustive approach

Program analysis approaches can be classified as exhaustive and demand-driven. An exhaustive approach analyzes all the intra- and interprocedural variable dependences

and stores them in persistent data structures. That is, exhaustive approaches represent a portion of the program (often a single function) in its entirety. All facts of the program appear explicitly in these data structures, usually as an edge between two nodes. [Lyle 95][Gris 95][Benn 92].

Demand-driven approach

An alternative approach is demand-driven retrieval of data flow information. This approach has been applied to the construction of complete program representations in a program slicer using abstract syntax trees (AST) and control flow graph (CFG) [Atki 96]. An implementation of the unified interprocedural graph (UIG) in a maintenance environment incrementally constructs sub-graphs for specified procedures when needed for a particular tool [Harr 93]. A framework for deriving demand-driven algorithms for interprocedural data flow analysis of imperative programs is given in [Dues 95]. A demand for data flow information is modeled as a set of data flow queries, and the responses to these queries are found through a partial reversal of the respective data flow analysis. *Cstructure*, a tool for meaning preserving transformations, performs an on demand control flow analysis on the AST representation of C programs, using a technique known as virtual control flow [Morg 97]. The analysis approach we use in this work is a hybrid one. First, we use the exhaustive approach to build a graph representation of the program. Then, the graph is restructured to group the nodes in the graph. Dependences within the grouped nodes are discarded. These discarded internal dependences among grouped statements are obtained on demand.

3.3 Program Representation

The choice of program representation plays a critical role in the analysis and restructuring of the source code. Any suitable representation must be a data structure that can be rapidly traversed to determine the dependence information. Program representations can be broadly divided into two categories [Morg 97]: multiple representations and unified representations. A multiple representation, as the name indicates, uses more than one representation for a program. Unified representations combine several representations into a single, all-purpose representation.

3.3.1 Multiple representations

Multiple representations can be further divided into discarding type and non-discarding type. Non-discard types use a different representation for presenting each view of the code. Discard types use a different representation in each stage of the translation process from source code to the target program.

Non-discard type

The process of program comprehension is often aided by providing the user with different views of the code. Some tools use a different program representation for each view of the code [Gris 95][Plat 91][Unra 95][Choi 94]. [Gris 95] uses abstract syntax trees (AST), control flow graph (CFG), and program dependency graph (PDG). [Plat 91] uses abstract syntax graph (ASG), cross reference graph (CRG), CFG, and data flow graph (DFG). [Choi 94] uses CFG and a static single assignment (SSA) form for program analysis. The SSA form ensures that each use of a variable is reached by

exactly one definition. [Benn 95] uses a language independent format (LIF) and PDG for slicing programs.

Restructuring tools have been built using multiple internal program representations. *Star diagram* [Gris 95] uses three representations, the AST, CFG, and PDG, together with two-way mappings to interrelate them. The mappings become problematic when the program is being modified, as all representations and the mappings between them must be updated. Although they provide comprehensive semantic information, multiple program representations in a software manipulation tool pose a number of serious drawbacks. First, each individual representation must be constructed and stored. In addition, these representations share some redundant information. Second, some types of mapping functions are required to relate items in one representation to items in another. The mappings also consume memory resources. Third, if the tool manipulates the program, some technique is needed to keep all the representations consistent.

Discard type

Discard type representations are built in stages. At each stage, a different representation is built from the previous representation and the source program. However, the steady abstraction of the program from one representation to the next involves a loss of some of the information contained in the previous representations. A tool may need information readily available in a final representation, but it must relate this information to the appropriate location in the source. A tool that does not alter the program could maintain a one-way mapping back to some prior representation. For example, the slicer in [Atki 96] discards the AST after constructing the more useful

CFG. Each CFG node records the node number from the associated location in the original AST. In order to display a slice once it has been computed, an AST identical to the original is recreated on demand from the source code, complete with matching node numbers. Another way to achieve this result is to update the mapping to the source from a prior representation to the current representation, similar to a compiler that saves source file and line number information for debugging purposes. Some of the most commonly used representations are:

- Abstract syntax trees (AST), or parse trees [Aho 86] - This representation, which is closest to source code, may be easily annotated to allow regeneration of the original source [Morg 97]. Each node in the AST usually reflects a production in the context free grammar for the programming language in which the program is written.
- Control flow graph (CFG) [Aho 86] - CFGs form more abstract representation than an AST. The CFG consists of nodes representing the computations in a program connected by edges showing the flow of control from node to node. It can be constructed directly by parsing the source or from an AST. This representation is used for solving many data flow analysis problems.
- Data flow graphs (DFG) [Ferr 87] - An important component of compilation is data-flow analysis that computes information about the potential flow of data throughout a program. Intra-procedural data-flow analysis considers the flow of data within a procedure, while assuming some approximation about definitions and uses of reference parameters and global variables at call sites. Interprocedural data-flow analysis computes information about the flow of data

across procedure boundaries caused by reference parameters and global variables.

- Cross reference graphs (CRGs) [Plat 91] - The CRG provides information on the definition and uses of objects (i.e. variables, types, functions, and macros). CRGs can be used to retrieve information such as a list of all the variables and functions used by a function or all users of a given program object.
- Program dependence graph (PDG) [Ferr 87] – The most abstract of all the representations, the PDG combines data flow and control-flow information into a single structure that is useful for a variety of program transformations and optimizations. The PDG is a labeled, directed, multi-graph where each node represents a program construct such as declarations, assignment statements, and control predicates. Edges of the PDG represent data values passed from one expression to another and control conditions that influence the order of execution. The PDG is useful for compiler optimizations [Ferr 87], program slicing [Otte 84], and transformation [Bow 95].

Other representations exists which use these representations, either modified or in combination. The inter-procedural flow graph (IFG) [Harr 94] is one example. The dependence flow graph, a variant of PDG used in [Ping 90] for analyzing program dependencies, is another example.

3.3.2 Unified representations

Some representations combine several representations into a single, all-purpose graph. This approach overcomes some of the mapping and consistency concerns that arise with the use of multiple representations; however, a single all-purpose

representation may fail to capture all aspects of a program. Many different representations, including the PDG, have been proposed. These representations include:

- System dependence graph (SDG) [Horw 90] - the SDG is an extension to PDG that is limited to a single procedure. The SDG is a super graph of the PDG, which captures the calling context of the called procedures.
- Unified interprocedural graph (UIG) [Harr 91] - The UIG combines SDG and important features of other program representations, while adding new information, to provide an integrated representation. The UIG combines relationships found in a call graph, a program summary graph, an interprocedural flow graph, and a system dependence graph. Data flow dependencies are explicitly represented. As a result the UIG can be quite large and costly to compute [Morg 97].
- Combined C graph (CCG) [Kinl 94] - The CCG is a fine-grained representation for programs written in the C language. The CCG is used for constructing program slices, call graph, flow-sensitive data flow, def-use and control dependence views. The CCG is an extension of the UIG that attempts to overcome some of the UIG's limitations as applied to C programs. The CCG allows the representation of embedded side effects, control flows, and value-returning functions with value parameters. The effects of pointer parameters are also modeled.
- Value dependence graph (VDG) [Weis 94] - The VDG is a sparse, parallel, functional, data flow-like program representation. It is composed of nodes

which represent computation and arcs which carry values between computations. The VDG, unlike SDG, is a very fine-grained representation.

- Jackson model [Jack 94] - Like the VDG, this model also addresses the coarseness of the PDG. In this model each statement is represented as a site and variables as ports within a site. Variable dependences within a statement are represented as internal edges; and data and control dependences are represented as external edges.

A single representation eliminates redundant information and reduces access times to different representations because an algorithm need only access one representation. It also helps comprehension by incorporating all program relationships into one representation [Kinl 94]. Single, all-inclusive representations may create scalability problems. Since none of these data structures explicitly contains all the information required by restructuring transformations, the question of how to obtain the remaining information must still be answered. The StDG representation we present in this work is a fine-grained, discard type, multiple representation. StDG uses a low-level intermediate representation known as language independent format (LIF). The LIF is discarded once the StDG is derived.

3.4 Program Restructuring Transformations

Transformations are the structural (syntactic constructs) changes made to a program to change its appearance, either to improve its maintainability or its speed. Meaning-preserving transformations are the transformations carried out in a controlled environment that allow only those transformations that do not affect the semantics of the

program. A major drawback of meaning-preserving transformations is that every step of the transformation process must ensure that the program semantics remains constant. In this research, we focus on source to source transformations only.

A meaning-preserving transformation has two parts: checks and modifications. Checks determine the legality of the modifications. A transformation check can be divided into four components [Morg 97]: syntactic, scoping, control, and data dependence checks. Syntactic checks ensure that the grammar of the programming language in which the program is written is correct. Scoping checks make sure that the variables are within their declared scopes. The control checks make sure that the dependence of a statement on a test expression is not disturbed. Finally, the data flow dependences are ensured through data dependence checks.

Examples of meaning-preserving transformations are [Opdy 92]:

- folding and unfolding. Folding replaces a code segment with a function call; unfolding expands a function call.
- abstraction. Abstraction substitutes a variable for every instance of an expression and defines the variable to be the value of the expression.
- splitting type. This transformation splits types into subtypes.
- bubble up. This transformation moves a function out of an enclosing module and expands its scope.

To date, transformations have been carried out using a catalog-based approach [Benn 92][Morg 97][Opdy 92]. The catalog includes a large set of transformations covering all aspects of the program development. The *Maintainer's Assistant* lists over 500 transformations [Benn 92]. This approach is useful for interactive tools where the

user selects a segment of the code and a transformation from the catalog. The tool either transforms the code segment, if legal, or fails otherwise. Catalogs are prepared for a particular domain; thus, when the domain changes it becomes less suitable. It is also time consuming to find the best transformation from a large catalog.

This research defines a cohesion-based transformation approach that can be used in non-interactive environments. This approach uses the dependence information in the StDG to ensure that a given transformation does not change the output behavior of the program. These transformations either check to see that changing the source code does not affect the StDG's form or change the StDG in a manner that guarantees semantic preservation.

3.5 Statement Dependence Graph

In this section, we present a new program representation. Previous works on the program representations were aimed at addressing the needs of the program slicing community. Program slicing is a technique for visualizing dependences and restricting attention to just the components of a program relevant to evaluation of certain expressions. The representation used in program slicing is not subjected to modification, and the communications between the representation and the program is straightforward. However, the representations used for restructuring transformations need to change along with the program, and the mapping between the representations and the program must also be updated. We consider the following criteria as important for representations used in program transformations:

- The representation should be simple to construct and use. It should be like a PDG based representation where the program is represented as a graph and dependences are analyzed through graph traversal [Horw 90].
- The representation should be fine-grained; it should indicate the dependences among the variables that are defined and variables that are used in the statements [Lyle 95][Jack 94].
- It should allow user input. Static analysis can only specify dependences conservatively [Opdy 92]. The dependences can be more accurately specified with the help of a maintainer. Therefore, the model should be user understandable and easily modifiable.
- It should allow statement level manipulation. A vast majority of the statements in programs are indivisible. A statement as a unit in the model can have a simple correspondence between the model and the program. All widely used models use many nodes to represent a single statement. Hence, a reference to a statement requires identification of all the nodes and edges that represent a statement, resulting in inefficient and complex algorithms.
- All parts of the program should be represented. Usually, the statements in which variables are either used or defined have a representation in the model. Other statements (e.g., library routines that control the environment) can be ignored in dependence analysis. But, the effect of these statements cannot be ignored in maintenance activities. Similarly, syntactic constructs that do not use any variables have no place in dependence analysis models. A change can be

contemplated and carried out at the model level if it includes all parts of a program.

- A group of statements should have a modular representation. This group can be a slice (defined later) or a procedure. Modularly represented slices or procedures can be analyzed and modified independent of the rest of the program.

All PDG based models satisfy the first criterion. All fine-grained models meet the second criterion. The NDM [Jack 94] satisfies the first two criteria and part of the final criterion (modular representation for a procedure). The representation model developed in this research, StDG, is a fine-grained, PDG based dependence model that satisfies all six criteria.

3.5.1 Terminology

In this section, we define terms used to express the semantics of the StDG.

Site. A site represents a statement in the dependence graph. The site is labeled with the statement number it represents. The site representation concept is taken from [Jack 94].

Port. A port represents a variable. A *use port* represents a variable used and a *def port* a variable defined in a statement. The combination of a variable name and the statement number in which it is used (or defined) is used as a port label. *Use ports* are placed at the top in a site and *def ports* at the bottom of a site in the dependence graph. In addition to the data variable ports, τ, φ, λ, η, are special ports used in the graph. The τ and φ ports are used as structure variables, and the

λ and η ports are used as temporary variables (the need for these variable is explained in section 3.5.2).

Edge. Ports are connected by edges. An edge represents a dependence among the variables; the dependence may be a data dependence, control dependence, or flow dependence. An edge has a source port and a sink port as well as a source site and a sink site.

Reaching defs. A data variable propagates from its definition sites (defined in section 3.2) to the sites where it is used. Reaching defs at a site are the variables used at the site that are defined at other sites. A variable definition reaches a site in three ways: the site is within the scope of the variable's declaration, the variable is global, or the variable is a parameter in a procedure call.

Exposed defs. Exposed defs at a site are the data variables defined at this site that are used at other sites. A variable definition may be exposed to other sites in four ways: the sites are within the same scope, the variable is global, the variable is a parameter passed by reference, or the variable is returned from a function.

Summary-site. A summary site is a site in which several statements are represented. Like other sites, the summary sites use ports are reaching defs, and the def ports are the exposed defs. Dependences within these statements are summarized and are indicated as internal edges among the use and def ports. The site label for the summary site is the smallest statement number among the statements it represents.

Final-use variable. Variable used in statements like output statements and in some function call statements are termed final-use. Statements using final-use

variables are sometimes excluded from certain kinds of operations [Gall 94]; the effects of the use of these variables needs to be evaluated. The effects of a final-use variable are analyzed using a temporary site known as *final-use site*.

Enter site. An enter site is one of two special sites used for each function definition in the dependence graph. It is used to model the calling context of a function call by representing the context before the call. Since only the exposed defs in a calling function reach a called function, the enter site has only the def ports.

Exit site. The exit site is the other special site used in a function representation. It represents the context after the call, thus it has only the use ports.

λ-Statement. A λ-statement defines more than one variable. For example, `a = b++;`, is a C statement with two variable definitions a and b.

φ-Statement. A statement that neither uses nor defines a data variable is a φ-statement. These ststements include certain library routines (e.g. `printf("\n");`, `exit();`), and language constructs (e.g. `continue;`, braces, etc.).

Kill def. A kill def assigns a new value to a variable, replacing its previous value. In other words, the reaching defs of a variable are redefined.

Preserve def. A preserve def redefines a variable conditionally or uses the previous value to assign a new value.

Block. Iterative, selection, and function definition statements define blocks. These statements are called *blockheads*. Statements within a block are *block members*. Block members and a blockhead constitute a *block*.

3.5.2 Semantics of StDG

A dependence relates a variable at one program point to a variable at another. Each statement in the program has two program points, one before and one after the statement. This information can be easily captured by defining a single *site* for each statement and separating variable uses and definitions in the statement [Jack 94]. Variables include the special symbols while sites include the entry and exit sites.

$$\text{Variables} = \text{ProgramVariables} \cup \{\tau, \lambda, \eta, \phi\}$$
$$\text{Var} \subseteq \{v_j \mid v \in \text{Variables} \wedge j \in \text{ProgramStatements}\}$$
$$\text{Site} = \text{ProgramStatements} \cup \{\text{enter, exit}\}$$

A program point is a triple consisting of a variable, a site, and type; where type indicates whether the variable is used or defined.

$$\text{Port} \equiv \text{ProgramPoint}$$
$$\text{ProgramPoint} = \text{Var X Site X Type}$$
$$\text{Type} = \text{Use} \mid \text{Definition}$$

A dependence is an *edge* from one program point to the other. There are two types of edges: internal and external. An internal edge (InternalEdge) is an edge from a program point of use type to a program point of definition type. An external edge (ExternalEdge) is an edge from a program point of definition type to a program point of use type. An external edge may be a data edge (DataEdge), control edge (ControlEdge), or a flow edge (FlowEdge) representing data dependence, control dependence, and flow dependence, respectively. These concepts are defined formally as:

$$\text{Edge: ProgramPoint} \leftrightarrow \text{ProgramPoint}$$

$$\text{InternalEdge} \subseteq \{((v_j, s_l, \text{Use}), (w_k, s_l, \text{Definition})) \mid v_j, w_k \in \text{Var} \wedge s_l \in \text{Site}\}$$

$$\text{ExternalEdge} \subseteq \{((v_j, s_1, \text{Definition}), (w_k, s_2, \text{Use})) \mid v_j, w_k \in \text{Var} \wedge s_1, s_2 \in \text{Site}\}$$

$$\text{DataEdge} \subseteq \{((v_j, s_1, \text{Definition}), (v_k, s_2, \text{Use})) \mid v \in \text{ProgramVariables} \wedge j, k \in \text{ProgramStatements} \wedge s_1, s_2 \in \text{Site}\}$$

$$\text{ControlEdge} \subseteq \{((\tau_j, s_1, \text{Definition}), (\tau_j, s_2, \text{Use})) \mid j \in \text{ProgramStatements} \wedge s_1, s_2 \in \text{Site}\}$$

$$\text{FlowEdge} \subseteq \{((\phi_j, s_1, \text{Definition}), (\tau_k, s_2, \text{Use})) \mid j, k \in \text{ProgramStatements} \wedge s_1, s_2 \in \text{Site}\}$$

Figure 3.1 shows a program and its statement dependence graph. The program counts the number of characters and words in the input stream (please note that the program is written in its current form to facilitate the explanation of the semantics of the StDG). The program has two functions: `main()` and `wordCount()`. Each statement (or a line of code) in the program is given a statement number (ProgramStatement) and is represented by a box (Site) in the graph. Each box has ports (ProgramPoint) at the top and bottom. The top ports represent the variables used and the bottom ports the variables defined in a statement. Representative ports from Figure 3.1(b) are:

Def port: $(nc_6, 6, \text{Definition}) \in \text{Port}$
Use port: $(nc_6, 6, \text{Use}) \in \text{Port}$

Ports are connected by directed edges which connect a use port to a def port. Edges connecting variable ports represent the data dependence (e.g. in Figure 3.1(b), (c_4, c_7) is a data dependence edge). That is,

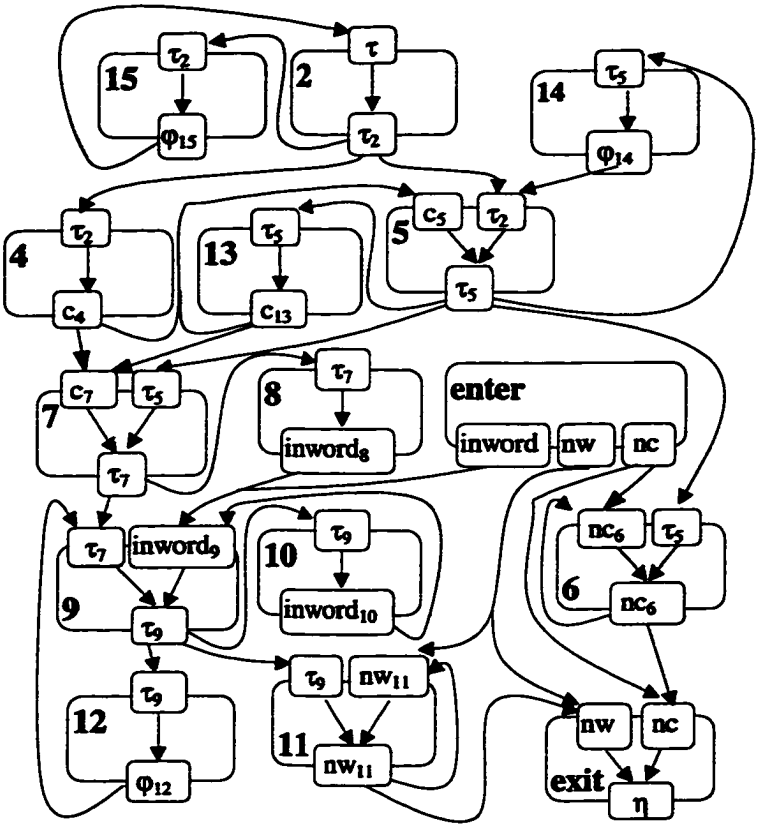
$((c_4, 4, \text{Definition})(c_7, 7, \text{Use})) \in \text{DataEdge}$

A blockhead defines a temporary variable τ (which can be viewed as the result of a conditional test); it is represented in graph as a τ def port with τ and the statement

```
1  int nw, nc;
2  void wordCount(int inword) {
3    int c;
4    c = getchar ();
5    while ( c != EOF ) {
6      nc = nc + 1;
7      if ( c == ' '||c == '\n'
       ||c == '\t' )
8          inword = 0;
9      else if ( inword == 0 ) {
10         inword = 1;
11         nw = nw + 1;
12     }
13   c = getchar ();
14   }
15 }

16 main() {
17   int inw;
18   inw = nc = nw = 0;
19   wordCount(inw);
20   printf ("\n") ;
21   printf (" %d %d\n",nc,nw ) ;
22 }
```

(a)



(b)

(c)

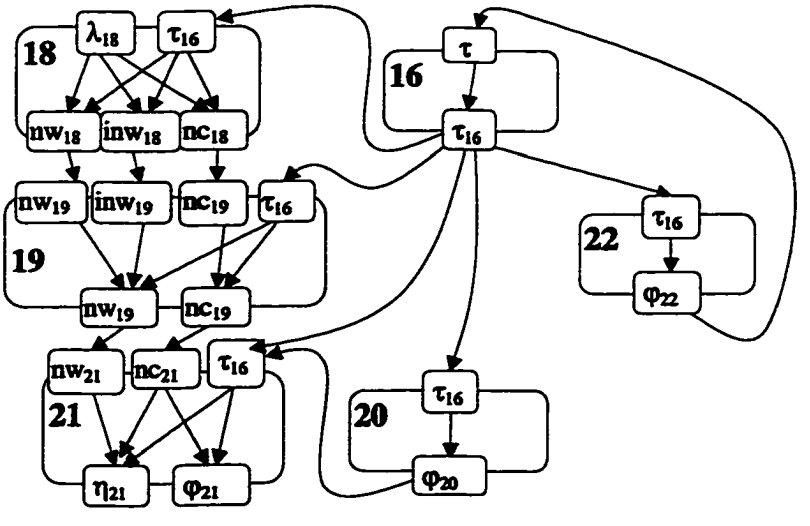


Figure 3.1. (a) Sample C program
(b) The StDG of function wordCount()
(c) The StDG of function main()

number of the blockhead as its label. Block members use the τ defined by their blockhead, with the same label as the blockhead, because the conditional test is the same for all members of a block. In Figure 3.1(b), site 5 defines τ_5, which is used by its member sites: 6, 7, 13, and 14. A function depends on a call statement, so the function name site has a τ-use port with no (unknown) label number. Therefore, all the sites in the dependence graph

have a τ-use port. The τ-def port of the blockhead is connected to τ-use ports of its members with edges. These edges indicate the control dependence of block members on the blockhead. The edge (τ_5, τ_5) in Figure 3.1(c) is an instance of control dependence. That is,

$((\tau_5, 5, \text{Definition}), (\tau_5, 6, \text{use})) \in \text{ControlEdge}$

The dependence of φ-statements among themselves or the dependence of other statements on φ-statements is indicated as flow dependence. User help may be needed to identify this dependence. A φ-def port is used in the φ-statement sites for this purpose; the dependence is indicated as an edge from the φ-port to the τ-use port of the statement that depends on φ-statement. The edge $(\varphi_{20}, \tau_{16})$ in Figure 3.1(c) is an instance of flow dependence. That is,

$((\phi_{20}, 20, \text{Definition}), (\tau_{16}, 21, \text{Use})) \in \text{FlowEdge}$

A def port η indicates the presence of final use variables in a site. An edge from the variable port to a η port indicates that the variable is a final use variable (e.g. variable nw_{22} of site 22 in Figure 3.1(c)). A λ-use port and the edges from the λ-port to

the variables defined in a λ-statement represent a λ-statement in the graph. Site 19 in Figure 3.1(c) is a λ-statement.

Three variables are defined in `main()`: *nw*, *nc* (global), and *inword* (parameter), reach `wordCount()`. Values of these three variables are used in *wordCount()* which is represented in the graph using a enter site. Values of two variables defined in `wordCount()`: *nw*, *nc* (global) reach back to main, which is indicated using a exit site in `wordCount()` (Figure 3.1(b)). The dependences of the variables in the exit site on the variables in the enter site of a function can be summarized into a summary site and used in place of a function call, as in site 20 of Figure 3.1(c).

3.5.3 Construction of StDG

Each statement in the program is given a statement number and is represented by a site labeled with the statement number in the graph. Each statement has a set of uses, *uses(i)*, and a set of definitions, *defs(i)*. Consider, for instance, a statement *i*:

```
i: a = b + c;
```

Uses(i) = (b_i, c_i,τ_x), and

Defs(i) = (a_i) where, *a*, *b*, and *c* are variables. The control variable is τ, and the blockhead statement number of which *i* is a member is *x*. The site representing statement *i* has four ports: two def ports (a_i and τ_x) and two use ports (b_i and c_i). That is,

Port(i) = {(a_i, i, Definition), (b_i, i, Use), (c_i, i, Use) , (τ_x, i, Use)}

Internal edges in a site are the summarized dependences between the variables it defines and the variables it uses. These edges are used as the specification of the site. The specification for a primitive statement follows from the syntax of the statement. For

a function (or a group of statements), the specification is the dependences in the summary site of the function (or a group of statements). The specification of statement *i* is:

$$\text{spec(i)} = ((a_i, \text{i, Definition}), (b_i, \text{i, Use})),$$
$$((a_i, \text{i, Definition}), (c_i, \text{i, Use})), ((a_i, \text{i, Definition}), (\tau_x, \text{i, Use}))$$

Three internal edges - (b_i,a_i), (c_i,a_i), (τ_x,a_i) – are inserted at site *i*. Another way to write the specification is:

$$(((a_i,i),(b_i,i)),((a_i,i),(c_i,i)),((a_i,i),(\tau_x,i)))$$

which is read "*a* of statement *i* (or port a_i) at site *i* is dependent on *b* of *i* (and also, *c* of *i* and τ of *x*) at site *i*". A blockhead *i* using a variable a_i has the specification:

$$(((\tau_i,i),(a_i,i)), ((\tau_i,i), (\tau_x,i)))$$

For each final use variable a_i at site *i*, an edge is added from a_i to η_i, and the following expression is added to its specification.

$$((a_i,i), (\eta_i,i))$$

A φ-statement *i* includes the following edge in its specification:

$$((\varphi_i,i), (\tau_x,i))$$

A λ-statement *i*, defining a_i and b_i, has edges (λ_i,a_i), (λ_i,b_i), (τ_x,a_i), and (τ_x,b_i) in its site and $\text{spec(i)} = ((a_i,i),(\lambda_i,i)), ((b_i,i), (\lambda_i,i)), ((a_i,i), (\tau_x,i)), ((b_i,i), (\tau_x,i))$.

The external edges in the dependence graph represent data, control, and flow dependences. They are known as data, control, and flow edges, respectively. The technique used to construct data flow and control dependence edges is the same used to compute the PDG [Lyle 95][Jack 94]. If data propagates from $(b_k,k$, *Definition)* to $(b_i,i$, *Use)* then a data edge is inserted from $(b_k,k$, *Definition)* to $(b_i,i$, *Use)*. This procedure

is carried out for each k in the program. For each block (i) in the program, the following control edges are inserted in the graph:

$((\tau_i, i, Definition), (\tau_i, k, Use))$, for each member ($k$) of the block.

For each φ-statement i, a flow edge $((\varphi_t i, Definition), (\tau_x j, Use))$ is added, where j is the statement that depends on i. For example, `break` and `continue` statements within a loop are represented in the graph as dependent on the loop statement. On the other hand, if i is a library routine, then the statements which depend on i must be specified by the user.

Library routines are given a surrogate specification in place of code. Site 4, using the `getchar()` routine in the sample program of Figure 3.1, has a specification

Table 3.1. Algorithm for constructing the StDG

```
//Algorithm for constructing the StDG
// input is a program with statement numbers
// each procedure is analyzed separately
//specification for library routines is taken from the user
currentBlock = 0; // statement number of the blockhead
Site = Exit ∪ Enter; //for each procedure add Enter and Exit sites
while not end of procedure {
        j = get next statement ;
        Site = Site ∪ j; //add j to site's set
        for (each v ∈ variables defined in j){
                defs(j) = defs(j) ∪ v_j ;
                if(v ∈ GlobalVariable ∨ v is a final use variable)
                        uses(Exit) = uses(Exit) ∪ v ;
        }
        if( #defs(j) > 1) uses(j) = uses(j) ∪ η_j ;
                //more than one variable defined in j
        for (each v ∈ variables used in j){
                uses(j) = uses(j) ∪ v_j ;
                if(v ∈ GlobalVariable ∨ v ∈Parameter)
```

table cont'd

```
                    defs(Enter) = defs(Enter) ∪ v ;
                if (v is a final use variable)
                    defs(j) = defs(j) ∪ η_j ;
}
        uses(j) = uses(j) ∪ τ_currentBlock ;
        if (statementType(j) == BLOCKHEAD) {
                currentBlock = j ; //current statement is a blockhead
                defs(j) = defs(j) ∪ τ_j ;
        }
        if (statementType(j) == φ-statement) defs(j) = defs(j) ∪ φ_j ;
                //current statement is a φ-statement
//add ports to site j
        for (each v ∈ defs(j))
                Port(j) = Port(j) ∪ (v, j, Definition) ; //Def ports
        for (each v ∈ uses(j))
                Port(j) = Port(j) ∪ (v, j, use) ; //use ports
//add internal edges
        for (each w ∈ uses(j))
        for (each v ∈ defs(j))
                Edge = Edge ∪ ((w, j, Use), (v, j, Definition));
                }
//external edges
        for (each j ∈ Site)
        for (each v ∈ defs(j))
            if(v propagates to w ∧ w ∈ uses(k))
Edge = Edge ∪ ((v, j, definition), (w, k, use));
```

($(c_4,4,Defintion)$, $(\tau_1,1,Use)$). The specification indicates that `getchar()` returns a constant, meaning that it has no specification. The correct specification should be: *((input,4,Definition), (input,4,use))*, which in turn results in the specification of *(($(c_4,4,Definition)$, (input,,Use)), ($(c_4,4,Definition)$, $(\tau_1,1,Use)$))* for site 4. Input has no label number because it doesn't change from site to site. We present an algorithm for constructing the StDG in table 3.1.

3.5.4 Representation of alias information in the graph

Intraprocedural algorithms for alias analysis can be classified into two categories: flow-sensitive and flow-insensitive. Flow-sensitive algorithms consider intraprocedural control flow information during the analysis while flow-insensitive algorithms do not. Flow-sensitive algorithms in general are more precise and less efficient than flow-insensitive algorithms [Burk 95]. Similarly, interprocedural alias analysis algorithms are also classified into two categories - context-sensitive and context-insensitive. Context-sensitive algorithms treat multiple calls to a single procedure independently. They reanalyze a procedure for each of its calling contexts while context-insensitive algorithms do not. Context-sensitive algorithms are more precise and less efficient than context-insensitive algorithms [Hind]. StDG includes flow-sensitive information only; but, at the interprocedural level, both context-sensitive and context-insensitive information can be included.

The data variables (non-pointer) and the pointer variables are represented differently in the StDG. A pointer variable has three components: the address of the object to which it points (reference variable), pointer to the object to which it points (points-to variable, dereferencing), and the object itself (pointed object). The following port labeling convention is followed for the pointer variables:

- All aliased variables are included in the label (both pointer and non-pointer), separated by a comma.
- If de-referencing is involved (indirect assignment by a pointer), the pointed objects are placed within the parenthesis.

S1: p = &r
S2: if( )
S3: q = p;
S4: else
S5: q = &s;
S6: *p = x1
S7: h = q;
S8: q = &t;
S9: x2 = *h;

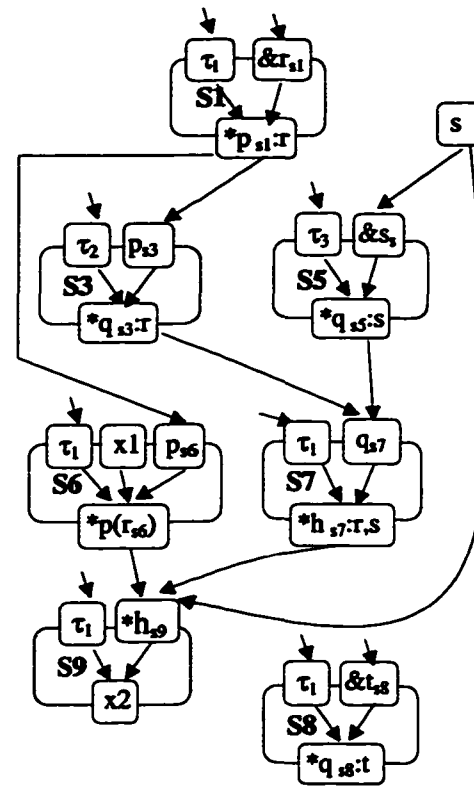


Figure 3.2. StDG with alias information

- If a pointer variable points to more than one variable or more than one pointer variable points-to a variable, then the list of variables is separated by a comma. The points-to label includes the points-to variable, followed by ':' and the pointed objects.

Figure 3.2 shows the StDG for a sample program which uses pointers. Depending on the role of a pointer variable usage in the program, its representation changes in the graph. Uses of reference variables are represented in the normal way. But, the definitions result in a def port with a label that includes points-to information (site S_1 in Figure 3.2). The use of a points-to variable has a use port connected with def ports in the site of the pointed object (site S9 in Figure 3.2). The definition of a points-to variable results in the definition to the pointed object, using the pointer (site S_6 in figure

3.2). Therefore, this pointer definition results in a use port of points-to variable and a def port whose label includes both points-to variable and the pointed object.

3.6 Summary

In this chapter, we presented an overview of static analysis of programs, related issues, and the program representations available in the literature. A program analysis tool may derive all the dependence information and store or derive it on demand. The choice between the two analysis approaches depends on the space or time constraints. The representations used for the analysis may be a multiple or a unified representation. A tool using multiple program representation has to manage the mappings between various representations. If the program changes, all the representations and the mappings must be updated. Single, all-inclusive representation may create scalability problems; it is difficult to store all the dependence information required in a data structure.

We have presented a new representation known as Statement Dependence Graph (StDG). The StDG uses discard type multiple representations; it discards the earlier representation once the next representation is obtained. The analysis approach used in StDG is exhaustive, but it discards the dependence information among groups of cohesive statements. The discarded information is not required for most applications; but, if needed, the discarded information can be obtained on demand. The StDG is a fine-grained representation with modular representation for functions and slices. In Chapter 4, we describe algorithms for restructuring the StDG to simplify it and make it more amenable to changes.

Chapter 4

Restructuring StDG

4.1 Introduction

The graph representation of a program is typically large and cumbersome. It is difficult for a user to manage and expensive for an algorithm to store and traverse all the ports and edges. These problems can be reduced through graph compaction. In most cases, two sites in a graph can be folded into one if they are inseparable. For example, the braces and a compound statement, or iterative statement and `break` statements, can be represented in one site. Further, consider the following code:

```
1: if(a)
2:     b = 10;
```

These two statements cannot be separated for most practical purposes. Hence, they can be represented together in one site and manipulated together. If needed, any interdependencies these two statements may have can be preserved in the new representation. Restructuring is the process of merging sites in the StDG to simplify the graph and make it amenable to changes. A restructured graph is known as a Restructured StDG (RSG). Restructuring retains all the information in the graph, and any operation that can be performed on the StDG can be performed on the RSG.

We define compaction as the process of identifying and merging sites. Compaction involves merging sites by moving a site (source site) into another site (destination site). The source site's ports can be placed at the top or bottom of the destination site, depending on how these two sites are connected. The type of edges

encountered in the merging sites and how the ports and edges are moved during merging are discussed in section 4.2. StDG sites with high cohesion can be merged. Section 4.3 explains cohesion in terms of the graph. Section 4.4 presents an algorithm for merging the sites. Section 4.5 presents RSG, the restructured graph. Section 4.6 presents the summary of the chapter.

4.2 Merging Sites

When a source port is moved, we need to decide whether it should be placed at the top or bottom of the destination site. The ports are moved in such a way that the dependence information is not lost, at the same time reducing the duplicate ports present in the StDG. Depending on where the port is moved, its connections (edges) need to be adjusted. The internal edges of the merging sites and the external edges between the merging sites become internal edges in the merged site. The remaining external edges of both the sites become the external edges of the new site. The notations and definitions needed to define the merging process are presented in section 4.2.1.

4.2.1 Notations and definitions

Port connections and the merging process is explained using the Z specification language [Jack 97]. In the following expressions • is a delimiter.

SITE. Set of all sites in the graph

PORT. Set of all ports in the graph

ExternalEdges. Set of all external edges in the graph

InternalEdges. Set of all internal edges in the graph

Let p1,p2,p:PORT; s1,s2,s3:SITE; p1 = (v_i,s1,Use) $\wedge$ $v_i \in$ Uses(s1) $\wedge$ i $\in$ ProgramStatements. (p1 is a use port in s1.); p2 = (w_j,s2,Use) $\wedge$ $w_j \in$ Defs(s2) $\wedge$ j $\in$ ProgramStatements (p2 is a def port in s2.); p = (y_k,s3,Type) $\wedge$ $y_k \in$ {Defs(s3) $\cup$ Uses(s3) $\wedge$ j $\in$ ProgramStatements $\wedge$ Type $\in$ {Use,Defintion} (p is a port (use or def) in s3).

UPS(s1) = {(v,s1,Use) | $\forall$v• v $\in$ Uses(s1) } – represents all the use ports in site s1 (the top ports).

DPS(s1) = {{(v,s1,Definition) | $\forall$v• v $\in$ Defs(s1) } – represents all the def ports in site s1 (the bottom ports).

DPU(p1) = {p2 | $\forall$p2• (p1,p2) $\in$ InternalEdges}- def ports of a use port p1 (i.e., the ports connected through the def edges of the use port). The DPU of p2 are ports like p1 such that (p1,p2) is an internal edge.

UPD(p2) = {p1 | $\forall$p1• (p1,p2) $\in$ InternalEdges}- use ports of a def port p1 (i.e., the ports connected through the use edges of the def port).

UPU(s1) = {p2 | $\forall$p1,p2• p1 $\in$ UPS(s1) $\wedge$ (p2,p1) $\in$ ExternalEdges}- the use ports of use ports of site (s1) (i.e., the ports connected through the use edges of the use ports of s1). The notation is read "UPU of s1 are ports p2 such that for all p1 and p2, p1 is a use port in s1 and (p2,p1) is an external edge."

UPU(p1) = {p2 | $\forall$p2• (p2,p1) $\in$ ExternalEdges}- the use ports of use port p1.

DPD(s2) = {p1 | $\forall$p2,p1• p2 $\in$ DPS(s2) $\wedge$ (p2,p1) $\in$ ExternalEdges}- the def ports of def ports of site s2. DPD represents all the def ports connected to def ports of the site.

DEU(p1) = {(p1,p2) | $\forall$p2• p2 $\in$ DPU(p1)} - def edges of a use port(p1) (i.e., the outgoing edges from the use port p1). DEU of p1 are edges (p1,p2) such that for all p2, p2 is a DPU of p1.

DPD(p2) = {p1 | $\forall$p1• (p2,p1) $\in$ ExternalEdges}- def ports of a def port p2.

UEU(p1) = {(p2,p1) | $\forall$p2• p2 $\in$ UPU(p1)} - represents the use edges of a use port p1 (i.e., the incoming edges to the use port p1).

UEU(s1) = {(p2,p1) | $\forall$p1,p2• p1 $\in$ UPS(s1) $\wedge$ (p2,p1) $\in$ ExternalEdges}- represents the use edges of all use ports of site s1 (i.e., the incoming edges to s1).

DED(p2) = {(p2,p1) | $\forall$p1• p1 $\in$ DPD(p2)} - def edges of a def port p2.

DED(s2) = {(p2,p1) | $\forall$p2,p1• p2 $\in$ DPS(s2) $\wedge$ (p2,p1) $\in$ ExternalEdges}- def edges of all def ports of s2.

UED(p2) = {(p1,p2) | $\forall$p1• p1 $\in$ UPD(p2)} - use edges of a def port p2.

Site(p1) – returns the site(s) in which the port(s) is present.

BLOCKEND(s1) = max Site(UPU(s1)) – block end site number of site (block) s1.

p1 $\Leftrightarrow$ p2 – indicates that p1 and p2 are the *same ports*. We say that p1 and p2 are the same ports if they have the same label, if variable definition at p2 is used only at p1, or only one variable definition (that of p2) reaches p1.

$\exists$p1,p2• $v_i == w_j \vee$ DPD(p2) == {p1} $\vee$ {p2} == UPU(p1)

$\leftarrow$ – Indicates the action to be taken.

UPS(s1) $\leftarrow$ p. UPS(s1) = UPS(s1) $\cup$ {p1} | $v_i = y_k$ – add port p to set of use ports of s1.

DPS(s2) $\leftarrow$ p. DPS(s2) = DPS(s2) $\cup$ {p2} | $w_j = y_k$ – add port p to set of def ports s2.

p1 ← p2. $v_i = w_j$ – replace p1 by p2. Port replacement is accomplished by replacing label v_i of p1 by w_j of p2.

E1 ← E2. – remove edges E2 and add E1. Edges in E2 are moved because one of the ports (p1 or p2) in it has been moved. Let us assume that the moved port has become port p in its new location (site s). E1 and E2 are sets of edges. Depending on the type of ports present in E1 and E2, the following actions are performed.

- If E2 == {(p1,p2)} and p2 moved. Then add E1 = $\{(p1,p) \mid s3 = s \wedge y_k = w_j\}$
- If E2 == {(p2,p1)} and p2 moved. Then add E1 = $\{(p,p1) \mid s3 = s \wedge y_k = w_j\}$
- If E2 == {(p2,p1)} and p1 moved. Then add E1 = $\{(p2,p) \mid s3 = s \wedge y_k = v_i\}$
- If E2 == {(p1,p2)} and p1 moved. Then add E1 = $\{(p,p2)\} \wedge y_k = v_i\}$. If ($p \in$ UPS(s) $\wedge$ p2 $\in$ UPS(s) (if both p2 and p are use ports) then remove E1 and add UEU(p) ← UEU(p2) and DEU(p) ← DEU(p2) or if ($p \in$ DPS(s) $\wedge$ p2 $\in$ DPS(s) (if both p2 and p are def ports) then remove E1 and add UED(p) ← UED(p2) and DED(p) ← DED(p2).

← E2. – delete edges E2 from the graph.

s1 ← s. merge site s in s1, by moving all ports and edges of s to s1 as explained in section 4.2.2.

← s. SITE = SITE \ {s}. Where SITE \ {s} = {e:SITE | e$\notin$ {s} (set difference). – delete site s from the graph.

G ← s. SITE = SITE $\cup$ {s} – add site s to graph G.

The graph can be traversed forward through def edges and def ports or backwards through use edges and use ports.

4.2.2 Site connections

When moving sites, def ports and their edges are moved first, followed by the use ports and their edges. If the source contains a multi-def port, care should be taken to move only ports and edges connected to the destination site. In the following cases, it is assumed that a multi-def port is not present in the source. All ports are given distinctive labels and are referenced by their labels instead of their specification. The types of ports and edges that can be encountered in the source site (say *j*) and destination site (say *i*), along with the process of merging, are discussed in the following 12 cases.

Case 1: Source def port (T_j) connected to destination use port (T_i). Connected ports are the same.

$$(T_j \Leftrightarrow T_i \wedge T_i \in DPD(T_j)) \wedge$$
$$((UEU(T_j,i) \leftarrow UED(T_j) \wedge T_i \leftarrow T_j) \wedge \leftarrow DED(T_j))$$

If T_i and T_j are the same ports and port T_i belongs to def ports of a def port (T_j), then replace use edges of a def port (T_j) (i.e. edge (τ_2,T_j) in figure 4.1(a)) as use edge of use port (T_j) at site i (i.e. edge (τ_2,T_j)), replace port T_i by port T_j, and remove def edges of def port T_j (i.e. edge (T_j,T_i) in figure 4.1(a)).

The first part (line) of this notation indicates the conditions satisfied in the source and the destination sites, as shown in figure 4.1(a) (i.e., ports are same and connected). The second part (line) indicates how the sites are merged. The merged sites are shown in figure 4.1(b).

Case 2: Source def port (T_j) connected to destination use port (T_i). Connected ports are not the same and source def port has only one def edge.

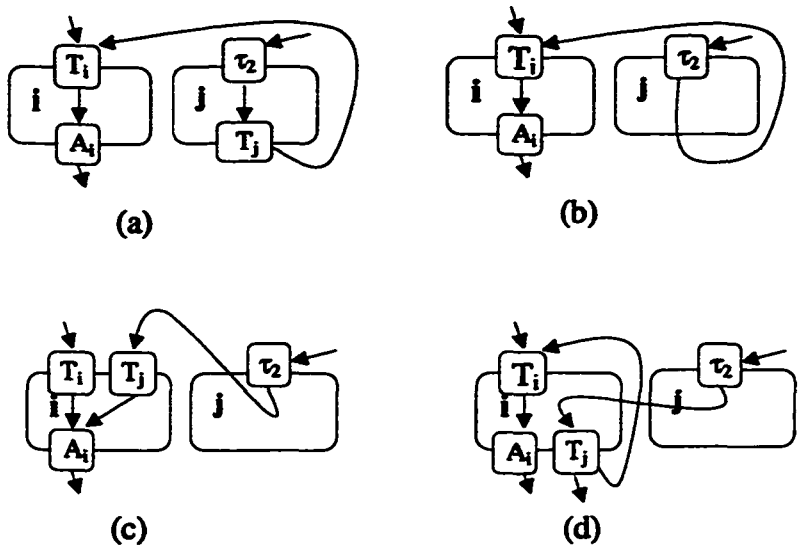


Figure 4.1. (a) Sites before merging (b) Port T_j replaced by T_i
(c) Port T_j moved to UPS(i) (d) Port T_j moved to DPS(i)

$$(\#DED(T_j) = 1 \wedge T_i \in DPD(T_j)) \wedge$$
$$(UPS(i) \leftarrow T_j \wedge UEU(T_j,i) \leftarrow UED(T_j,j) \wedge DEU(T_j,i) \leftarrow DED(T_i,j))$$

If the number of def edges at def port T_j is one and port T_i belongs to def ports of a def port (T_j) (i.e. T_i and T_j are connected), then add port T_j to use ports of site i, replace use edges of a def port (T_j) (i.e. edge (τ_2,T_j) in figure 4.1(a)) as use edges of use port (T_i) (i.e. edge (τ_2,T_i) in figure 4.1(c)), and replace def edges of def port (T_j) at site j (i.e. edge (T_j,T_i) in figure 4.1(a)) as def edges of use port (T_j) in site i (i.e. edge (T_j,A_i) in figure 4.1(c)).

If the number of elements in set $DED(T_j)$ is one and if ports T_j, T_i are connected (as shown in figure 1(a)), then the sites are merged by: 1) moving port T_j to use ports of i, 2) moving use edges of T_j to use edges of T_j in site i, and 3) copying def edges of T_i to def edges of T_j in site i. The source and the destination sites after the def port and its edges are moved to the destination are shown in figure 4.1(c).

Case 3: Source def port (T_j) connected to a destination use port (T_i). Connected ports are not the same and source def port has more than one def edge. The moved port and edges of figure 4.1(a) are shown in figure 4.1(d).

$$(\#DED(T_j) > 1 \wedge DPD(T_j) \cap UPS(i) \neq \varnothing) \wedge$$
$$(DPS(i) \leftarrow T_j \wedge DED(T_j,i) \leftarrow DED(T_j,j) \wedge UED(T_j,i) \leftarrow UED(T_j,j)$$

Case 4: Source def port (T_j) is not connected to the destination site i.

$$DPD(T_j) \cap UPS(i) \neq \phi \wedge$$
$$DPS(i) \leftarrow T_j \wedge UED(T_j,i) \leftarrow UED(T_j,j) \wedge DED(T_j,i) \leftarrow DED(T_j,j)$$

Case 5: Destination def port (T_i) connected to source use port (T_j). Connected ports are the same and source def port is already moved to destination def ports. The two sites are shown in figure 4.2(a), and the merged sites are shown in figure 4.2(b).

$$(T_j \Leftrightarrow T_i \wedge T_i \in UPU(T_j) \wedge DPU(T_j) \in DPS(i)) \wedge (UED(DPU((T_j)) \leftarrow UED(T_i))$$

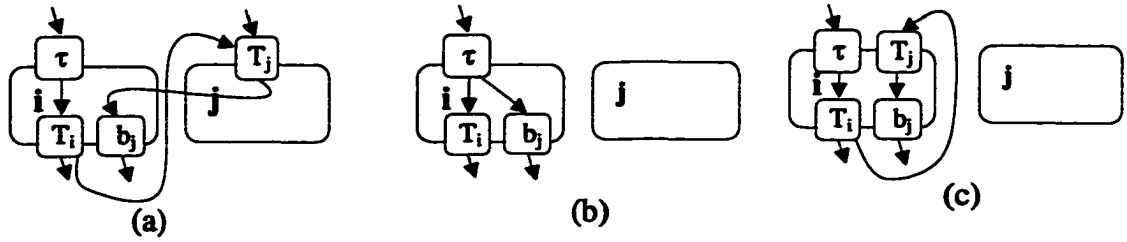


Figure 4.2. (a) The sites before merging.
(b) Sites after replacing port T_j by T_I
(c) Sites after moving port T_j to UPS(i)

Case 6: Destination def port (T_i) connected to source use port (T_j). Connected ports are not the same and the source def port is already moved to the destination def ports. The ports and edges of the two sites are shown in figure 4.2(a), and the merged sites are shown in figure 4.2(c).

$$(T_i \in UPU(T_j) \wedge DPU(T_j) \in DPS(i)) \wedge$$
$$UPS(i) \leftarrow T_j \wedge UEU(T_j,i) \leftarrow UEU(T_j,j) \wedge DEU(T_j,i) \leftarrow DEU(T_j,j))$$

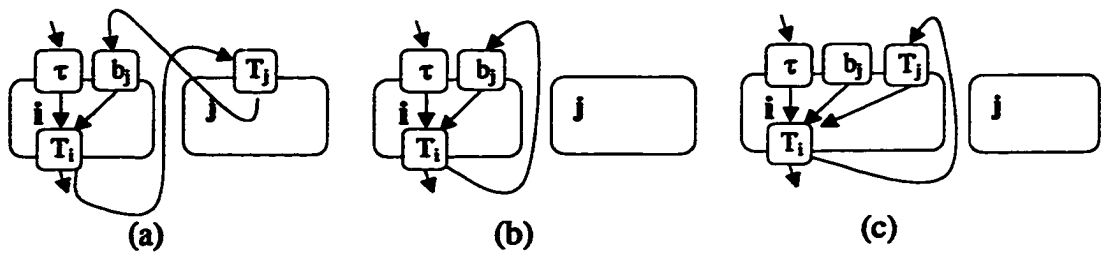


Figure 4.3. (a) The sites before merging.
(b) Sites after replacing port T_j by T_i
(c) Sites after moving port T_j to UPS(i)

Case 7: Destination def port (T_i) connected to source use port (T_j). Connected ports are the same and the source def port is already moved to the destination use ports. The port and edges of the two sites are shown in figure 4.3(a), and the merged sites are shown in figure 4.3(b).

$$(T_j \Leftrightarrow T_i \wedge T_i \in UPU(T_j) \quad \wedge DPU(T_j) \in UPS(i)) \wedge$$
$$(UEU(DPU((T_j)) \leftarrow UEU(T_j))$$

Case 8: Destination def port (T_i) connected to source use port (T_j). Connected ports are not the same and the source def port is already moved to destination use ports. The merged sites are shown in figure 4.3(c).

$$(T_i \in UPU(T_j) \wedge DPU(T_j) \in UPS(i)) \wedge$$
$$UPS(i) \leftarrow T_j \wedge UEU(T_j,i) \leftarrow UEU(T_j,j) \wedge DEU(T_j,i) \leftarrow DEU(DPU(T_j,j)))$$

Case 9: Source use port (T_j) is connected to destination use port (i.e., def port of source already moved to destination use ports). There is a use port (A) in destination whose label is the same as that of the source use port (T_j). No source use port is connected with the destination def ports. The two sites are shown in figure 4.4(a), and the merged sites are shown in figure 4.4(b).

$$(DPU(T_j) \in UPS(i) \wedge (\exists A: PORT \bullet A \in UPS(i) \wedge A \Leftrightarrow T_j) \wedge$$
$$UPU(T_j) \cap DPS(i) = \varnothing) \wedge$$
$$(UEU(A) \leftarrow UEU(T_j) \wedge DEU(A) \leftarrow DEU(DPU(T_j))$$

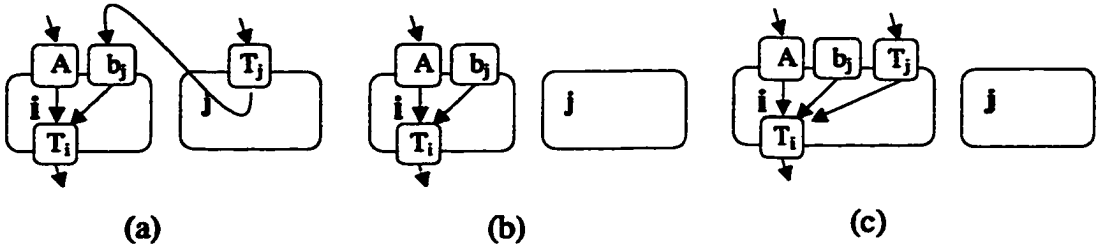


Figure 4.4. (a) The sites before merging.
(b) Sites after replacing port A by T_j
(c) Sites after moving port T_j to UPS(i)

Case 10: Source use port (T_j) is connected to a destination use port (i.e., def port of source already moved to destination use ports). No two use ports in source and destination have the same label. The source use port is not connected with destination def ports. The merged sites are shown in figure 4.4(c).

$$(DPU(T_j) \in UPS(i) \wedge (\forall A: PORT \bullet A \in UPS(i) \wedge A \notin UPS(j)) \wedge$$
$$UPU(T_j) \cap DPS(i) = \varnothing) \wedge$$
$$(UPS(i) \leftarrow T_j \wedge UEU(T_j,i) \leftarrow UEU(T_j) \wedge DEU(T_j,i) \leftarrow DEU(DPU(T_j))$$

Case 11: Source use port (T_j) is connected to destination def port (i.e., def port of source already moved to destination def ports). There is a use port in the destination port with the same label as the source use port (T_j). The source use port is not connected with the destination def ports. The port and edges of the two sites are shown in figure 4.5(a), and the merged sites are shown in figure 4.5(b).

$$(DPU(T_j) \in DPS(i) \wedge (\exists A: PORT \bullet A \in UPS(i) \wedge A \in UPS(j)) \wedge$$
$$UPU(T_j) \cap DPS(i) = \varnothing) \wedge$$
$$(UEU(A) \leftarrow UEU(T_j) \wedge DEU(A) \leftarrow DEU(T_j))$$

Case 12: Source use port (T_j) is connected to destination def port (i.e., def port of source already moved to destination def ports). No two use ports in source and destination have

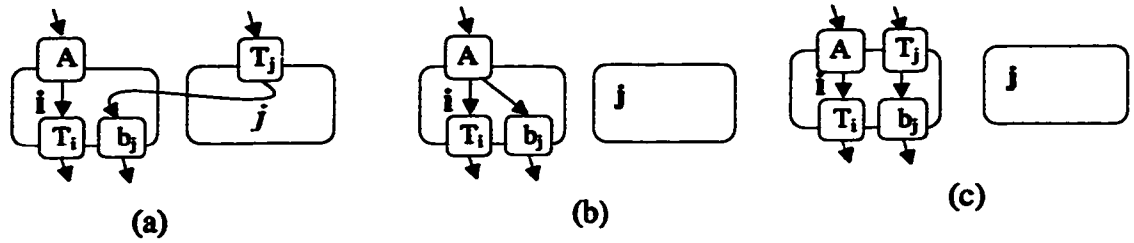


Figure 4.5. (a) The sites before merging.
(b) Sites after replacing port A by T_j
(c) Sites after moving port T_j to UPS(i)

the same label. The source use port is not connected with the destination def ports. The merged sites are shown in figure 4.5(c).

$$(DPU(T_j) \in DPS(i) \wedge (\forall A: PORT \bullet A \in UPS(i) \wedge A \notin UPS(j)) \wedge$$
$$UPU(T_j) \cap DPS(i) = \varnothing) \wedge$$
$$(UPS(i) \leftarrow T_j \wedge UEU(T_j,i) \leftarrow UEU(T_j,j) \wedge DEU(T_j,i) \leftarrow DEU(T_j,j))$$

In section 4.3, we describe which sites in the graph can be merged.

4.3 Compaction

Compaction is based on cohesion. Cohesion indicates the bonding strength between two elements of a program; binding strength is indicated as edges (dependences) in the graph. If a variable definition at a site is used at only one other site, then the cohesion between the variable definition and use sites is high. On the other hand, if the variable is used at more than one site, the cohesion is divided among all the sites that share the variable. We consider cohesion among two sites A and B as high if "A and B are connected, and every site reachable from A is also reachable from B." In terms of the graph, two connected sites are considered highly cohesive if they satisfy any of the three conditions:

i. All the def edges of site A reach only site B,

ii. A group of sites are circularly connected, or

iii. All the sites connected to site A are also connected to site B.

If the cohesion between two statements is high, they can be put together and used as a unit. Cohesion among statements using structural variables and data variables is further explored as structural compaction and data compaction.

4.3.1 Structural compaction

A procedure in a program can be viewed as a block. A block consists of statements and other blocks (members). In a block, all members are dependent (dependence includes data, control, and flow dependence) on the blockhead and, in turn, on all the members on which the blockhead depends. Block members on which the blockhead depends and the blockhead form a circular chain of sites. Hence, these member sites can be merged into the blockhead site. By this merging, a block is separated into a structure part and a data part. The structure part includes the blockhead and a subset of the members (these may include members of a deeper block) which define variables referenced by the blockhead. The data part consists of the remaining members of the block.

A block member site is merged in its blockhead site if there is an edge from the member's def port to the blockhead's use port. That is, if the site number of the site from where a use edge of a blockhead originates is between the block begin and block-end site numbers, then that site can be merged with the blockhead. Let S_1 be a blockhead and S_2 one of the members of S_1 (S_2 may be a member of a deeper block within S_1). S_2 is merged in S_1 if there is an edge from S_2 to S_1. That is,

$$\exists a: \text{PORT}; S_1, S_2: \text{SITE} \bullet$$
$$a \in \text{UPU}(S_1) \wedge a \in \text{DPS}(S_2) \wedge S_2 > S_1 \wedge S_2 < \text{BLOCKEND}(S_1) \wedge$$
$$S_1 \leftarrow S_2$$

In other words, if a port (*a*) belongs to UPU of site S_1 and def ports of a site S_2 (S_1 and S_2 are connected), and the site number of S_2 is greater than S_1 and less than the site number of the block end of S_1 (S_2 is a member of S_1), merge S_2 in S_1. In figure 4.6, Sites 5 (blockhead), 13 (block member) and 14 (another block member) are candidates for structural compaction.

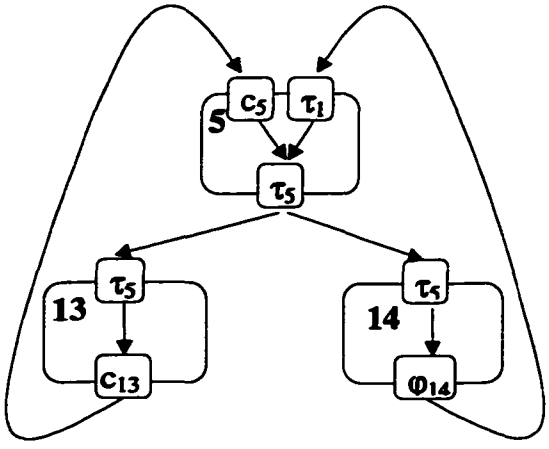


Figure 4.6. Candidate sites for structure compaction

4.3.2 Data compaction

Data compaction involves a data variable defining site (source) and (sink) sites that use the variable. Data compaction depends on how many sink sites are present and how they are connected to the source and among themselves. One of the sinks will be a destination site. If a sink site has a *final use* variable as its def port, then a temporary sink site known as *final use site* is created. This temporary site is used as the destination site to combine the source. The temporary site is needed because a final-use site cannot be merged with any site as it might use variables from different sources.

Site S_2 is merged with site S_1 if the two sites satisfy any of the following three conditions:

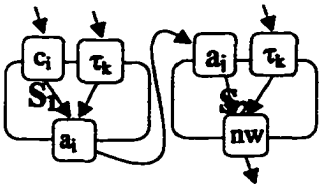


Figure 4.7.All def edges from S_1 reach S_2

1. All edges from S_1 reach S_2 only (e.g. a definition of a variable is used at one another statement only). That is,

$\forall S1,S2: SITE \bullet$
$DPD(S_1) \subseteq UPS(S_2) \cup UPS(S_1) \wedge S_1 \leftarrow S_2$

In other words, if the set of all def ports of def ports (DPD) of S_1 are either use ports in S_1 or S_2, then merge S_1 in S_2. The sites are shown in figure 4.7.

2. There is an edge from S_1 to S_2 and also an edge from S_2 to S_1 (circular dependence), as shown in figure 4.8. That is,

$\exists a,b: PORT; S1,S2: SITE \bullet$
$a \in DPS(S_1) \wedge b \in DPS(S_2) \wedge DPD(a) \cap UPS(S_2) = \varnothing \wedge$
$DPD(b) \cap UPS(S_1) = \varnothing \wedge S_1 \leftarrow S_2$

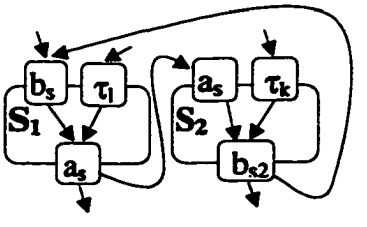


Figure 4.8. Sites S_1 and S_2 are interdependent

In other words, 'a' is a def port in S_1, 'b' is a def port in S_2. DPD of 'a' is a use port in S_2 (S_1 is connected to S_2). DPD of 'b' is a use port in S_1. Then merge S_2 in S_1.

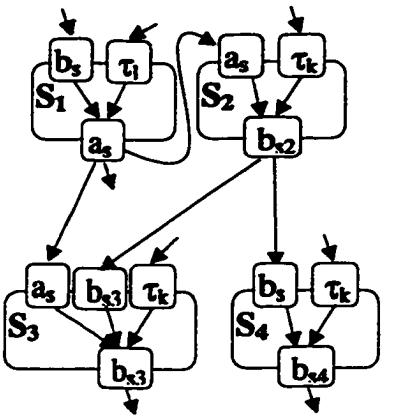


Figure 4.9. Sites reachable from S_1 are also reachable from S_2

3. There is an edge from S_1 to S_2 and all the sites connected with S_1 are also connected with S_2, as shown in figure 4.9. That is,

$$\exists S1, S2: SITE \bullet$$
$$S_2 \in Site(DPD(S_1)) \wedge (Site(DPD(S_1)) - S_2) \subseteq Site(DPD(S_2)) \wedge$$
$$S_2 \leftarrow S_1$$

In other words, S_2 is a site in the set of sites connected to def ports of def ports of S_1. Sites that contain def ports of def ports of S_1, excluding S_2, are a subset of sites that contain def ports of def ports of S_2. Then merge S_1 in S_2. A special case of this type of connection is a blockhead (S_2) and, S_1 connected to S_2 and one or more of its member sites.

4.3.3 Edge compaction

Reducing the number of sites connected to a site can increase the possibilities for compaction. This reduction in site connections can be achieved by changing a direct edge between two sites into an indirect connection through another connected site. For example, def edges of a site S_1 (connected with S_2 and S_3) can be reduced by changing a direct edge between S_1 and S_3 into an indirect connection through S_2, as shown in figure

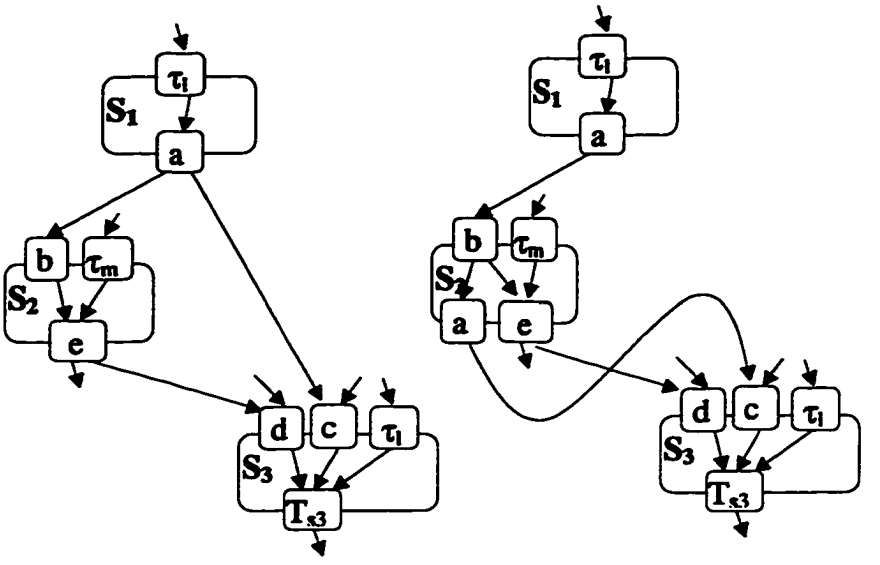


Figure 4.10. (a) Candidate sites for edge compaction (b) Edge compacted sites.

4.10(a). This concept is similar to replacing a global variable by a local variable and passing it as a parameter. In figure 4.10, the dashed lines indicate that the connection need not be a direct edge. Sites S_1 and S_2 in figure 4.10(b) can be merged as all def edges from S_1 reach S_2. Edge compaction is carried out only to test if two sites can be merged. That is,

$$\exists a,b,c: \text{PORT}; S_1,S_2,S_3: \text{SITE} \bullet$$
$$a \in \text{DPS}(S_1) \wedge b \in \text{UPS}(S_2) \wedge c \in \text{UPS}(S_3) \wedge$$
$$b \in \text{DPD}(S_1) \wedge c \in \text{DPD}(S_1) \wedge S_3 \in \text{FSLICE}\ S_2$$

In other words, conditions for edge compaction: 'a' is a def port in S_1, 'b' is a use port in S_2, and 'c' is a use port in S_3. 'b' is a DPD of S_1 (i.e. S_2 and S_1 are connected), 'c' is a DPD of S_1 (i.e. S_3 and S_1 are connected), and S_3 belongs to FSLICE of S_2 (i.e. S_3 and S_2 are connected indirectly). Where,

$$\text{FSLICE}: \text{SITE} \rightarrow \text{SITE}$$
$$\forall S_1: \text{SITE} \bullet$$

$$\text{FSLICE } S_1 = \text{Site}(\text{DPD}(S_1)) \wedge$$
$$\text{FSLICE } S_1 = \text{FSLICE FSLICE } S_1 \cup \text{FSLICE } S_1$$

(FSLICE of S_1 is sites belonging to def ports of def ports of S_1 or $(\text{DPD}(S_1))^+$.)

Edges and ports of sites, being edge compacted, are moved as follows:

$((a \Leftrightarrow b) \wedge$	- a & b same ports
$\text{DPS}(S_2) \leftarrow a \wedge$	- add def port a to S_2
$S_2 \leftarrow (b, a) \wedge$	- add an internal edge
$((a, S_2),(c, S_3)) \leftarrow ((a, S_1),(c, S_3))) \vee$	- move edge
$((a \text{ not}\Leftrightarrow b) \wedge$	- a & b not same ports
$\text{DPS}(S_2) \leftarrow a \wedge$	- add def port
$\text{UPS}(S_2) \leftarrow a \wedge$	- add use port
$S_2 \leftarrow (a, a) \wedge$	- add an internal edge
$S_2 \leftarrow ((a, S_1),(a, S_2))) \wedge$	- move edge
$((a, S_2),(c, S_3)) \leftarrow ((a, S_1),(c, S_3)))$	- move edge

An algorithm for merging sites using data compaction, structure compaction, and edge compaction is presented in section 4.4.

4.4 Compaction Algorithm

The compaction algorithm, which follows, includes numerous embedded comments to help explain the details of the algorithm.

****

/* Process each site by testing if one of the def ports label is τ. If τ is present then the site is a blockhead. If the site is a blockhead, then call STR_COMP else call DATA_COMP. */

PROCESS_SITE: SITE → SITE
$\forall j$: SITE•
$(\tau_j \in \text{Defs}(j) \wedge \text{STR\_COMP } j) \vee (\tau_j \notin \text{Defs}(j) \wedge \text{DATA\_COMP } j)$

/* For each blockhead site, call FOLD_STR & DATA_COMP, and for each member site of the block, call PROCESS_SITE. */

STR_COMP: SITE → SITE
∀j: SITE•
FOLD_STR j ∧
DATA_COMP j ∧
∀k: SITE• (τ_j,k,Use) ∈ DPD(τ_j) ∧ PROCESS_SITE k

/* CHANGE is of boolean type. Call FOLD_DATA. If FOLD_DATA results in site folding, then call DATA_COMP for each site connected to the use ports of the site. Also, call FOLD_EDGE, if FOLD_EDGE changes edges, then call DATA_COMP. */

DATA_COMP: SITE → SITE
∀j: SITE; change: boolean •
FOLD_DATA j ∧
(change = true ∧
∀k: SITE• k ∈ site(UPU(j)) ∧ DATA_COMP k)
FOLD_EDGE j ∧
(change = true ∧
DATA_COMP j)

/* FOLD_DATA uses the cases in data compaction (previous section) for testing the type site connections and for merging the sites. Input is a site and a boolean variable, which is set to true if FOLD_DATA merges any sites. For explanation, see data compaction, in section 4.3.2. */

FOLD_DATA: SITE → (SITE, boolean)
∀S_1: SITE; ch: boolean; ∃a,b: PORT;S_2: SITE •
($a \in UPU(S_1) \wedge a \in DPS(S_2) \wedge S_2 > S_1 \wedge S_2 < BLOCKEND(S_1) \wedge S_1 \leftarrow S_2 \wedge ch := true$) ∨
- case 1 in data compation.

($a \in DPS(S_1) \wedge b \in DPS(S_2) \wedge DPD(a) \cap UPS(S_2) = \varnothing \wedge DPD(b) \cap UPS(S_1) = \varnothing \wedge S_1 \leftarrow S_2 \wedge ch := true$) ∨
- case 2 in data compation.

($S_2 \in Site(DPD(S_1)) \wedge (Site(DPD(S_1)) - S_2) \subseteq Site(DPD(S_2)) \wedge S_2 \leftarrow S_1 \wedge ch := true$)
- case 3 in data compation.

/* For an explanation see structure compaction, in section 4.3.1. */

FOLD_STR: SITE → SITE
$\forall S_1$: SITE; $\exists$a: PORT;S_2: SITE •
$a \in UPU(S_1) \wedge a \in DPS(S_2) \wedge S_2 > S_1 \wedge S_2 < BLOCKEND(S_1) \wedge$
$S_1 \leftarrow S_2$

/* For an explanation see edge compaction, in section 4.3.3. */

FOLD_EDGE: SITE → SITE
$\exists a,b,c$: PORT; S_1,S_2,S_3: SITE •
$(a \in DPS(S_1) \wedge b \in UPS(S_2) \wedge c \in UPS(S_3) \wedge$
$b \in DPD(S_1) \wedge c \in DPD(S_1) \wedge S_3 \in FSLICE\ S_2) \wedge$
$(((a \Leftrightarrow b) \wedge DPS(S_2) \leftarrow a \wedge S_2 \leftarrow (b, a) \wedge ((a, S_2),(c, S_3)) \leftarrow ((a, S_1),(c, S_3))) \vee$
$((a\ not\Leftrightarrow\ b) \wedge DPS(S_2) \leftarrow a \wedge UPS(S_2) \leftarrow a \wedge S_2 \leftarrow (a, a) \wedge$
$S_2 \leftarrow ((a, S_1),(a, S_2))) \wedge ((a, S_2),(c, S_3)) \leftarrow ((a, S_1),(c, S_3))))$

FSLICE: SITE → SITE
$\forall S_1$: SITE•
$FSLICE\ S_1 = Site(DPD(S_1)) \wedge FSLICE\ S_1 = FSLICE\ FSLICE\ S_1 \cup FSLICE\ S_1$

****

4.5 Restructured StDG (RSG)

The restructured statement dependence graph is known as a Restructured StDG (RSG). Applying the structure and data compaction to the StDG of figure 3.1(b) results in the RSG shown in figure 4.11. In figure 4.11, only the statements represented by each site with reaching defs and exposed defs are shown.

Restructuring has several other advantages apart from graph size reduction. It helps to understand programs by localizing the slices of code. Program slices can be easily computed. These slices, unlike the traditional slices [Horw 90] can be closed-ended [Jack 94]. That is, a slice can have both a beginning and an end. The RSG depends on the type of compactions applied to the StDG, and the type of compactions needed depends on the type of application.

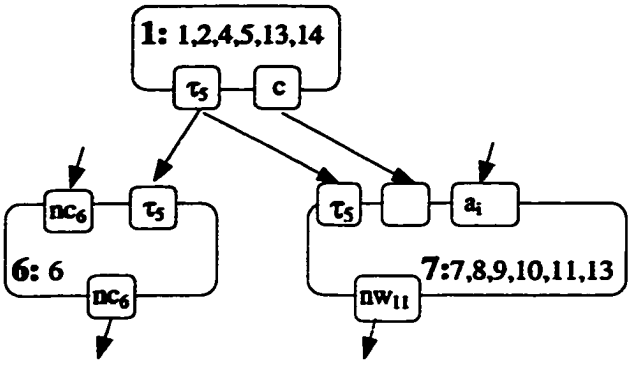


Figure 4.11. Restructured StDG

4.6 Summary

The StDG can be compacted by merging the sites that exhibit high cohesion. The StDG is restructured through compaction, and the resulting graph is known as the Restructured StDG (RSG). In this chapter, we described how cohesive sites in the graph are identified and merged. The type of restructuring that is required changes from application to application; however; the fundamental representation graph StDG and the compacted graph RSG provide a theoretically sound framework that provides support for many problems found in the reengineering domain. In Chapters 5 and 7 we present convincing evidence of the value of StDG and RSG in three such domains.

Chapter 5

Application of RSG to Slicing and Maintenance

5.1 Introduction

The RSG can be applied to multiple applications such as program slicing, maintenance, program understanding, and function extraction. In this chapter, we apply the RSG to program slicing and maintenance. The application of the RSG to program slicing is presented in Section 5.2, and the application of the RSG to program maintenance is presented in Section 5.3. Section 5.4 presents the summary of the chapter.

5.2 Slicing

The *slice* (also known as a *backward slice*) of a program with respect to program statement p and variable x consists of all statements and predicates of the program that might affect the value of x at p. A *forward slice* of a program with respect to program statement p and variable x consists of all statements and predicates of the program that might be affected by the value of x at p. $S(x, p)$ is called a *slicing criteria*. Weiser [Weis 81] introduced slicing. Ottenstein [Otte 84] presented a linear time algorithm to find an intraprocedural slice using a PDG representation of the program. Horwitz [Horw 90] improved these algorithms to construct interprocedural slices by using the SDG. Slicing is used to isolate individual computation threads within a program. Slicing has been successfully used in a variety of application like, program understanding and debugging [Weis 82][Lyle 84][Lyle 86], integrating programs [Horw 89], automatic parallelization

[Erra 96], function recovery [Visa 93], program maintenance [Gall 91], and reverse engineering [Jack 94],

PDG-based slicers reflect statement level dependences but they fail to answer queries regarding dependences among program variables. For instance consider the statement:

```
a = b + c.
```

A PDG-based slicer equates three distinct criteria: the definition of *a*, the use of *b* and the use of *c*. To address this problem, fine-grained dependence models like value dependence graphs (VDG) [Weis 94] and statement dependence graph based on parse trees [Liva] have been proposed. However, the utility of slicing for maintenance and reverse engineering is less obvious. Not all questions can be cast as slice criteria [Jack 94]. This situation can be seen in the approach to maintenance (in [Gall 91]), where dozens of definitions and propositions are used in place of slicing criteria with limited success. Furthermore, slices often turn out to be too coarse to be useful. One reason for this coarseness is because slicing cannot discriminate origins; every statement that affects the given variable is included in the slice, whatever the source of its dependence. The Jackson [Jack 94] chopshop tool aims to overcome these problems. In place of slice criterion, this model allows the user to pick a source and a sink. The tool identifies the statements that cause the source to affect the sink.

The RSG based model we developed is a further improvement over existing models. These improvements are:

i. Current models include only the statements that contribute to the dependences in the graph. Hence, these models require some kind of mapping function to

identify syntactic constructs that do not contribute to the dependences during slicing. The RSG model requires no such mapping function.

ii. Statements with high cohesion (defined earlier) are either all present or absent in a traditional slice. Hence, these statements can be put together, and the slicer can include all these statements in a slice when one of these statements is reached. The merged graph will have fewer nodes to store and traverse.

iii. A user can select multiple sources and sinks; this feature is useful in function extraction. The RSG-based slicer can answer queries like – given certain variables as input (like parameters to a function), what computation is required to compute certain variable(s).

iv. Statement groups help the user select appropriate sources and sinks.

v. Slicing algorithms of PDG based models compute a slice with two traversals of the SDG. The cost of each traversal is linear in size of the SDG [Horw 90]. These models have one node in the graph for each statement in the program. In the RSG model, one node (site) represents several statements and a slice is performed with one traversal of the RSG.

5.2.1 Formalization of modular slicing

Two sets of variables and statement numbers form a criterion for slicing: *source* and *sink* (a set of ports). A *modular slice* includes all the statements needed to compute the variables in the *sink*, using the variables in the *source*. That is,

$$\text{source} = \textit{Variables} \times \textit{StatementNumber} = \{(v,s1) \mid v \in \text{ProgramVariable} \wedge s1 \in \text{ProgramStatement}\}$$

$$\text{sink} = \textit{Variablse} \times \textit{StatementNumber} = \{(w,s2) \mid w \in \text{ProgramVariable} \wedge s2 \in \text{ProgramStatement}\}.$$

By translating the variables and statement numbers to ports, we have;

$$source = \{(v_{s1}, s1, T) \mid s1{:}SITE \bullet (v_{s1} \in Uses(s1) \wedge T = Use) \vee (v_{s1} \in Defs(s1) \wedge T = Definition)\}$$

$$sink = \{(w_{s2}, s2, T) \mid s2{:}SITE \bullet (w_{t2} \in Defs(s2) \wedge T = Definition) \vee (w_{t2} \in Uses(s2) \wedge T = Use)\}$$

In the StDG, site numbers and statement numbers are the same, whereas in the RSG they differ because a site may represent more than one statement.

The StDG can be traversed forward (from enter to exit site) through the def edges and the def ports. So, to reach sink from source, we go from source to def ports of source, from def ports of source to def ports of def ports of source, and so on until we reach the sink. Def edges and def ports of source are internal edges and def ports in source sites, if the source has use ports. Def edges of def ports of source are the external edges connected to source sites. Let P be set of ports in source and S be set of source sites. The ports and edges in the forward slice are computed as follows:

P = Source ports
Internal edges in S connected to P = EF1 = DEU(P) (if P has use ports)
= Null (if P has no use ports)
Internal ports in S connected to P = PF1 = DPU(P)(if P has use ports)
= P (if P has no use ports)
External edges connected to PF1 = EF2 = DED(PF1)
External ports connected to PF1 = PF2 = DPD(PF1)

Let CS be the sites connected to S through external edges (EF2). PF2 represents use ports in CS. Then,

CS = Site(PF2)
Internal edges in CS = EF3 = DEU(PF2)
Internal ports in CS connected to PF2 = PF3 = DPU(PF2)
External edges connected to CS = EF4 = DED(PF3)
External ports connected to CS = PF4 = DPD(PF3)

The site connected to CS is Site(PF4). Continuing further, we have:

EF5 = DEU(PF4)
PF5 = DPU(PF4)
EF6 = DED(PF5)
PF6 = DPD(PF5), and so on.

A forward slice (FS) on a port (or variable) is all the edges (FSE), external edges (FSEE), internal edges (FSEI), and ports (FSP) that are reached by traversing forward from the port. Firmally,

FS = Site(PF1) ∪ Site(PF2) ∪ Site(PF3) ∪ Site(PF4) . . .
FSP = P ∪ PF1 ∪ PF2 ∪ PF3 ∪ PF4 . . .
FSE = EF1 ∪ EF2 ∪ EF3 ∪ EF4 ∪ EF5 ∪ . . .
FSEI = EF1 ∪ EF3 ∪ EF5 ∪ EF7 ∪ EF9 ∪ . . .
FSEE = EF2 ∪ EF4 ∪ EF6 ∪ EF8 ∪ EF10 ∪ . . .

Similarly, the StDG can be traversed backward (from exit to enter site) through the use edges and the use ports. So, to reach source from sink, we go from sink to use ports of sink, from use ports of sink to use ports of use ports of sink, and so on until we reach the source. Use edges of sink are internal edges in sink sites and use ports of sink are ports within sink sites, if sink has def ports. Use edges of use ports of sink are the external edges connected to sink sites. Let P be the set of ports in sink and S be set of sink sites. The ports and edges in the backward slice are computed as follows:

Internal edges in S connected to P = EB1 = UED(P) (if P has def ports)
= Null (if P has no def ports)
Internal ports in S connected to P = PB1 = UPD(P)(if P has def ports)
= P (if P has no def ports)
External edges connected to PB1 = EB2 = UEU(PB1)
External ports connected to PB1 = PB2 = UPU(PB1)

Let CS be the sites connected to S through external edges (EB2). PB2 represents def ports in CS. Then,

CS = Site(PB2)
Internal edges in CS connected to PB2 = EB3 = UED(PB2)
Internal ports in CS connected to PB2 = PB3 = UPD(PB2)

External edges connected to CS = EB4 = UEU(PB3)
External ports connected to CS = PB4 = UPU(PB3)

The site connected to CS is Site(PB4). Continuing further, we have:

EB5 = UED(PB4)
PB5 = UPD(PB4)
EB6 = UEU(PB5)
PB6 = UPU(PB5), and so on.

A backward slice (BS) on a port (or variable) is all the edges (BSE), external edges (BSEE), internal edges (BSEI), and ports (BSP) that are reached by traversing backward from the port. Formally,

$BS = Site(PB1) \cup Site(PB2) \cup Site(PB3) \cup Site(PB4) \ldots$
$BSP = PB1 \cup PB2 \cup PB3 \cup PB4 \ldots$
$BSE = EB1 \cup EB2 \cup EB3 \cup EB4 \cup EB5 \cup \ldots$
$BSEI = EB1 \cup EB3 \cup EB5 \cup EB7 \cup EB9 \cup \ldots$
$BSEE = EB2 \cup EB4 \cup EB6 \cup EB8 \cup EB10 \cup \ldots$

Finally, a modular slice is computed as follows:

$Sites = FS \cap BS$
edges in the slice = $FSE \cap BSE$, and
$ports = FSP \cap BSP$.

5.2.2 RSG for slicing (RSGS)

The sites with high cohesion are merged in the StDG. This merging is achieved through structure compaction and data compaction. We present three types of RSGs for slicing from which the user can select the one best suited to the application. These RSGs differ in the amount of dependence information carried by the merged sites. Slicing criterion using RSG as a graph is:

$source = \{(v_{t1}, s1, Use) \mid s1{:}SITE\bullet$
$\quad v_{s1} \in Uses(s1) \wedge v \in ProgramVariable \wedge t1 \in ProgramStatement \}$
$sink = \{(w_{t2}, s2, Definition) \mid s2{:}SITE\bullet$
$\quad w_{t2} \in Defs(s2) \wedge w \in ProgramVariable \wedge t2 \in ProgramStatement\}$

Statements represented by a site (S) are:

$$\text{Statement}(s1) = \{k \mid \forall k \bullet k \in \text{StatementNumber} \wedge ((v_k, s1, \text{Use}) \in \text{UPS}(s1) \vee (v_k, s1, \text{Definition}) \in \text{DPS}(s1))\}$$

In the StDG, a slice is obtained by traversing and adding each port and edge in the graph. In the RSG, a slice is obtained by traversing and adding each site in the graph, where a site represents several statements. Several variants of slice definitions are available in the literature. One of them is a decomposition slice [Gall 91] that computes slices with respect to the last statement of a procedure. For decomposition slices, type 3 RSGS is best suited. Another variant of a slice is based on value dependence graphs [Weis 94] that allow a slice to be computed on a used or defined variables, including temporary variables (result of conditional expressions); for these type of slices, type 1 RSGS is best suited. Type 2 RSGS is best suited for Weiser's slice [Weis 84]. We now present three RSGS types.

5.2.2.1 Type 1 RSGS

This RSGS includes all the dependence information (all ports and edges) from the StDG. Data and structure compaction is used for merging sites. When a source (S_2) is moved to a destination (S_1) site, the use and def ports are moved to use and def ports of the destination, respectively. Ports and edges of S_1 are moved to S_2 as follows:

Let $A = \text{UPS}(S_2)$
$B = \text{DPS}(S_2)$
$C =$ ports A after moved to S_1
$D =$ ports B after moved to S_1
$S_1 \leftarrow S_2 = \{\text{UPS}(S_1) \leftarrow A \wedge \text{UEU}(C) \leftarrow \text{UEU}(A) \wedge \text{DEU}(C) \leftarrow \text{DEU}(A) \wedge \text{DPS}(S_1) \leftarrow B \wedge \text{UED}(D) \leftarrow \text{UED}(B) \wedge \text{DEU}(D) \leftarrow \text{DEU}(B)\}$

Sites in the type 1 RSGS of the StDG in figure 3.1(b) are shown in figure 5.1. In the figure, for clarity the ports and edges in the grouped sites are not merged. The dotted boxes represent the three sites in the RSGS.

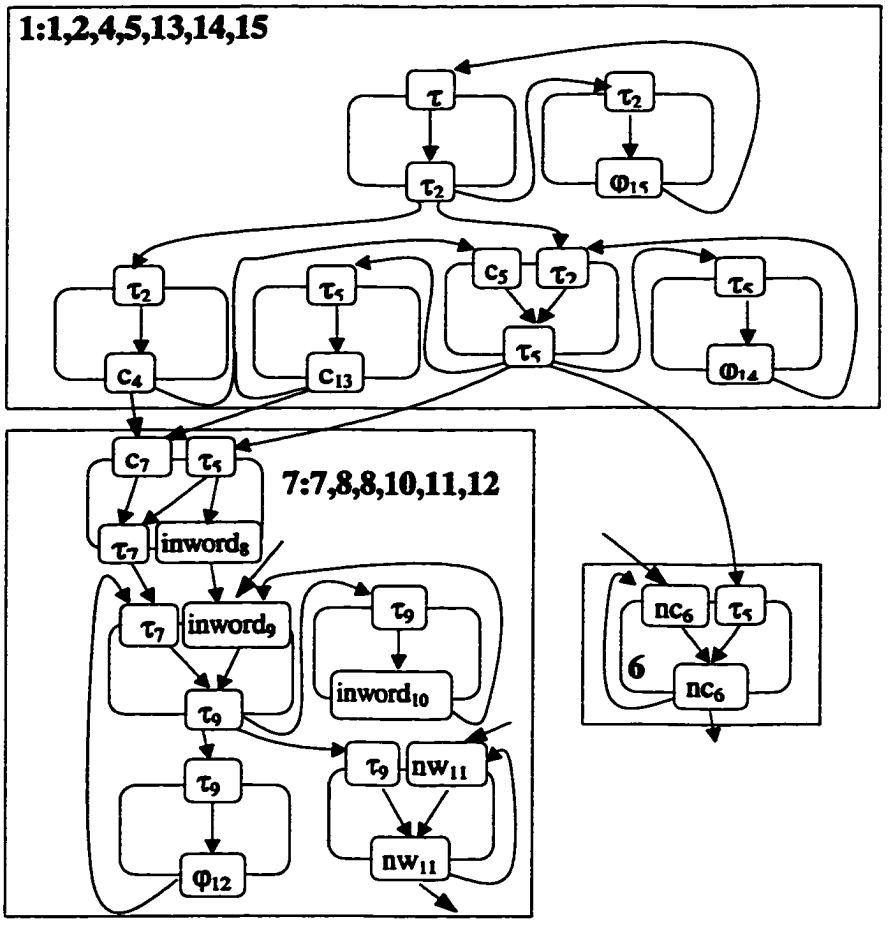


Figure 5.1. Type 1 RSGS

To compute a slice using the type 1 RSGS, only the ports and edges in source sites, sink sites, and multi-def sites are individually tested and added to the slice. The rest of the graph is tested at the site level, and all the ports and edges in a chosen site are added to the slice. For a review of the notation, see Section 5.2.1. A forward slice, using the type 1 RSGS, is computed as follows:

P = slice criteria (ports in the source)

S = Site(P)
EF1 = DEU(P) or Null – source edges
PF1 = DPU(P) or P – source ports
SF1 = Site(PF1) – source sites
EF2 = DED(PF1) – external edges of source
PF2 = DPD(PF1) – external ports of source
SF2 = Site(PF2) – sites connected to source
EF3 = DEU(SF2) – internal edges in SF2
EF4 = DED(SF2) – external edges to SF2
PF3 = DPD(SF2) – ports connected to SF2
SF3 = Site(PF3) – all sites connected to SF2

EF5 = DEU(SF3) – internal edges in SF2
EF6 = DED(SF3) – external edges to SF2

A forward slice using the above definitions is computed as in section 5.2.1. A backward slice is computed as follows:

P = slice criteria (ports in the sink)
S = Site(P)
EB1 = UED(P) – sink edges
PB1 = UPD(P) – sink ports
SB1 = Site(PB1) – sink sites
EB2 = UEU(PB1) – external edges of sink
PB2 = UPU(PB1) – external ports
SB2 = Site(PB2) – sites connected to sink
EB3 = UED(SB2) – internal edges in SB2
EB4 = UEU(SB2) – external edges to SB2

PB3 = UPU(SB2) – ports connected to SB2
SB3 = Site(PB3) – all sites connected to SB2
EB5 = UED(SB3) – internal edges in SB2
EB6 = UEU(SB3) – external edges to SB2

The last four steps are repeated until all the sites are exhausted. A backward or modular slice using the above definitions is computed as explained in Section 5.2.1.

5.2.2.2 Type 2 RSGS

The sites are merged through structure compaction and case 2 of data compaction. Sites are merged as explained in Chapter 4, depending on the site

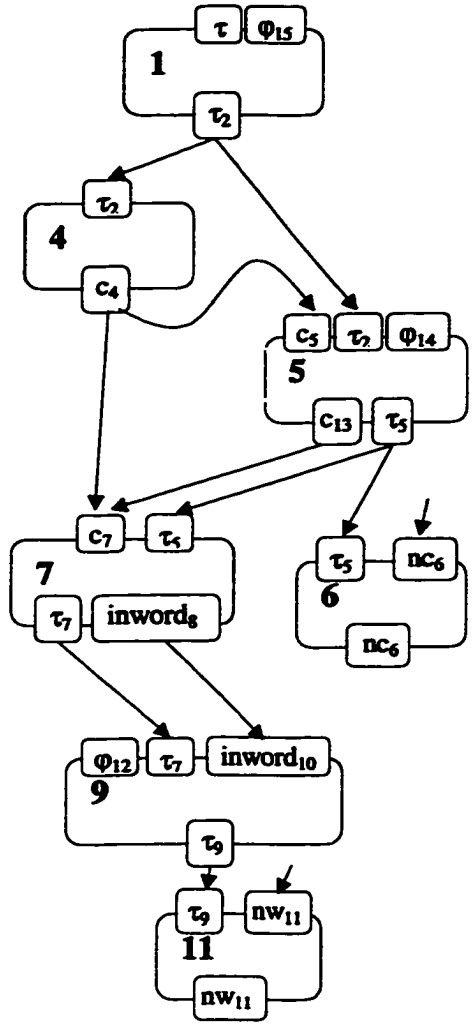


Figure 5.2. Type 2 RSGS

connections. Internal edges and external edges connecting ports within the merged site are not necessary and can be discarded. The sites in type 2 RSGS of the StDG in figure 3.1(b) are shown in figure 5.2.

Each site in the graph is analyzed as a unit and is either included or excluded from the slice.

SF1 = Site(P) – source sites
SF2 = Site(DPD(SF1)) – sites connected to source
SF3 = Site(DPD(SF2)), and so on.
SB1 = Site(P) – sink sites
SB2 = Site(UPU(SB1)) – external edges of sink

SB3 = Site(UPU(SB2) , and so on.
Modular slice = ((SF1 ∪ SF2 ∪ SF3 ∪ . . .) ∩ (SB1 ∪ SB2 ∪ SB3 ∪ . . .)

5.2.2.3 Type 3 RSGS

Sites are merged using data compaction and structure compaction. All internal details of the merged sites are discarded. Only the ports exposed to other sites are included in the merged sites. When a slice originates or ends within a merged site, the slicer must derive on demand the internal dependences of the group of statements in the merged site. This graph is simple and sufficient for most practical purposes. This program analysis approach is a hybrid of the demand driven and exhaustive approaches

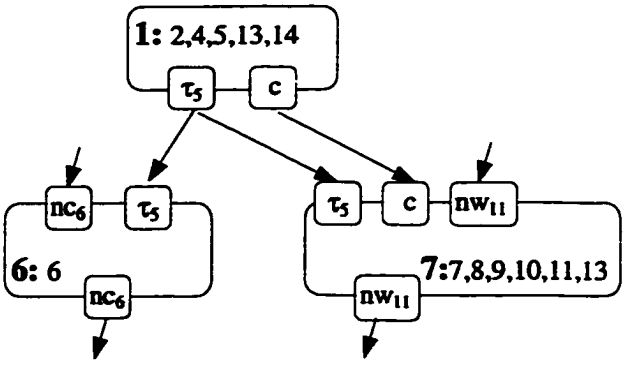


Figure 5.3. Type 3 RSGS

to. Figure 5.3 is a type 3 RSGS of the StDG shown in figure 3.1(b). The source and sink sites are sliced as explained for Type 1, and the rest of the sites are sliced as explained for Type 2.

5.2.3 Modular slicing

A traditional PDG-based slice includes all the sites that contribute to the variables in a set V just before the execution of a statement *p*. Consider, for example, a slice on nw at site 11 (figure 3.1(b), StDG).

Source = {(v_s,s,Use)| ∀s• s:SITE} (all use ports), Sink = (nw_{11},11,Definition)

The statements in the slice are:

```
1  void wordCount(int inword)
2  {
4    c = getchar ();
5    while ( c != EOF ) {
7        if ( c == ' '||c == '\n'||c == '\t' )
8            inword = 0;
9        else if ( inword == 0 ) {
10           inword = 1;
11           nw = nw + 1;
12       }
13    c = getchar ();
14    }
15 }
```

Using Type 3 RSGS, the slice includes sites 1 and 7 (statements in sites 1 and 7 are the same as above). Similar results can be obtained using any type of RSGS. Consider another slice on the use of variable `inword` at site 9 (note that many PDG-based slicers do not differentiate between a use and def at a site). The slice includes the following statements (Source = $\{(v_s,s,Use)| \forall s\bullet\ s:SITE\}$ (the entire program), Sink = $(inword_9,9,Use)$.

```
1  void wordCount(int inword)
2  {
4    c = getchar ();
5    while ( c != EOF ) {
7        if ( c == ' '||c == '\n'||c == '\t' )
8            inword = 0;
9        else if ( inword == 0 ) {
10           inword = 1;
12       }
13    c = getchar ();
14    }
15 }
```

We can obtain the same results using Types 1 and 2. To use type 3, we need to derive dependencies among statements 7-12 before slicing. A forward slice on uses and defs of variable at site *i*:

Source = DPS(i), Sink = $\{(v_s,s,Definition)| \forall s\bullet\ s:SITE\}$ (all def ports)

The advantage of RSGS can be seen when the user selects a set of variable use sites (source) and a set of variable def sites (sink). The slice includes all the statements required to compute the variables in the sink using the variables in the source. Consider a slice with nw (site 11) as sink and c as source. The resulting slice should not include any def sites of c or any sites in the slice of c at its def port. That is,

Source = {(c_i,s,Use)|∀s• s:SITE;∀i• i:ProgramStatement},
Sink= (nw_{11},11,Definition)

The statements in the slice are:

```
7          if ( c == ' '||c == '\n'||c == '\t' )
8              inword = 0;
9          else if ( inword == 0 ) {
10             inword = 1;
11             nw = nw + 1;
12     }
```

Table 5.1. Comparison of slicing methods

	PDG	Type 1	Type 2	Type 3
Slice type	Forward or backward	Forward, backward, and modular	Forward, backward, and modular	Forward, backward, and modular
Application	General slicing	General slicing, extraction, and understanding	function extraction and understanding	Program understanding
Graph size	One node per expression	One site per several statements. Several nodes (ports) in a site. Graph bigger than PDG.	One site per several statements. Smaller than PDG.	One site per several statements. Smallest graph.
Slice time	Linear to size times 2	Linear to size plus size of source or/and sink sites and multi-def sites encountered.	Linear to size plus size of source or/and sink sites and multi-def sites encountered.	Linear to size

5.2.4 Comparison of slicing techniques

PDG based slicing and RSG based slicing are compared in table 5.1. In section 5.3, we show the application of RSG to maintenance activities.

5.3 Maintenance

Understanding the system, incorporating the change, and testing the system to ensure that the change had no unintended effect on the system are the three facets of software maintenance. Generally, two approaches are followed in dealing with the latter two facets. One approach is to allow the maintainer to implement the change, and then provide a tool that will pinpoint any inconsistencies introduced due to change [Gris 95]. The second approach is to "provide the maintainer with a semantically constrained problem and let him construct the solution which implements the change within these constraints" [Gall 91][Morg 97].

The former approach allows the maintainer more freedom, but at a cost. If the tool finds inconsistencies, the changes need to be rolled back. Opdyke [Opdy 92] suggests that we make changes to a copy of the system. If inconsistencies are found after the change, the earlier version can be used and the current version discarded. Moreover, the problem of finding inconsistencies is found to be NP-hard [Gall 91]. The latter approach, though restrictive, uncovers inconsistencies before the changes are incorporated. Gallaghar et al. claim that the benefits outweigh the inconvenience that may be encountered due to the imposition of the constraints [Gall 91].

Representations generally available are used only to reason about the correctness of program changes. When changes are made to the program, its representation must

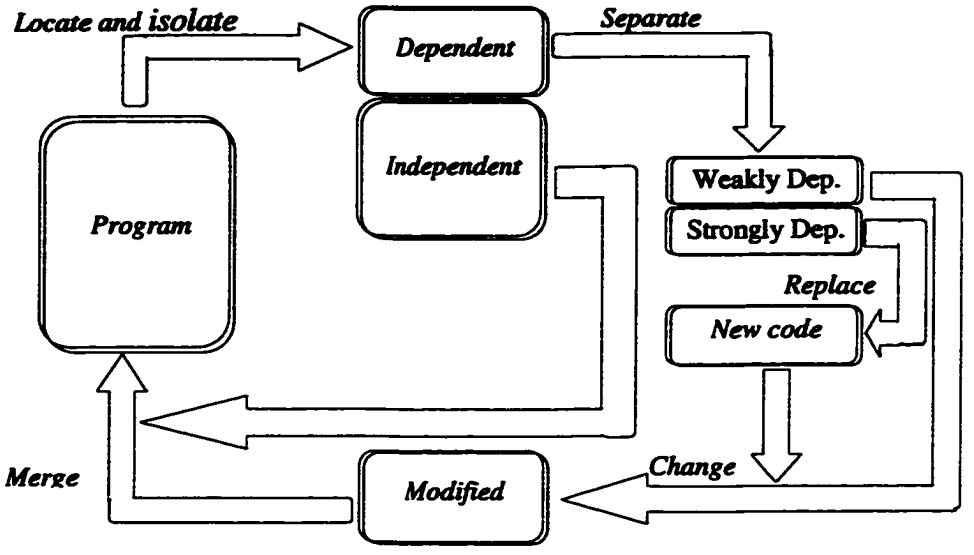


Figure 5.4. The maintenance model

also be updated, as maintenance is a continuous process. However, the incorporation of the changes into the programs and subsequently into the representation has not been adequately addressed. The representations generally must be re-derived from the program when the program is changed. The RSG based maintenance model is a semantically constrained maintenance process that allows simultaneous updates to both the representation model and the program.

5.3.1 Maintenance model

The RSG based maintenance model is a five-step process, as shown in figure 5.4.

i. **Understand and locate.** Understand and locate the program location where the change will be introduced. This location may be a procedure or a block within the procedure or a member of a block.

ii. **Isolate.** Identify and isolate the change dependent statements from the change independent statements. Change independent statements are that part of the program that do not reference any variable affected due to the change. This isolation allows the maintainer to modify the program freely, considering only the dependent part and not the whole program. An optimal situation is one where we can identify and isolate the absolute minimum amount of code that will be affected by the change.

iii. **Separate.** Separate the dependent statements into strongly dependent and weakly dependent parts. We consider the statements that will be replaced due to the change as strongly dependent on the change and the statements that can be reused as weakly dependent. For example, consider a modification requiring change in the number of iterations of a loop. Step 2 identifies the entire loop block as change dependent. But, the loop members may not require any change. Hence we consider the loop statement as strongly dependent and loop members as weekly dependent.

iv. **Change.** Introduce the necessary changes. This step involves replacing the strongly dependent statements, while generally reusing the weakly dependent statements.

v. **Merge.** Merge the independent and the modified parts. Make sure that any new names introduced in the changed part do not conflict with any names in the independent part of the program.

Steps 2 and 3 break the program into manageable pieces, and automatically assist the maintainer in ensuring that there are no ripple effects induced by

modifications in the change dependent part. In the next section we show how the StDG can be restructured for maintenance activities.

5.3.2 RSG for maintenance

In the StDG, variable definitions propagate from the definition sites to the reference sites through the def edges. To delete a site from the graph, we follow the def edges of the site and remove all the sites encountered. If one of the sites encountered is a member of a circularly connected chain of sites, we traverse and remove the entire chain. Similarly, to add new statements we traverse the entire chain of sites to analyze the effects of the additions. Moreover, it is easy to understand the code if the statements represented by circular chain of sites are present together. Hence, instead of having these sites as separate sites, we represent them all in one site in the graph because maintenance operations must consider them together.

A procedure in a program can be viewed at as a block. A block consists of statements and other blocks (members). In a block, all members are dependent (dependence includes data, control, and flow dependence) on the blockhead and, in turn, on all the members on which the blockhead depends. Block members on which the blockhead depends and the blockhead form a circular chain of sites. Hence, these members' sites can be merged into the blockhead's site. By this merging, a block is separated into a structure part and a data part. The structure part includes the blockhead and (see section 4.3.1) members (these may include members of a deeper block) in the block which define variables referenced by the blockhead. The data part consists of the remaining members of the block. Any changes made to the data part of a block will not

affect its structure part and any changes made to structure part affects the constituents of the data part equally.

Structure compaction is used to merge circularly connected sites. Figure 5.5 is a RSG for maintenance (RSGM) of the StDG shown in figure 3.1(a). In the next section, we show how the RSGM can be used for maintenance activities.

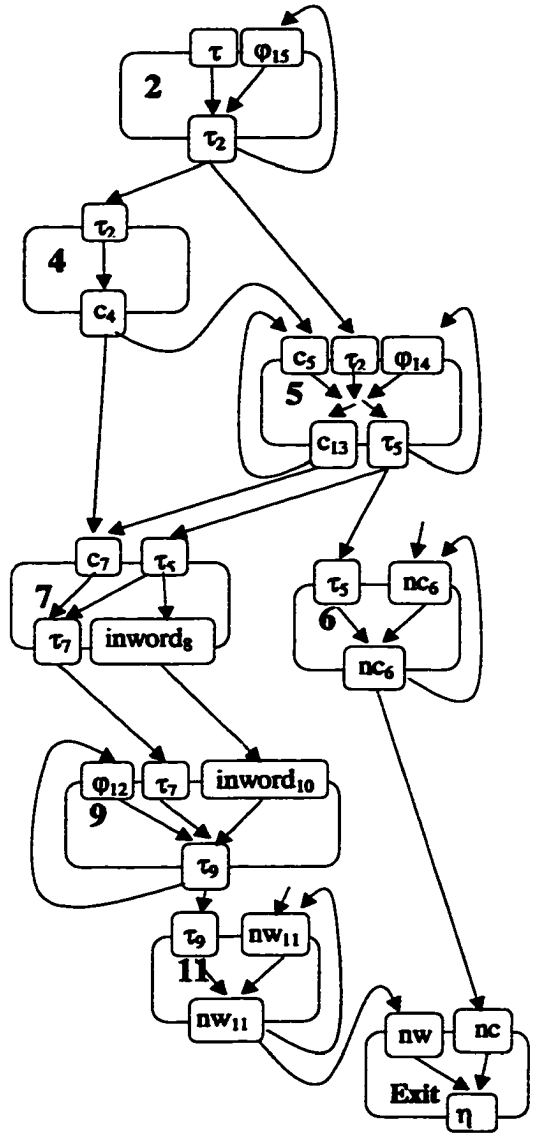


Figure 5.5. RSG for maintenance (RSGM)

5.3.3 Maintenance activities

The RSGM for a procedure is useful for the modifications such as addition, deletion, change, and code movement.

a. Additions. Addition involves adding new computation (statements) to a procedure. In the maintenance model, we skip to the final step, *merge*. Additions can be made if they do not result in the addition of new use edges to the sites of the original graph. There are two ways statements can be added to the program without affecting the program computation. First, statements that do not define any existing variables can be added. These statements can be placed in any block.

Second, statements that redefine existing variables can be added if the new definitions do not reach the statements in the original program. The new definitions will not reach the original program if the statements redefining an existing variable are placed between a last use statement and a new def statement of the variable. Or, the new statements can be added after the final-use statement of the redefined variable. We define graph parameters for the statements being added to the program as:

$AVariables = AProgramVariables \cup \{\tau, \lambda, \eta, \phi\}$
$AVar \subseteq \{v_j \mid v \in AVariables \wedge j \in AProgramStatements\}$
$ASITE$ = sites of the added statements

The conditions for addition are:

$\{\forall s{:}ASITE; \forall v{:}AVar; \forall v{:}Var; \forall e{:}EDGE; \forall i,j{:}ProgramStatements; s1,s2{:}SITE; \bullet$
$((v_s,s,Definition) \in DPS(s) \wedge v \notin \{\tau, \lambda, \eta, \phi\} \wedge v \notin Variable) \vee$
$((v_s,s,Definition) \in DPS(s) \wedge e \in ((v_i,s1,Definition),(v_j,s2,Use)) \wedge s < i \wedge s > j\}$

That is, a variable defined in the added statements is not present in the original program, or a variable defined in the added statement is not added between a definition statement and a use statement of the variable.

The addition of new statements in a block results in a def edges from the blockhead to added statements (control dependence edges). The statement number of the added statements should lie within the block begin and block end site numbers (i.e. between τ and φ port numbers of the blockhead). If the added statements use any variables from the original program, additional def edges (data dependence edges) will result. Again, the statement number of the added statement should be greater than the statements (number) containing the variable references in the added statement.

Three cases arise when merging new code:

i. If an addition results in no new data dependence edges from the original graph, then new statements are added in the chosen block. They become control dependent on the current block's blockhead (b). For each statement in the added statements (AS), add a control edge from b. That is,

$\{\forall j$:ASITE;G:RSG• $G \leftarrow ((\tau_b,b,\text{Definition}),(\tau_j,j,\text{Use}))\}$

ii. If the new statements use a variable defined both inside (v_i) and outside (v_k) the block, then new statements are added before the site in which the used variable (v_j) is defined within the block. For each definition of v reaching AS from outside the block, external edges are added. An external edge is also added for each definition of v defined within the block which reaches AS. That is,

$\{\forall v_j; \forall j$:ASITE;$\forall k$:SITE;i,s1,s2:SITE;G:RSG•
$G \leftarrow ((v_k,s1,\text{Definition}),(v_j,j,\text{Use})) \wedge$
$G \leftarrow ((v_i,s2,\text{Definition}),(v_j,j,\text{Use})) \wedge j<i \wedge v_k \in \text{Defs}(s1) \wedge v_i \in \text{Defs}(s2)\}$

iii. If the new statements use a variable defined only within the block, then the new statements are added after the site in which the used variable is defined. An

external edge is added from the definition of v within the block (v_i) to the use of v in AS (v_j). That is,

$\{\forall v_j,j:ASITE;i,s2:SITE;G:RSG\bullet$
$G \leftarrow ((v_i,s2,Definition),(v_j,j,Use)) \wedge j>i \wedge v_i \in Defs(s2)\}$

In the merged program the ascending order of the statement numbers is maintained.

b. Deletions. Any leaf site can be removed. A leaf site has no def edges (i.e., no other site is using it); hence, it can be removed without affecting the rest of the program. Any site in the graph can be made a leaf site by merging all the sites in the forward slice of the site (FSS), as explained in Section 5.2.1. If *r* is the site that needs to be deleted, then all the sites in the forward slice of *r* are merged in *r*. Then *r* is removed. That is,

$\{\forall s:SITE\bullet\ s \in FSS(UPS(r)) \wedge r \leftarrow s \wedge \leftarrow r\}$

In the final program the ascending order of the statement numbers should be maintained. Care should be taken when sites are removed to avoid dead code. A non-final-use site without a def port represents dead code and can be removed. That is,

$\{\exists s: SITE\bullet\ DED(s) = \phi \wedge \leftarrow n\}$

c. Changes. The change may be viewed as a deletion followed by addition. We show how the changes can be incorporated in the RSGM of the StDG of figure 3.1(b) using the maintenance model. Statement 6 in figure 3.1(a) computes the number of characters in the input stream. To change this statement to count only the non-blank characters, the statement should be removed and the following statements added.

```
a1. if( c != ' ' )
a2.     nc = nc + 1;
```

Next we use the 5-step maintenance process to incorporate the change.

i. Locate the sites in the restructured graph that needs to be changed or removed. In our example, site 6 needs changes. Site 6 is a member of block 5.

ii. The change dependent part includes the graph slices of all the sites identified in

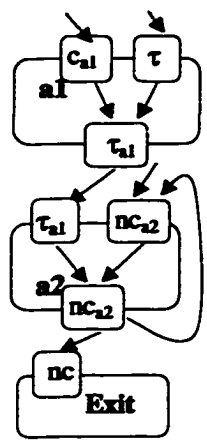


Figure 5.6. The StDG of the changes.

the first step. The graph slice of site 6 includes site 6 and 'nc' port of exit site (sites r1 and Exit in figure 5.6). The rest of the graph is change independent.

iii. Separate the dependent part into reusable and replaceable parts by identifying sub-graph slices from the change dependent part identified in step 2. The sub-graph slices may be reused. The entire graph slice identified in step 2 can be reused.

iv. Change involves addition of a new site a1 and reuse of slice identified in step 3, as shown in figure 5.6.

v. Merge the change independent part of step 2 and modified part from step 4. This process involves adding the modified part of the StDG to the restructured graph with the change dependent part removed. The modified part is to be added in block 5; hence the modified part can be placed between statements 5 and 14.

The modified part uses variable 'c' defined both outside (statement 4) and inside the block (statement 13), hence it should be placed before statement 13. Merging results in three data dependence edges from c_4, c_{13}, and nc, and one control dependence edge from τ_5, as shown in figure 5.7 (dotted lines). The modified program RSG is as shown in figure 5.7.

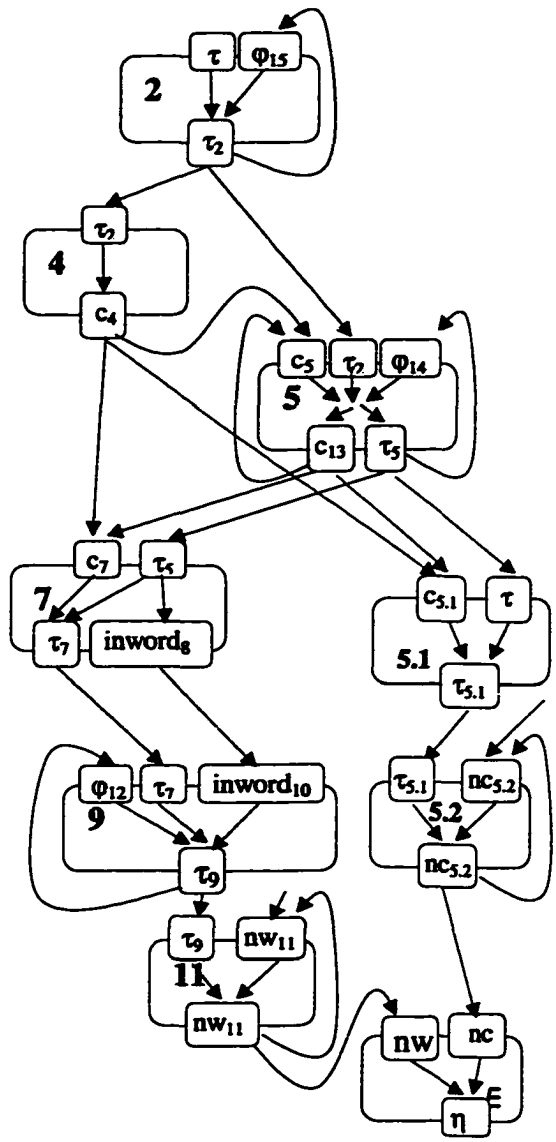


Figure 5.7. The modified program RSG

d. Code movement. A statement redefining an existing variable cannot be added between the variable definition and use statements in the original program. In a typical program, any number of statements may be present between the variable definition and use statements. By moving the variable definition statement closer to the use statement we can provide more space for adding new statements. Moreover, code movement increases code readability. Code movement is widely addresses as part of program restructuring transformations [Morg 97][Gris 95][Bowi 95].

In the RSG, code movement can be accomplished easily. A statement can only be moved within its block. We consider two scenarios before moving the code.

i. If a definition statement and a use statement of the variable are present in the same block, then the definition statement can be placed before the use statement. If more than one use statement is present, then the statement is moved to the use statement closest to it.

ii. If the definition statement and the use statement of the variable are present in different blocks, then the definition statement can be placed before the blockhead statement of which the use statement is a member. If the block of which the use statement is a member and the definition statement are members of different blocks (i.e. the use statement is in a deeper block), then we find the block of which the previous block is a member. This process is done repeatedly until we find a block that is a member of the same block as the definition statement.

Let s1 be the set of sites in the graph where a set of variables is defined. Then s1 can be moved near s2, where s2 = min Site(DPD(s1)). If s1 < s2, then s1 is moved to a

site that is less than s1. If s1 > s2 then s1 is moved to a site that is greater than s1. If s1 and s2 are in different blocks, then either s1 can be moved in its block or the entire block of s1 can be moved together.

The maintenance model is similar to the one presented in [Gall 91], but the approaches to implement the models differ. [Gall 91] defines a decomposition slice, which is obtained using union and complements of traditional slices to isolate the code that needs change. The decomposition slice has worst-case times of $O(n\ e\ \log(e))$, where n is the number of variables and e is the number of edges in the flowgraph [Gall 91]. The graph slice can be computed from the RSG in time proportional to the size of the graph. Moreover, formulation of the decomposition slice requires a better understanding of the program.

5.4 Summary

In this chapter, we applied RSG to program slicing and maintenance. A group of sites with high cohesion will be present (or not present) in every slice, unless the slice starts or ends in the merged site. The StDG is restructured through compaction that merges the sites with high cohesion. The type of restructuring required changes from application to application. We introduced three types of RSGS for slicing. Depending on the time and space requirements, the software engineer can select the type best suited to the application. We also showed how forward, backward, and modular slices can be computed using different types of RSGS.

Generally two approaches are followed when incorporating changes in a program. One approach allows changes to be made without any constraints and then checks for inconsistencies introduced. Finding inconsistencies after the changes are

made was found to be NP-hard. The second approach constraints the type of changes that can be made. We present a maintenance process that uses the latter approach. We show how the restructured graph for maintenance (RSGM) is used in activities like addition, deletion, changes, and code movement. The restructured graph (RSG) yields better results than the generally used graphs when used for slicing and maintenance.

Chapter 6

Reverse Engineering

6.1 Introduction

The term "reverse engineering" has its roots in the hardware world where the primary objective is to decipher products whose design is not available. In software engineering, the term is used to describe the process of examining a software system to aid maintenance, gain insight, and enhance overall understandability [Chik 90]. The central theme of reverse engineering research involves the development of tools, techniques, and methodologies for the analysis, synthesis, and representation of information about existing software systems. The reverse engineering activities can be broadly identified as 1) identifying the functionality of an existing system, 2) modeling it at a physical (or design) level, and then 3) modeling at a logical (or analysis) level [Scha 96].

This research is applicable to the first two activities; for this purpose we use a language independent format (LIF) and the statement dependence graph (StDG) representations. The LIF captures the details of the program from which the StDG and other design views of the program are derived. In section 6.2, we examine the LIF representation. In section 6.3 we present a synthesis of the StDG from the LIF. Various views of the reverse engineered design are presented in section 6.4. The chapter summary is presented in section 6.5.

6.2 Code to LIF

The language independent format was developed as part of the *Unravel* [Lyle 95] project, developed at the National Institute of Standards and Technology (NIST). *Unravel* is a Computer Aided Software Engineering (CASE) tool that can be used to statically evaluate ANSI C source code using program slicing. For *unravel*, the LIF is an intermediate step in obtaining the PDG of a program; in this work we use it to derive the StDG. Sections 6.2.1 and 6.2.2.present the semantics of the LIF.

6.2.1 Language independent format (LIF)

The LIF represents the program as an annotated flow graph of nodes and edges. Nodes are generated to represent semantic or syntactic units of the program that correspond to statements or parts of statements. A flow graph consists of two edge types: control flow and requires. A control flow edge between two nodes indicates the flow of control from one node to the other. A requires edge is a general mechanism for specifying control or syntactic dependence between nodes. A node may have one or more requires nodes; these nodes are known as the requires set. The annotations specify location of the corresponding source code.

The rules for representing statements as flow-graph nodes and for specifying requires sets are as follows:

1. An expression is represented as a dataflow node.
2. A statement that is composed of noncontiguous tokens is divided into two or more dataflow nodes such that each group of contiguous tokens is one or more nodes. Examples are the matching braces of a compound statement and the do ... while.

3. An additional dataflow node is used to represent each C prefix (++x), postfix (x++) or comma (x+y, z) operator in an expression. A conditional operator uses three additional dataflow nodes.
4. Any compound statement that is represented with more than one dataflow node has one node designated for inclusion in requires sets. Any node controlled by the compound statement references the designated node in its requires set. The other nodes of the compound statement are referenced in requires set of the designated node.
5. Each flow-graph node is annotated to provide a mapping from flow-graph nodes to source code statements.
6. The compound statement generates one flow-graph node for the beginning bracket and another for the ending bracket.

The LIF handles the following language features:

- Expression statements
- Compound control statements
- Structure variables
- Indirect assignment by pointer
- Indirect reference by pointer
- Dynamic structures
- References to structure members by pointer
- Assignment to structure members by pointer
- Procedure calls

6.2.2 Language independent representation

In this section, we present the details of the LIF codes used for the representation. The codes used to specify declarations and expressions are presented in table 6.1. Declarations do not generate a flow-graph node. They generate a positive id

for each variable, and each global variable is allocated a unique id. Each procedure has a separate set of ids for local variables and formal parameters, starting from 1. The variable attributes, static, pointer, external, and array are indicated by the codes: S, P, X and A.

Table 6.1. LIF codes used to specify declarations and expressions

Type	Code	Syntax
LOCAL ID	4	4(id, name[,S][,P][,X][,A])
GLOBAL ID	5	5(id, name[,S][,P][,X][,A])
REF	7	7(node, id[,level])
DEF	8	8(node, id[,level])
GREF	9	9(node, id[,level])
GDEF	10	10(node, id[,level])
AREF	24	24(node, address)
ADDRESS	25	25(address, procedure id, id)

Expressions generate codes for variables referenced and defined. REF code is used for local variables whose values are used. GREF code is used for global variables whose values are used. Similarly, DEF and GDEF codes are used for local and global variables that are assigned new values. The level indicates the level of indirection of the ref or def. A level of zero, which represents no indirection, is omitted. A level of -1 indicates the *address of* operator (&). ADDRESS is used for each object of the *address of* operator, indicating the variable, the procedure where the variable is declared (zero for global declaration) and a unique address id. Address ids are assigned sequentially from 1.

The codes used to specify the flow-graph are presented in table 6.2. Each flow-graph node produced is annotated by SOURCE MAP to provide a mapping from flow-graph nodes to source code statements. Flow of control from node-to-node is specified

with SUCC. An `if` statement without an `else` generates at least two nodes: one node for the `if`, left parenthesis token and the condition expression, and one for the right parenthesis to serve as an exit point from the statement. The nodes for the controlled statement must exit through the right parenthesis node. The controlled statement generates a REQUIRES entry for the `if` node. The `if` node requires the parenthesis node. An `if` statement with an `else` generates an additional node for the `else`. Nodes of the second controlled statement require the `else` node. The `else` node requires the `if` node. The flow-graph of an `if` statement is presented in Figure 6.1.

Table 6.2. LIF codes used to specify the flow-graph

Type	Code	Syntax
RETURN	14	14(node,1|0)
GOTO	15	15(node, G|B|C)
SUCC	16	16(from node, to node)
REQUIRES	17	17(node, required node)
SOURCE MAP	18	18(node, from line, from column, to line, to column)

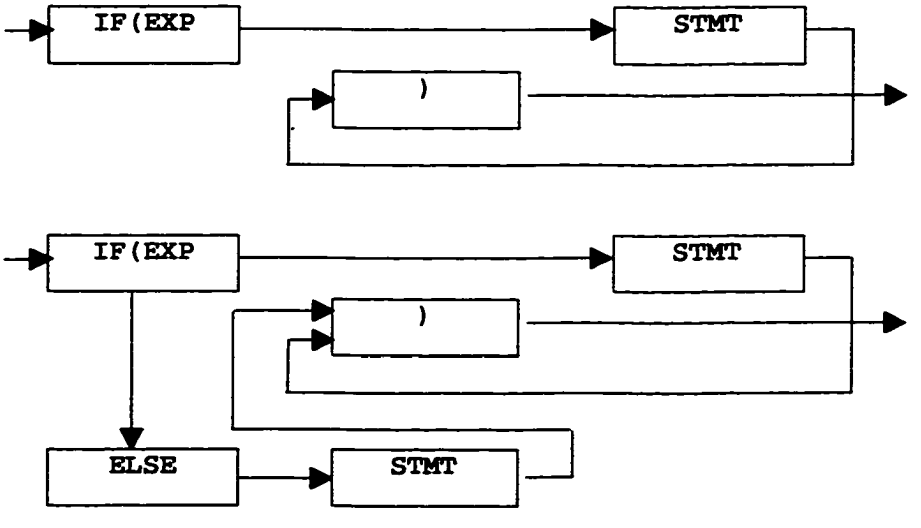


Figure 6.1. `if` statement flow-graph

A `switch` statement generates two nodes, one for the `switch` token and expression and one for the right parenthesis token. The right parenthesis token is used as an exit point for each case in the controlled statement. The controlled statement generates a REQUIRES entry for the `switch` node. A `while` statement generates two nodes, one for the `while`, left parenthesis and expression and one for the right parenthesis. The right parenthesis node is a successor (SUCC) to the `while` node and the last node of the controlled statement. The controlled statement generates a REQUIRES entry for the `while` node. The `while` node requires the right parenthesis node. The `do . . . while` generates three nodes: the `do`, the `while` and condition, and the right parenthesis. The successor of the `do` node is the first node of the controlled statement. The `while` node is the successor of the last node of the controlled statement. The `while` node has two successors: the `do` node and the right parenthesis. The `do` node is required by the controlled statement, and the `do` node requires the `while` node and the right parenthesis. The `for` statement generates three nodes. The first node contains the `for`, left parenthesis, and the initialization. The second node encompasses the test expression, and the third node contains the increment and the right parenthesis. The test is a successor of the `for` and initialization. The statement is a successor of the test, and the increment is a successor of the statement. The `for` and the initialization expression require the test, the increment, and the statement. The statement and increment are both required by the test, and the right parenthesis requires the `for`. Nodes corresponding to `return` statements are identified by RETURN. The second field of the RETURN indicates a return with expression by 1 and a return without

expression by 0. The statements `goto`, `break` and `continue` are identified by a corresponding G, B or C code in a GOTO entry.

Procedure headers and calls use the codes presented in table 6.3. The PROC END indicates a static declared procedure with an S flag. Procedures that return an expression are indicated with an R flag. For formal parameters, the variable attributes pointer and array are indicated by the code: P and A in the FORMAL ID record. All local variable declarations, (LOCAL ID), and flow graph node related LIF codes appear between the PROC START and the PROC END.

Table 6.3. LIF codes to specify procedures

Type	Code	Syntax
PROC START	1	1(node, procedure id, name)
PROC END	2	2(node[,S][,R])
FORMAL ID	3	3(id, name[,A][,P])
CALL START	11	11(node, procedure id)
ACTUAL SEP	12	12
CALL END	13	13

Procedure calls are handled using CALL START, and the actual parameters are listed as expressions in order separated by ACTUAL SEP entries. Structure fields are represented using the codes presented in table 6.4.

Table 6.4. LIF codes to specify structure fields

Type	Code	Syntax
CHAIN	19	19(node, chain, id)
GCHAIN	20	20(node, chain, id)
FIELD	21	21(node, chain, seq, field id, field)
CREF	22	22(node, chain)
CDEF	23	23(node, chain)
STRUCT	26	26(procedure id, id, offset)

At an expression node, each reference or assignment through a pointer to the fields of a structure generates a chain (CHAIN or GCHAIN). The chains of a node are given a chain number sequentially from 1. The variable at the head of the chain is specified in the id field of the CHAIN for local variables and in the id field of the GCHAIN for global variables. CREF and CDEF indicate if the chain specifies a ref or a def. FIELD is used to specify each field of a chain by sequence number. The field id is the sequence number of the field within the data structure and field is the field name. STRUCT indicates that the variable identified by the procedure id, and id is a structure with offset members. LIF code FILE, *6(file id, file name)* is used to indicate the source file associated with each procedure. Derivation of the StDG from the LIF codes is presented in the next section.

6.3 LIF to StDG

The source code for deriving the StDG has one statement per line. Recall, the statement terminators in the StDG are: ';', '{', '}' (an exception is a `do . . . while` loop, '`}while(expression);`' is placed in one line) , and ')', only if ')' is not followed by a '{'. Each statement in the source is given a sequential number starting from 1. Figure 3.1(a) is a sample program with statement numbers. In the StDG the nodes are represented as sites. The sites representation includes use/def ports, use/def/internal edges, four special ports (τ, ϕ, λ, and η), and three special sites for each procedure (enter, exit, and summary). The aim is to convert the control flow (SUCC) and control dependence (REQUIRES) information from the LIF to data, control, and flow dependences while at the same time determining the internal data dependences at

each node (which variables are used to define a particular variable is lost in LIF). Codes similar to the LIF codes are used to describe the StDG. These codes are shown in table 6.5.

Table 6.5. StDG codes

Type	Code	Syntax
LOCAL ID	4	4(id, name[,S][,P][,X][,A])
GLOBAL ID	5	5(id, name[,S][,P][,X][,A])
FORMAL ID	3	3(id, name[,A][,P])
USE PORT	31	31(site #, id, level, G|L|P, port #)
DEF PORT	32	32(site #, id, level, G|L|P, port #)
INT EDGE	35	35(site #, id, level, G|L|P, port #, id, level, G|L|P, port #)
EXT EDGE	36	36(site #, id, level,G|L|P, port #,site #, id, level, G|L|P, port #)
SUMMARY	37	37(procedure id, site #, name)
FUN CALL	38	38(calling procedure id, called procedure id)

Ports (USE PORT and DEF PORT) are represented using the site number, variable id from the LIF, level of indirection, variable type (global, local, or parameter), and port number (which is the site number, initially). A level of zero indicates no indirection, and –1 indicates the *address of* operator (&). The external edges (EXT EDGE) at a site are specified using a from-port and a to-port. The same external edge acts as a def edge with respect to the first site # in the code and as a use with respect to the second site #. The internal edges (INT EDGE) are specified with site number (in which the edge is present) and two ports (from and to). The same ids used in LIF for local variables, global variables, formal parameters, procedures, and array variables are also used in the StDG. Each procedure's summary site is specified with SUMMARY, using the procedure id and a unique site number. A procedure call (FUN CALL) is identified with ids of calling and called procedures. Section 6.3.1 presents the derivation

process of StDG codes from the LIF. For variable declarations, the LIF codes (LOCAL ID, GLOBAL ID, and FORMAL ID) are also used as StDG codes.

6.3.1 Definitions

This section contains definitions relevant to the StDG extraction.

site($node_1$). Statement to which the $node_1$ belongs. It is found from the SOURCE MAP. For example, site($node_1$) = s_1 is found from LIF code *18($node_1$, s_1, . . .)*.

VT(id_1). Variable type of id_1, is a P (parameter) if there is a LIF code *3(id_2, name)* such that $id_1 \leq id_2$. Otherwise, it is an L (local variable).

CP(pid_1). Current procedure id and name of pid_1 found from *1(node, pid_1, name)*.

RV(id_1). Is id_1 used in a `return` statement? Yes, if there are pairs of LIF codes *24($node_1$,$addr_1$)* and *25($addr_1$,pid,id_1)*, *14($node_1$,1|0)* and *8($node_1$,id_1)*, or *14(node,1|0)* and *7($node_1$,id_1)*, tested in that order (in `return a = b+c;`, only `a` is used in the return statement).

6.3.2 Ports

This section presents the derivation of use ports of the StDG. Table 6.6 contains extractions of use ports from the LIF. The codes in the LIF column are used to derive ports as shown in use ports column, if the conditions mentioned in the same column are satisfied. The comments column indicates the type of variables.

All sites of statements using data variables have use ports. The LIF uses REF and DEF for both local variable and parameters. Hence, VT is used to identify a local variable from a parameter. Modified global variables and variables returned from a procedure are specified as use ports in $EXIT_{pid}$ site. Two different def nodes belonging

to the same statement indicates the presence of multiple definitions in the statement (multi-def variable). All block members in LIF require a blockhead. This feature is represented in the StDG with a control variable in all block members (items 8 and 9 in table 6.6). Table 6.7 shows how the def ports are obtained from the LIF codes.

Table 6.6. Determination of use ports from the LIF

#	LIF	Use ports	Comments
1	7(node, id[,level])	31(site(node), id, level, VT(id), site(node)) 31($EXIT_{pid}$, id, level, VT(id), 0), if RV(id)	Local variable
2	9(node,id[,level])	31(site(node), id, level, G, site(node))	Global variable
3	10(node, id[,level])	31($EXIT_{pid}$, id, level, G, 0)	Defined global variable
4	24($node_1$,$addr_1$) 25($addr_1$,pid,id_1)	31(site(node), id, -1, VT(id), site(node)), VT(id) = G, if pid=0 above. 31($EXIT_{pid}$, id, -1, VT(id), 0), if RV(id)	Address ref
5	8($node_1$, id_1) 8($node_2$, id_2)	If site($node_1$) = site($node_2$) then 31(site($node_1$), λ, 0, 0, site($node_1$))	Multi-def
6	10($node_1$, id_1) 10($node_2$, id_2)	If site($node_1$) = site($node_2$) then 31(site($node_1$), λ, 0, 0, site($node_1$))	Multi-def
7	1(node, pid, name)	31(site(node), τ, 0, 0, 0)	New procedure
8	17(node, mode)	If site(mode) > site(node) then 31(site(mode), τ, 0, 0, site(node))	Control variable
9	17(node, mode1, mode2)	31(site(mode), τ, 0, 0, site(node)), for each mode satisfying mode1 ≤ mode ≥ mode2 and site(mode) > site(node)	Control variable
10	17(node, mode)	If site(mode) < site(node) then 31(site(node), τ, 0, 0, site(node))	}, as a member of block

All sites of statements defining data variables have def ports (items 1-3 in the table). All parameters and global variables used in a procedure are identified in the $ENTER_{pid}$ site. The user has to specify if a final-use variable is present in a statement. In the LIF, a blockhead requires the '}' and all members (except '{') require the blockhead. This feature is exploited by item 7 in table 6.7 to find the blockhead's '}'

(blockhead statement number is the smallest of all members and, '}' has the highest number in its block). The next section presents the derivation of internal edges.

Table 6.7. Determination of def ports from the LIF

#	LIF	Def ports	Comments
1	8(node, id[,level])	32(site(node), id, level, VT(id), site(node))	Local variable
2	10(node, id[,level])	32(site(node), id, level, G, site(node)) 32($ENTER_{pid}$, id, 0, G, 0)	Global variable
3	3(id, name)	32($ENTER_{pid}$, id, 0, P, 0)	Parameter
4	User Defined	If user specifies that site is a final-use site 32(site), η, 0, 0, site($node_1$))	Final-use
5	1(node, pid, name)	32(site(node), τ, 0, 0, site(node))	New procedure
6	17(node, rnode)	If site(rnode) > site(node) then 32(site(node), τ, 0, 0, site(node))	Control variable
7	17(node, rnode)	If site(rnode) < site(node) then 32(site(node), ϕ, 0, 0, site(node))	Flow variable, }
8	11(node, pid)	If 7(node, . .) and 8(node, . .) are not present then 32(site(node), ϕ, 0, 0, site(node))	Function calls not using or defining a variable
9	15(node, B|C)	32(site(node), ϕ, 0, 0, site(node))	Break, continue

6.3.3 Internal edges

Table 6.8 shows how the internal edges of sites are obtained from the LIF. At each site, there are internal edges from τ use port to all def ports. There are edges from data variable use ports and the multi-def port to a data variable def port.

6.3.4 Control and flow dependence edges

Control dependence edges (external) and part of the flow dependence edges are obtained from the LIF. The remaining flow dependence edges must be provided by the user (example, dependence among two output statements). The data dependence edges

are derived from the already built StDG and the LIF using data flow analysis. Table 6.9 gives the control and flow dependence edges obtained from the LIF. An external edge is a use edge of one site and also a def edge of another site.

Table 6.8. Determination of internal edges from the LIF

#	LIF	Internal edges	Comments
1	7(node, id_1[,level]) 8(node, id_2[,level])	35(site(node), id_1, level, VT(id_1), site(node),id_2, level, VT(id_2), site(node))	Use and def of local variables
2	9(node, id_1[,level]) 10(node,id_2[,level])	35(site(node), id_1, level, VT(id_1), site(node), id_2, level, G, site(node))	Global variables
3	8(node, id_2[,level]) 17(rnode, node)	If site(node) > site(rnode) then 35(site(node), τ, 0, 0, site(rnode), id_2,level, VT(id_2), site(node))	τ use to all def ports, local
4	10(node,id_2[,level]) 17(rnode, node)	If site(node) > site(rnode) then 35(site(node), τ, 0, 0, site(rnode), id_2, level, VT(id_2), site(node))	τ use to all def ports, global
5	1(node, pid, name)	35(site(node), τ, 0, 0, 0, τ, 0, 0, site(node))	New procedure
6	User identified	An edge from each of the use ports to final use port.	Final-use
7	8($node_1$, id_1[,level]) 8($node_2$, id_2[,level])	If site($node_1$) = site($node_2$) then 35(site($node_1$), λ, 0, 0, site($node_1$),id_1, level, VT(id_1), site($node_1$)), and 35(site($node_1$), λ, 0, 0, site($node_1$),id_2, level, VT(id_2), site($node_2$))	Multi-def Local variable
8	10($node_1$, id_1[,level]) 10($node_2$, id_2[,level])	If site($node_1$) = site($node_2$) then 35(site($node_1$), λ, 0, 0, site($node_1$),id_1, level, VT(id_1), site($node_1$)), and 35(site($node_1$), λ, 0, 0, site($node_1$),id_2, level, VT(id_2), site($node_2$))	Multi-def Global variable
9	11(node, pid)	If 7(node, . .) and 8(node, . .) are not present then 35(site(node), τ, 0, 0, site(node), ϕ, 0, 0, site(node))	Function calls using/defining no variables
10	15(node, B|C)	35(site(node), τ, 0, 0, site(node), ϕ, 0, 0, site(node))	Break, continue
11	17(node, rnode)	If site(rnode) < site(node) then 35(site(node), τ, 0, 0, site(node), ϕ, 0, 0, site(node))	Flow variable, }

The τ_a def port of a site (blockhead) is connected to τ_a use ports in all sites (members), shown as items 1-3 in table 6.9. In items 4 and 5, ϕ_{node} def ports are connected to τ_{r2node} use port; rnode is a member of r2node, and node is a member of rnode (flow edges).

Table 6.9. Determination of external edges from the LIF

#	LIF	External edges	Comments
1	17(node, rnode1, rnode2)	If site(rnode) > site(node) then 36(site(node), τ, 0, 0, site(node), site(rnode), τ, 0, 0, site(node)) for each rnode1 ≤ rnode ≥ rnode2	Blockhead (node) to members (rnode)
2	17(node, rnode)	If site(rnode) > site(node) then 36(site(node), τ, 0, 0, site(node), site(rnode), τ, 0, 0, site(node))	Blockhead (node) to a member (rnode)
3	17(node, rnode)	if site(rnode) < site(node) then 36(site(rnode), τ, 0, 0, site(rnode), site(node), τ, 0, 0, site(rnode))	Blockhead (rnode) to '}' (node)
4	17(node, rnode) 17(r2node, rnode)	if site(rnode) < site(node) then 36(site(node), ϕ, 0, 0, site(node), site(rnode), τ, 0, 0, site(r2node))	'}' (node) to blockhead (rnode)
5	11(node, pid) 17(rnode, node) 17(r2node, rnode)	If 7(node, . .) and 8(node, . .) are not present and site(rnode) < site(node) then 36(site(node), ϕ, 0, 0, site(node), site(rnode), τ, 0, 0, site(r2node))	Function calls using/defining no variables.

6.3.5 Data dependence edges

To add data dependence edges to the StDG, we need variables def/use information and the control flow information. The control flow information is represented (SUCC) in the LIF as the flow of program execution from node to node. A def to a variable can be preserving or killing (section 3.5.1). A preserving or a killing def is always with respect to a previous def of the same variable; a def may be

preserving with respect to some and killing with respect to other defs. A def of variable *V* in block *B* is a killing def of all previous defs of *V* within *B* or within a deeper block of *B*, and is a preserving def of all previous defs in the outer blocks of *B*. Different types of def types include:

```
Block B1      // all defs in the following lines are to a same variable
      def1    // def1 is killing def
      Block B2
          def2  //def 2 is preserving of def1 and def1 is not visible in this block
          def3  //def 3 kills def2 and preserves def1
      Blockend B2
      def4            //def4 kills both def1 and def3
Blockend B1
```

A node in the LIF may have one or more successor nodes. A node with two successor nodes indicates the beginning of a new block. An algorithm for obtaining the data dependence edges is given in section 6.3.6. Each user-defined procedure is analyzed for data dependence edges separately. The summary site of an analyzed procedure is obtained (as explained later) and used in place of its call statement during the analysis. The summary sites of the library functions are taken from the user. A loop block is analyzed twice and so is a recursive procedure; the summary site obtained in the first analysis (by ignoring the recursive calls) of a recursive procedure is used in the second analysis.

6.3.6 Algorithm for deriving data dependence edges

In this section, we present an algorithm for deriving data dependence edges in the StDG of a procedure.

/* Variable def and use information is used in deriving the data dependence edges. Variable def and use information is obtained from the ports of the StDG. Each

```
        procedure is analyzed separately. Each loop block is analyzed twice. A call to a
        procedure is replaced by the summary site of the procedure. See section 6.3.7 for
        an algorithm for obtaining the summary site of a procedure. */

struct PORT{
   variable_id; // each variable is given a unique id
   variable_level; //pointer or address information of the variable
   variable_type;// use or def
   variable_scope;//local or global
   variable_port_num; // number of the statement in which the variable is present
   variable _site; // site in which the port is present
};

/*      The port of each variable defined is added to the active ports list. Ports of
        variables whose def is a killing def are removed from the list. The block depth of
        each port in the list is also stored along with the port. Initially block depth is
        zero. The block depth counter is increased when a block begin is encountered,
        and it is decreased when a block begin is reached.   */

struct ACTIVE_PORT { live ports (variables)
   PORT ports;
   bdepth; //block depth of the port
};

ACTIVE_PORT active_port_list[];
PORT P1, P2;
HEADR block_header[];
block_depth = 0;//current block depth
required_node = 0; //require information from the LIF codes

/*      Find the PROC START LIF code of the procedure under consideration. The
        code is in the form 1(node_1, pid_1, name_1) where node_1 is the graph node number,
        pid_1 is current procedure id and name_1 is the procedure name. Enter site includes
        def ports of formal parameters and global variables. These ports are add to the
        active ports list with block depth = 0.          */
```

```
current_node = node1;
current_pid = pid1; //
For each port P1 in ENTER(current_pid) site {
      Add P1 and block_depth of 0 to active_port_list[]
}
call process_sites(current_node)

/*    The successor of a node is obtained from the LIF. It is in the form - 16(node1,
      node2) where node2 is the successor node to node1. The site number of a node is
      obtained from the LIF code, it is in the form – 18(node, number, . ) where,
      number is the statement number of node.      */

process_sites(node)
{
   loop forever{ // loop till the proc end is reached
  if node has more than one successor then
  {
      for each successor S call process_sites(S) and
      exit
  }
  next_node = successor(node);
  Site_no = get_Site_number(next_node) // see comments above
  Call process_use_def(site_no)
  switch(site_no){ // site type
   case PROC END: //end of the procedure site . Add edges to ports in the exit site.
     for each P2 in use ports of EXIT(current_pid) site
          for each i, 0<= i < I,  if (P2.id = active_port_list[i] .active_ports.id)
              add an egde from active_port_list[i] to P2;
     end of analysis for the procedure
     break out of the loop;

   case BLOCK BEGIN: // new block beginning
     exit if site_no is a loop and was visited twice already
     block_depth = block_depth + 1
     break;

   case BLOCK END: // block end site
     block_depth = block_depth – 1
     break;

   case CALL START: //site is a call to a procedure
     replace current site by the summary site of the called procedure and
```

```
        replace formal parameters by actual parameters, and
        call process_use_def(summary_site)
        break;

    }// end of switch
  }//end process_sites()

/*  For each use port (P1) of variable V in the site, edges are added from the ports
    (P2) of variable V in the active ports list to P1 if block depth of P2 >= block
    depth of P1. For each def port (P1) of variable V in the site, P1 and block depth
    of P1 are added to the active ports list. Ports (P2) of variable V in the active
    ports list with block depth >= block depth of P1 are removed from the active
    ports list.      */

process_use_def(site)
{
    for each use port P1 in site
    {
        for each port p2 in active_port_list[] .ports
        if (P2.id = p1.id && P2.bdepth >= block_depth) add an egde from p2 to P1
    }
    for each def port P1 in site
    {
        for each port p2 in active_port_list[] .ports
        if (P2.id = p1.id && P2.bdepth >= block_depth)
        {
            remove P2 from active_port_list[]
            add P1 and block_depth to active_port_list[]
        }
    }
    }
```

6.3.7 Summary site

An algorithm for computing the summary sites of procedures is presented in table 6.10. A summary site of a procedure can be computed only after its StDG is completely derived.

Table 6.10. Algorithm for computing summary site

```
For each port P1 in Enter site of the procedure
        add P1 to use ports of the summary site
For each port P1 in Exit site of the procedure
        add P1 to def ports of the summary site
For each port P1 in Enter site of the procedure
{
   find FSP(P1) // see section 5.2.1 of Chapter 5 (forward slice ports)
   for each port P2 in FSP(P1) and Exit site
        add an edge from P1 to P2 in summary site
}
```

6.4 Visual Representation of Design

The software design is presented in the form of an architectural design and a detailed design. The architectural design or the high-level design consists of procedures and their interconnections. The detailed design or the low-level design includes the design associated with the individual procedures [Scha 96]. The reverse engineered architectural design, in the form of a call graph, and the detailed design, in the form of the StDG, is represented in the StDG codes. The overall program understandability and maintainability can be improved by generating graphical representations of different views of the program.

6.4.1 Control flow and dataflow graphs

Control flow and dataflow graphs (CFG and DFG) are obtained from the StDG by considering each site as a node. Each external edge between τ ports of the sites is made a control flow edge in the CFG. For each loop site s, a control flow edge is added from the last member of s (site with highest site number within block s) to s. Similarly, each external edge joining data variable ports in the StDG is made a data flow edge in

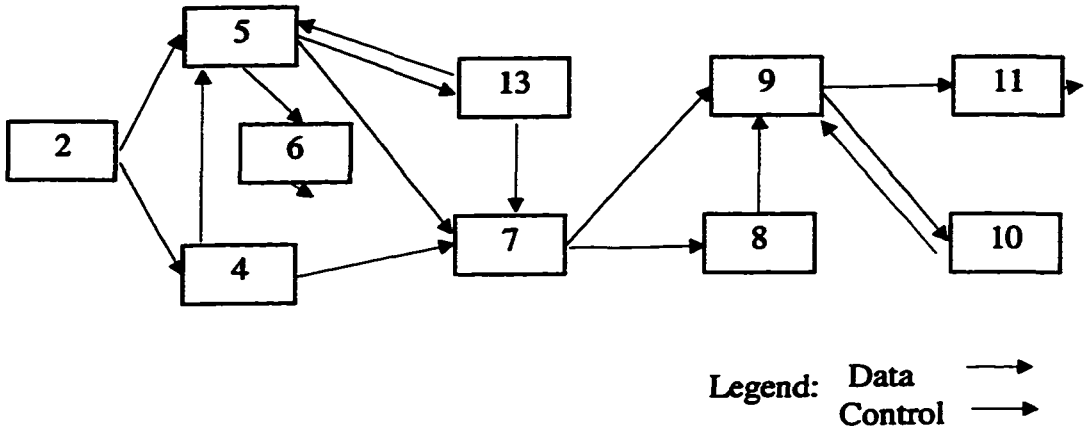


Figure 6.2. CFG and DFG

the DFG. The CFG and DFG of the function wordCount() of figure 3.1(a) is shown in figure 6.2.

6.4.2 Call graph

The StDG code FUN CALL lists the calling and called procedures, including the library routines. The FUN CALL and SUMMARY codes, *38(calling procedure id, called procedure id) and 37(procedure id, site #, name)*, can be used to obtain the call graph. The library routines can be excluded from the call graph by excluding the procedures with zero procedure ids. The call graph of *wc* program of figure 3.1(a) is presented in figure 6.3.

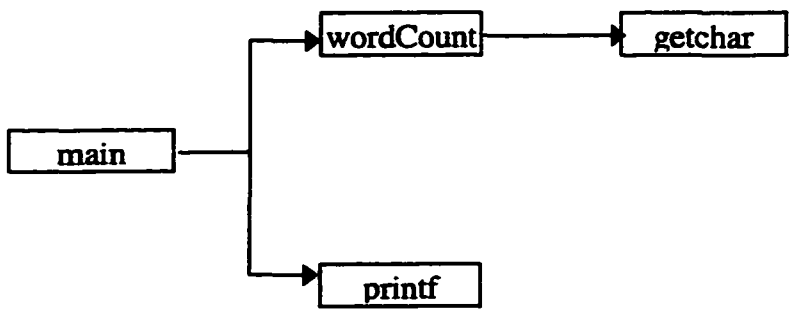


Figure 6.3. Call Graph

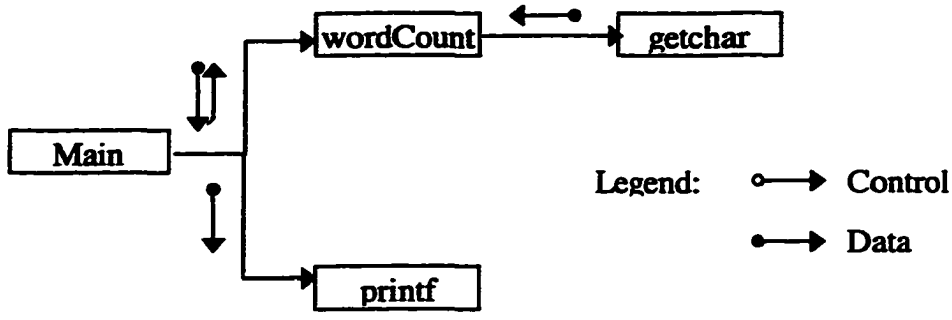


Figure 6.4. Structure chart representation of call graph

6.4.3 Structure charts

The structure chart of the modules and their hierarchical relationships of the *wc* program of figure 3.1(a) is shown in figure 6.4. The structure chart also identifies the data passed between the modules.

6.5 Summary

Reverse engineering is the process of analyzing a subject system to understand and represent it at a higher level of abstraction. The subject system can be in the form of code or design documents. A process for reverse engineering the source code is presented in this chapter. The language independent format (LIF) is an intermediate representation for C programs. The LIF was developed as part of *Unravel* CASE tool, available in the public domain. We use the *Unravel* for generating the LIF, and use the LIF for generating the design of the system. The design obtained is represented in the form of statement dependence graphs (StDG). From the StDG, different views of the system are generated. These graphical representations aid overall comprehensibility and improve the maintainability of the source system.

Chapter 7

Design Change

7.1 Introduction

An object-oriented system uses the principles of abstraction, encapsulation, modularity, and hierarchy together in a synergistic manner [Booc 94] unlike a procedural system which may encompass one or more of these principles. However, the meaning of these principles as used in a procedure-oriented system differs vastly from an object-oriented system. To convert a procedure-oriented system to an object-oriented system, we need to either identify these principles (if present in procedural systems) or introduce them.

An object has a state, exhibits some well-defined behavior, and has a unique identity [Booc 94]. Behavior is how an object acts and reacts in terms of its state changes. Object behavior is expressed through operations (methods) like, modifiers, selectors, constructors and destructors. A modifier alters, a selector accesses, a constructor initializes, and a destructor frees the state of an object. However, in a procedural system, no distinction is made between state and behavior. Also, different types of operations exist as interleaved code.

Rugaber [Ruga 95] defines interleaving as merging of two or more distinct plans within some contiguous textual area of a system. A plan denotes a group of statements present in a system to achieve some purpose or goal. In terms of objects, a plan may consist of one or more operations. Interleaving may occur for several reasons such as efficiency considerations and the sequential nature of procedural programming.

For example, it is more efficient to compute two related values at one place rather than separately. Also, constructors and modifiers of data structures are typically interleaved throughout a procedure.

To convert a procedure-oriented system to an object-oriented system, the norm has been to identify state from user-defined data structures and/or global variables, and then identify procedures as behavior. The procedures may be fine-tuned by removing extraneous code. The scope of these techniques is limited in two ways. First, it limits the object identification to user-defined data structures (and global variables) and to procedures that access these data structures. Second, it uses traditional program slicing [Horw 90] for splitting large procedures. Program slicing is a focusing technique based on dependences. A slice includes every statement that affects the slice (whatever the source may be). This approach may result in large slices with a broad focus [Jack 94]. Moreover, the objects obtained using these techniques are usually coarse-grained.

To narrow the focus of a slice, a decomposition slicing for function extraction by duplicating certain statements to be shared by a slice and its complement is given in [Gall 91][Lanu 93]. However, these decomposition slices are still large and also require a criterion for slicing, which is not always easy to formulate. Furthermore, slicing is more tuned to function extraction than to transformational reengineering [Gall 91][Lanu 93]. In section 7.2, we present approaches available in the literature for object identification. Section 7.3 describes a new approach to object identification. In section 7.4, a brief summary of the chapter is presented.

7.2 Object Identification Approaches

Research in object identification has been mainly focused on developing techniques for extracting objects from data that has already been aggregated in programmer-defined data structures [Ruga 95]. Concept analysis is applied in [Siff 97][Liu 90] to identify potential modules. Concept analysis uses functions (or a slice) and attributes of functions to identify potential objects. The attributes of functions may include parameter and return types of functions, global variable usage information, and slice criteria. The concept analysis approach generates a variety of possible decompositions from which a user can select an appropriate decomposition. The optimality of this approach depends on how well the attributes of functions are formulated for the concept analysis, which requires that the user have a good understanding of the system.

In [Wigg 97][Canf 96][Yeh 95], a clustering technique is used to identify objects. The technique uses a graph with procedures and external (and global) variables as nodes and references by the procedures to the variables as edges. Each isolated sub-graph contained in the graph is a candidate to implement an object. For these techniques to apply, either the state variables must be identified by some mechanism or they must be declared as global variables. Moreover, these techniques use traditional slicing to extract relevant functionality from the functions.

Cohesion-based object identification approaches are presented in [Ache 95][Chu 92]. The usage information of pairs of global or parameter variables is used to arrive at different decompositions of a system in [Ache 95]. In [Chu 92], functions that refer to the same group of global variables are grouped into packages. These approaches use

functions as operations of objects, but techniques can only be applied to programs that are already divided into operations.

Several automatic object identification approaches are based on graphs and their properties. The most common approach consists of defining a model of the subject system as a graph on which notable sub-graphs and/or patterns are identified. Each sub-graph or pattern is a potential object [Canf 96].

Generally these approaches follow 4 steps to identify object like features. These steps are:

i. Identify target variables as candidates for object state. [Gall 95] uses programmer-defined data structures while [Ache 95] uses actual parameters, common variables and array variables as target variables. [Liu 90][Dunn 93][[Yeh 95][Canf 96][Siff 97] use global or external variables as target variables.

ii. Establish a relationship between the target variables and the procedures (functions) in the system. The most commonly used relationship is 'uses'. [Liu 90][Dunn 93][[Yeh 95][Canf 96] use - procedure 'p' uses variable 'v' - type of relationship. [Siff 97] uses both 'uses' (positive information) and 'doesn't use' (negative information) relationships. [Ache 95] uses 'pairs of variables used together' relationship. This usually results in a graph or a matrix, with both target variables and procedures as nodes and the relationship as edges.

iii. Identify clusters or sub-graphs or patterns within the graph or matrix. Each of these sub-graphs or clusters is a candidate object.

iv. Fine tune the sub-graphs or clusters. A procedure may belong to more than one sub-graph. It may need to be sliced or removed from some or all sub-graphs. [Canf 96] calls these undesired links as coincidental and spurious connections. Coincidental connections are due to routines that implement more than one function, each function logically belonging to a different object. Spurious connections are related to routines that access more than one data structure. Slicing is proposed to separate objects connected by coincidental connection, and routines resulting in spurious connections are discarded from the graph. [Yeh 95] follows a similar approach. [Siff 97] overcomes this obstacle using a language feature 'friend' available in C++, which provides access to state variables of other objects.

In section 7.2.1, we discuss how different methods identify object-like features in the source code.

7.2.1 Liu and Wilde approach

For each global variable 'v', a set F(v) of procedures that directly reference 'v' is computed. A graph is constructed with each F(v) as a node. For each pair of nodes ($F(v_1)$ and $F(v_2)$), an edge connecting the two is added if there is a common procedure in the sets, $F(v_1)$ and $F(v_2)$. Figure 7.2 is a graph built using the Liu and Wilde approach for the program in figure 7.1. Each of the two strongly connected sub-graphs recognized in the graph is a candidate for object. Objects stack and queue are easily identified from figure 7.1.

```
long stackItems[MAX];
int stackPoint;
int queueItems[MAX];
 int queueHead, queueTail, numQueueElm;
void stackInit() {/* referencess stackPoint */}
void stackPush(elm) {/* references stackItems and stackPoint */}
int stackPop() {/* references stackItems and stackPoint*/
int stackTop() {/* references stackItems and stackPoint */
int stackEmpty() {/* references stackPoint */
void queueInit() {/* references queueHead, queueTail, numQueueElm */}
void queueEnq(elm){/*references queueItems, queueHead,
numQueueElm*/}
int queueDeq() {/* references queueItems, queueTail, numQueueElm */
int queueEmpty() {/* references numQueueElm */
```

Figure 7.1 A sample C program for Liu and Wilde approach

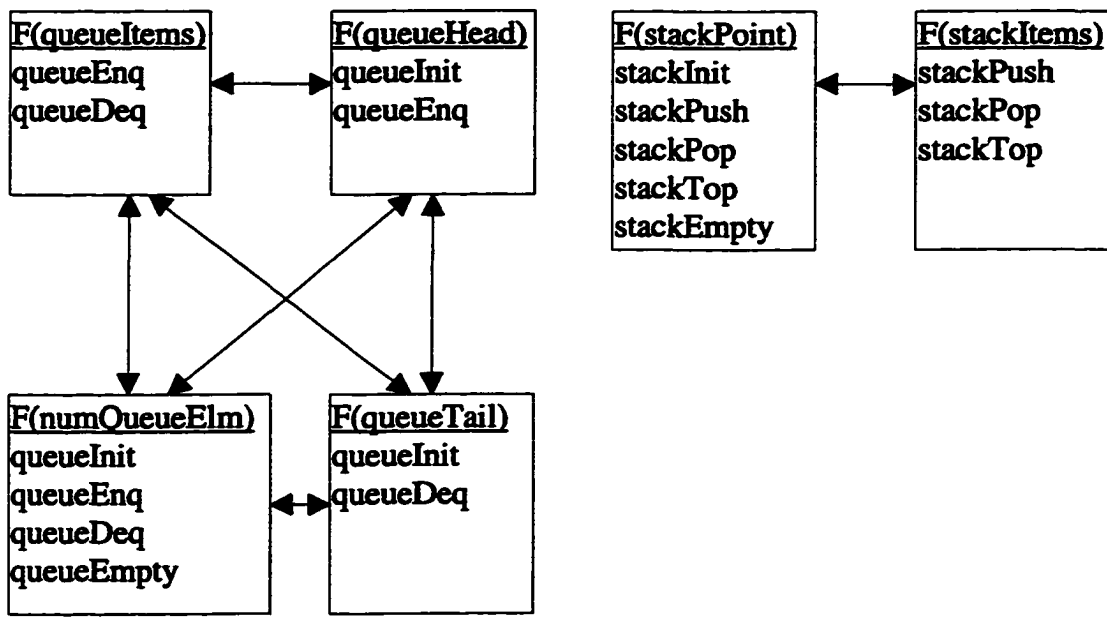


Figure 7.2. Strongly connected sub-graphs in Liu and Wilde approach

7.2.2 Dunn and Knight approach

In the Dunn and Knight approach, a program is represented as a graph with global variables and procedures as nodes. Edges, which are directed from procedure nodes to variable nodes, specify the 'uses' relation. The graph is traversed depth-first

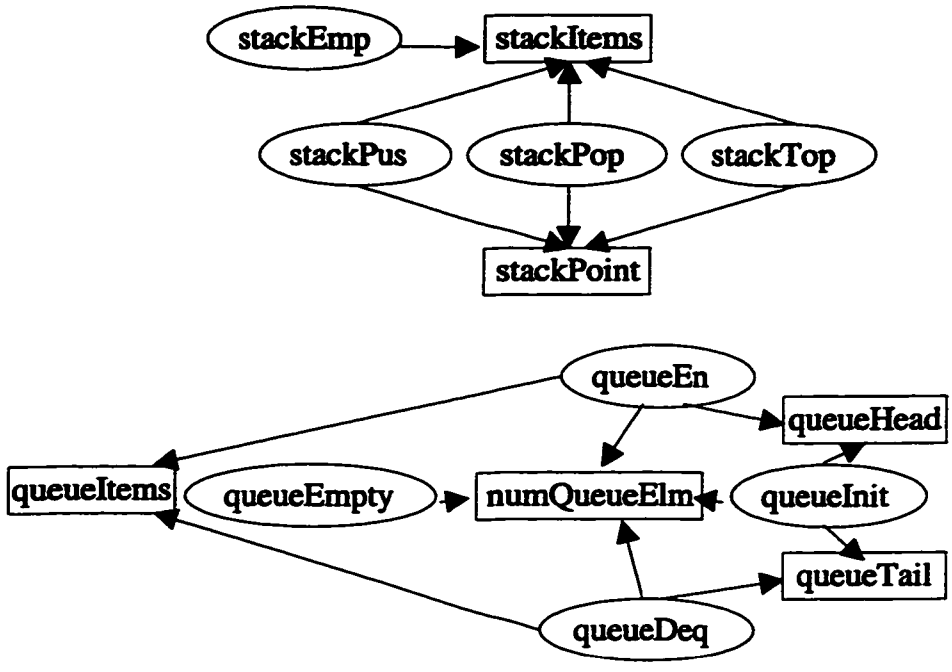


Figure 7.3. Strongly connected sub-graphs in Dunn and Knight approach

looking for strongly connected components; each component is regarded as a candidate object. Figure 7.3 shows the graph for the sample program given in Figure 7.1. The results are essentially equivalent to those obtained from the Liu and Wilde approach.

7.2.3 Siff and Reps approach

Approaches based on search for notable sub-graphs or patterns may produce low quality objects by clustering more than one object within the same candidate. A procedure using state variables of two different objects creates a link between the corresponding sub-graphs, thus causing the two objects to be identified as a unique one. As an example, consider the sample program as shown in Figure 7.4.

Routine 'queueEnq' references fields of both stack and queue structures. This routine creates a link between the two objects, thus recognizes the entire program as a single object.

```
        struct stack {
                long stackItems[MAX];
                int stackPoint;
        };
        struct queue {
                struct stack *front, *back;
        };/* two stacks are used to simulate a queue */

void stackInit() {/* references stackItems and stackPoint */}
void stackPush(elm) {/* references stackItems and stackPoint */}
int stackPop() {/* refernces stackItems and stackPoint*/
int stackTop() {/* refernces stackItems and stackPoint */
int stackEmpty() {/* refernces stackPoint */
void QueueInit() {/* references front and back */}
void queueEnq(queue *Q, elm) {/* references Q->front->stackPoint, and
                                        Q->front->stackItems */}
        /* queueEnq is  making changes to stackPoint and stackItems directly */
int queueDeq() {/* references front and back*/
 int queueEmpty() {/* references front and back*/
```

Figure 7.4 A sample C program for Siff and Reps approach

To alleviate this problem, Siff and Reps make use of both positive and negative information, unlike other approaches which make use of only "positive" information. For example, knowledge that "function 'f' uses the fields of 'struct queue' but not the fields of 'struct stack'" is sometimes helpful in solving these problems. In addition to "uses" information, this approach uses the fact that 'queueEnq' has argument of type queue and not of type stack to determine if the routine belongs to a stack or queue object. This approach identifies the routine as belonging to queue object. But, one problem that is not solved completely is that the queue object accesses the state variables of a different object.

```
long stackItems[MAX];
int stackPoint;
int queueItems[MAX];
int queueHead, queueTail, numQueueElm;

void globalInit() {/*references stackPoint,
                        queueHead, queueTail, numQueueElm */}
 void stackPush(elm) {/* references stackItems and stackPoint */}
 int stackPop() {/* references stackItems and stackPoint*/
 int stackTop() {/* references stackItems and stackPoint */
 int stackEmpty() {/* references stackPoint */
 void queueEnq(elm) {/* references queueItems, queueHead,  numQueueElm
*/}
 int queueDeq() {/* references queueItems, queueTail, numQueueElm */
 int queueEmpty() {/* references numQueueElm */
 void stackToQueue() {/* references stackItems, stackPoint, queueItems,
                                queueItems, queueHead, numQueueElm */}
```

Figure 7.5 A sample C program for CCM approach

7.2.4 Canfora, Cimitile and Munro (CCM) approach

Another problem with Siff and Reps approach is that it is not always the case that every procedure in the system can be identified with one group or the other. As an example, suppose that the program in Figure 7.4 initializes both a stack and a queue in one routine. In this case the routines 'stackInit' and 'queueInit' are substituted with a single routine, 'globalInit' (figure 7.5), which accesses the state of both the objects:

```
void globalInit() {/* references stackItems, stackPoint, front, and back */}
```

Consider another example in which we have another routine stackToQueue that accesses 'stackItems' and 'queueItems' to copy items from the stack to the queue:

```
void stackToQueue() {/* references stackItems and queueItems, etc. */}
```

The two routines create links between the objects, thus forcing their clustering into the same candidate object. [Canf 96] calls these types of links as coincidental and spurious

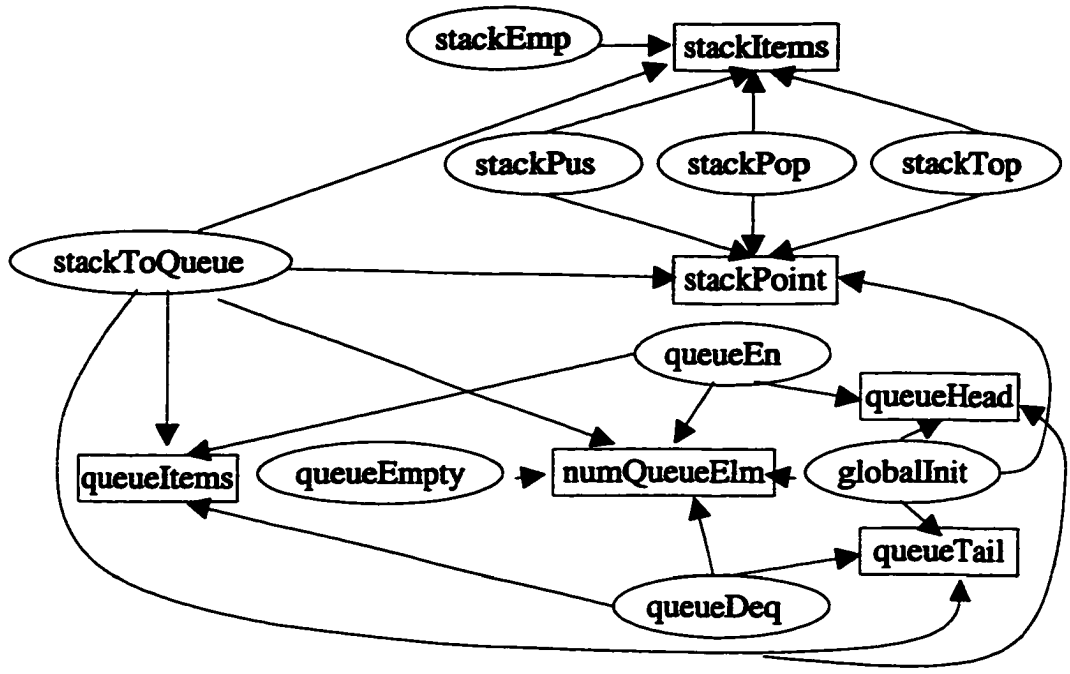


Figure 7.6. Connected components in the CCM approach.

links, respectively. Routines creating coincidental links, like globalInit, are sliced and used in different objects. In the CCM approach, routines that create spurious links, like stackToQueue, are assumed to exist to implement system specific operations and to access several objects (e.g. main function in C programs). These routines do not belong to any object and are removed from the identification process.

The CCM graph and the Dunn and Knight graph are built in a similar fashion. The graph, known as the *variable-reference graph*, records the usage of global variables. If GV is the set of global variables in a software system and F is the set of routines, a variable reference graph is a bipartite graph G(N,E) with nodes $N \equiv GV \cup F$ and edges $E \equiv \{(f,d) \mid f \in F \wedge d \in GV \wedge$ 'f references d'$\}$ [Canf 96]. The variable-reference graph for the program in Figure 7.5 is shown in Figure 7.6.

The CCM approach attempts to identify the relevant, internally connected, sub-graphs through an iterative process. In each iteration, every routine f in the system is associated with an index f_x. f_x measures the variation in the internal connectivity of the graph using f, to generate a new cluster. Such a cluster would include all the data items referenced by f (say, set d), and all the routines that reference the data items in set d. f_x of f measures the difference between the internal connectivity of the cluster generated by f and the internal connectivity of the clustered sub-graphs. This index (f_x) is used to discriminate the routines that implement the operations of an object (high index) from the routines that access data items that belong to different objects (spurious/coincidental connections – low index). Routines with low index are either sliced or deleted with the programmer's help. Data items (set d) referenced by routines with high index are merged into one node, and the clustering process is repeated till the graph is in the form of a set of isolated sub-graphs each consisting of a single GV node and one or more F nodes.

The CCM approach can identify the two objects in the program in Figure 7.5 and also identify that routines 'globalInit' and 'stackToQueue' need to be sliced or deleted. Though the scope of this approach is wider than previous approaches, the CCM approach has several limitations. The determination of low or high indexes is subjective. It varies from program to program and also from iteration to iteration. Another limitation is that it is not always possible to decide if a particular procedure belongs to an object by simply looking at the number of data variables it references since a procedure may access object state of several potential objects.

```
1.    int main()
2.        {
3.            long items[MAX];
4.            int sp;
5.            long i, j;
6.            sp = -1;
7.            ...
8.            items[++sp] = j;
9.            ...
10.       i  = items[sp];
11.       ...
12.       items[++sp] = j;
13.       ...
14.       items[++sp] = j;
15.       ...
16.       i = items[sp--];
17.       ...
18.       items[++sp] = j;
19.       ....
20.   }
```

Figure 7.7. Sample C program for RSG approach

7.3 The RSG Approach

Implicit assumptions common in the approaches in section 7.2 are that the system under consideration is built around well-defined data structures and that the procedures are well-designed. In reality these assumptions do not always hold. Moreover, the approaches fail to identify objects that may be present within a procedure, as in figure 7.7. Further, they make no distinction between a use and a definition of variables. Consider as an example a procedure using a variable (v_1) and defining another variable (v_2). Assume that v_1 and v_2 belong to two different potential objects. Previous approaches create two equally weighed links, therefore failing to identify the procedure with one object or the other. A procedure (p) using a state variable (v_3) need not

necessarily be part of the object belonging to the state variable. There are three choices for p and v_3:

i. Make p and v_3 part of the same object.

ii. Make p and v_3 part of different objects, and provide access to v_3 through language features (e.g. 'friend' in C++).

iii. Make p and v_3 part of different objects, and introduce a selector operation for v_3.

If p and v_3 belong to different objects, then we have only the latter two choices. The second choice can only be applied in limited cases; we cannot declare every object in the system as a friend of every other object. The third choice is not explored by any of the approaches presented in section 7.2.

7.3.1 Code localization

The opposite of interleaving is localization. In a procedural program, code for the individual plans is interleaved due to the following reasons:

i. Programming style. Procedural programming does not prevent a programmer from mixing the code of different plans in some textual area of the program.

ii. Sharing of intermediate results. If two plans share some code, then instead of duplicating the common code the result of common code is shared by the two plans, resulting in interleaving of the two plans.

iii. Sharing of variable names. The same variable name can be used for two different purposes (plans) resulting in some kind of relationship between the two plans.

iv. Sharing of resources. Plans also result in interleaving by sharing loop or conditional structures.

While interleaving is introduced to take advantage of commonalties, in contrast, the interleaved plans have a distinct purpose [Ruga 95]. These plans may belong to different objects in an object-oriented decomposition of the program. To identify these objects, first we need to identify these plans, localize the delocalized code belonging to individual plans, and finally group these plans into objets.

Traditionally, slicing is proposed for code localization and for extracting individual plans. A traditional backward slice on 'items' at the end of the sample program (Figure 7.7) includes all the statements in the program. Slicing requires a criterion, a begin statement, an end statement, and variable (s). For the sample program, it is not obvious where to start and end slicing for identifying a plan.

7.3.2 Plan identification

Plans are interleaved to share:

i. Name space. To identify name sharing we need to separate a definition of a variable and its uses from other definitions of the variable. Each definition-uses combination may belong to different plans.

ii. Intermediate results. A value shared by two plans is identified by identifying a variable definition that is used in more than one context. Statements computing the intermediate value (sub-plan) are duplicated if the two contexts where the intermediate value is used are grouped into different objects.

iii. Resources. Resources like loops, control structures, or flags, can be identified by the values they compute and the contexts where the values are used. For example, a loop initializing two arrays can be identified by how the two array variables are used. If they are used for two different purposes, then they do not

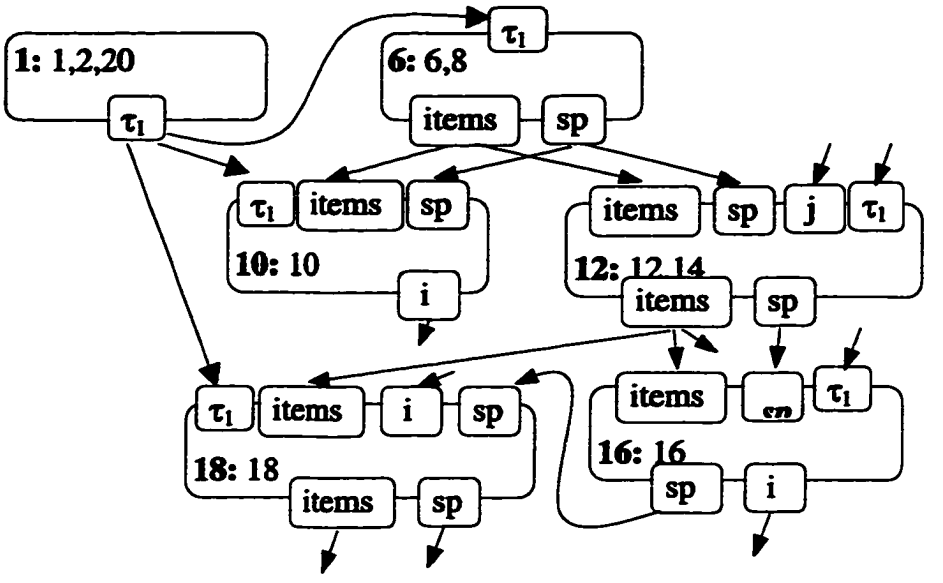


Figure 7.8. RSG of the program in figure 7.7

belong together. The two initializations (or plans) are separated by duplicating the resource (code).

From the above discussion, we can see that plans can be identified by using variable(s) definitions and their use contexts. In this work, code is localized and program plans are identified through restructuring of the graph representation of the program (StDG), as explained in section 7.3.3.

7.3.3 RSG for object identification

The RSG of the StDG of the program in figure 7.7 is shown in figure 7.8. In figure 7.8, only the statements represented by each site and reaching defs and exposed defs are shown. Code is localized by bringing together all statements with high cohesion which are scattered throughout the program. The sites of statements with high cohesion are merged through graph restructuring, as explained in Chapter 4, using data compaction, structure compaction, and edge compaction.

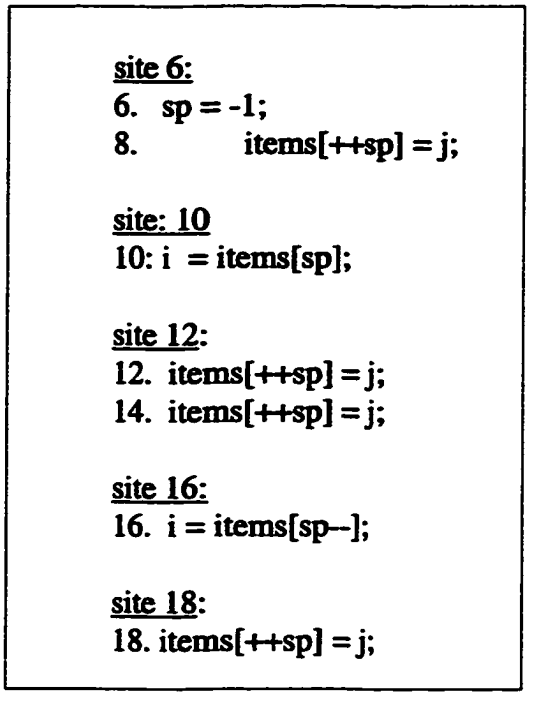


Figure 7.9. Statements in the sites of RSG in figure 7.8

The statements in sites of the RSG (figure 7.8) are shown in figure 7.9. Each of these sites is a potential module. One or more sites in the RSG constitutes a plan. A site defining a control variable (e.g. site 1) is a sub-plan shared by two or more other sub-plans (sites connected by a control dependence edge, e.g. (τ_1, τ_1)). As shown in figure 7.8, several plans (sites 1 and 6, and sites 1 and 10) are interleaved to take advantage of the common computation present in site 1. To make these plans independent, we need to duplicate the computation in site 1.

7.3.4 Restructured program design

The RSG of the program in figure 3.1 is shown in figure 7.10. In figure 7.10, two sub-plans (sites 6 and 7) share the resources (site 1). The structure chart representation of the RSG is shown in figure 7.11. Typically, modules or group of modules are represented within structure charts. The granularity of the call graph node

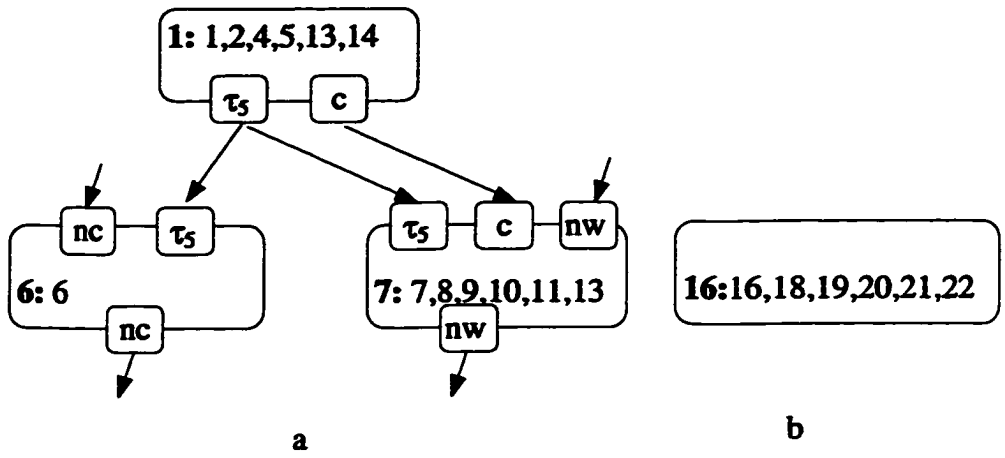


Figure 7.10. (a) RSG of figure 3.1(b). (b) RSG of figure 3.1(c)

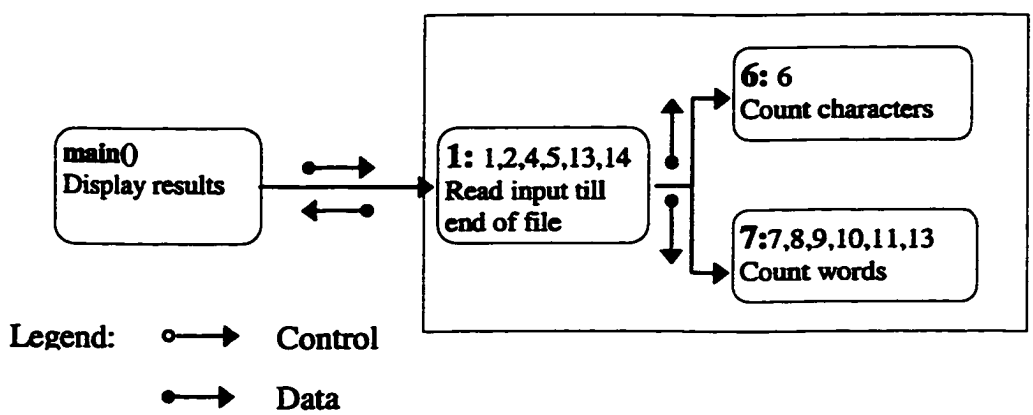


Figure 7.11. The structure chart representation of RSG

(procedure) as a module is too coarse, and a site of StDG (statement) as a module is too fine for understanding a program. The nodes in figure 7.11 are annotated with the help of the programmer.

7.4 Summary

Code and program plans are interleaved in procedural programs. In this chapter we discuss how the code is localized and plans are identified, using the RSG of the program. Usually, slicing is used for this purpose, but slicing is not always feasible to

identify plans because specifying criterion for a plan is difficult. The plans identified are potential candidates for object operations. These object operations are grouped into objects, as explained in Chapter 8. We use the restructured design, call graph (figure 6.3), and the program code (figure 3.1(a)) to identify objects. We also discussed object identification approaches available in the literature.

Chapter 8

Forward Engineering

8.1 Introduction

We use the reverse engineered and restructured design of the programs (RSG) for object identification. The procedural program and the identified objects are used to translate the program to an object-oriented program. This work is focused on identification and extraction of objects. In section 8.2, we present the object identification process. In section 8.3, sample programs are used to show how objects are identified. Object extraction is presented in section 8.4. In section 8.5, we present a case study where we apply the object identification process to a commercial program. Finally, the chapter summary is presented in section 8.5.

8.2 Object Identification Process

An object has state (variables) (SV) and operations (P) in the form of constructors, modifiers, selectors, and destructor. Each site in the RSG is a potential operation, and the program variables defined by these sites are potential state variables of objects. The StDG object identification is a three-step process:

Identification of object state (SV). Variables present in a program (V) are selected as potential candidates for object state (potential objects state variable) from three sources. These are:

1. programmer defined data structures, local and global, present in the program (DV).

2. programmer identified variables (IV). A programmer can choose any variable as a candidate for SV.

3. exposed defs with more than one def edge in the sites of the RSG are candidates for SV (PV). That is,

$SV \subseteq V$:ProgramVariables
$SV = DV \cup IV \cup PV$
$PV = \{v:V;\exists j$:ProgramStatement;s:SITE $|$
$(v_j,s,\text{Definition}) \in DPS(s) \wedge \#DPD(v_j,s,\text{Definition}) > 1\}$

$(\#ECDP(S) = 1 \wedge (\tau_j,S,\text{Definition}) \in ECDP(S)) \vee (\#ECDP(S) > 1 \wedge (\tau_j,S,\text{Definition}) \in ECDP(S) \wedge (\tau_k,S,\text{Definition}) \in ECDP(S) \wedge \forall k \bullet j < k)$.

(v_j is a def port in s and number ports connected to v_j is greater than one). Next we define terms to explain the object identification process, using the example in figure 8.1.

Exposed def ports (EDP). Exposed def ports are the def ports in a site that are connected to ports in other sites. These ports can be control (ECDP) or data (EDDP) ports.

$EDP(S:SITE) = \{(v_j,S,\text{Definition}) \mid (v_j,S,\text{Definition}) \in DPS(S) \wedge DPD(v_j,S,\text{Definition}) \cap UPS(S) \neq \emptyset \bullet$ v:Variables;j:StatementNumber. EDP is a def port in S and def ports of the def port v_j are all not use ports in S.

$EDDP(S:SITE) = \{(v_j,S,\text{Definition}) \mid (v_j,S,\text{Definition}) \in EDP(S) \wedge v \neq \tau \bullet$v:Variables j:StatementNumber $\}$

$ECDP(S:SITE) = \{(v_j,S,\text{Definition}) \mid (v_j,S,\text{Definition}) \in EDP(S) \wedge v = \tau \bullet$v:Variables j:StatementNumber $\}$

Merged use ports (MUP). Merged use ports are the use ports and non-exposed def ports in a site.

$MUP(S:SITE) = \{(v_j,S,\text{Use}|\text{Definition}) \mid (v_j,S,\text{Use}) \in UPS(S) \vee ((v_j,S,\text{Definition}) \in DPS(S) \wedge (v_j,S,\text{Definition}) \notin EDP(S)) \bullet$ v:Variables;j:StatementNumber$\}$

Final def variable (FDV). FDV of a site is the variable defined last in the site. FDV of a site is determined using one of the steps following. The steps are followed in the order presented. Let S be the site under consideration.

Table 8.1. C Code to implement a queue with two stacks.

```
struct stack { int *base, *sp, size; };
struct stack { struct stack *front, *back; };
struct queue* q;
struct stack * initStack(struct stack* s, int sz)
{ s = (struct stack*) malloc (sizeof(struct stack)));
  s->base = s->sp = (int*) malloc (sz * (sizeof(int)));
  s->size = sz; }
struct queue* initQ()
{  q = (struct queue*) malloc (sizeof (struct queue));
 initStack(q->front ,10);
 initStack(q->back,10}

int isEmptyStack(struct stack* s)
{  return (s->sp == s->base) ;}

int isEmptyQ()
{   return (q->front->sp == q->front-base && q->back->sp == q->back-
>base) ;}

void push(struct stack* s, int i)
{  *(s->sp) = i;
  s->sp++; }

void enq(int i)
{  *(q->front->sp) = i;
  q->front->sp++; }

void pop(struct stack* s)
{  if (isEmptyStack(s)) return -1;
  s->sp--;
  return (*(s->sp)); }

int deq()
{  if(isEmptyStack(s)) return -1;
  if(!isEmptyStack(q->front)) push(q-back, pop(q->front));
  return pop(q->back); }
```

1. If ECDP of S is not null, then one of the ECDP variables is the FDV. The τ port with the least statement number is the FDV. That is,

   $FDV(S:SITE) = (v \bullet v:Variables;j,k:StatementNumber \mid v = \tau_j \wedge (\#ECDP(S) = 1 \wedge (\tau_j,S,Definition) \in ECDP(S)) \vee (\#ECDP(S) > 1 \wedge (\tau_j,S,Definition) \in ECDP(S) \wedge (\tau_k,S,Definition) \in ECDP(S) \wedge \forall k \bullet j < k)$.

FDV of S is τ_j if τ_j is the only port in ECDP of S or ECDP has more than one port, τ_j and τ_k are two ports in ECDP of S, and for all k, $j < k$.

2. If ECDP(S) is null, then one of the variables in the EDDP(S) is the FDV. The variable of the port in the EDDP with the highest statement is the FDV. If more than one port has the same highest statement number, then the variable not used within S (variable of port with no internal def edges) is the FDV (there will be only one such variable in a site). That is,

   $FDV(S:SITE) = (v \bullet v,w:Variables;j,k,l:StatementNumber;S1:SITE \mid (v = v_j \wedge \#ECDP(S) = \varnothing \wedge (EDDP(S) = 1 \wedge (v_j,S,Definition) \in EDDP(S)) \vee ((EDDP(S) > 1 \wedge (v_j,S,Definition) \in EDDP(S) \wedge (w_k,S,Definition) \in EDDP(S) \wedge \forall k \bullet j > k) \vee (v_j,S,Definition) \in EDDP(S) \wedge (w_k,S,Definition) \in EDDP(S) \wedge \forall k \bullet j >= k \wedge (v_n,S1,Use) \in DPD(v_j,S,Definition) \wedge S \neq S1 \wedge ((v_j,S,Use),(w_j,S,Definition)) \notin InternalEdge(S))$.

   The FDV of S is v_j if number of ports in ECDP(S) is null and 1) EDDP(S) has one port (v_j), or 2) EDDP has more than one port, v_j and w_k are any two ports in EDDP and $j > k$ always, or 3) EDDP has more than one port, v_j and w_k are any two ports in EDDP and $j >= k$ always, use port v_n in S1 is one of the DPDs of v_j, S and S1 are different, and edge $((v_j,S,Use),(w_j,S,Definition))$ is not an internal edge in S.

3. FDV of S is null if EDP of S has no ports. That is,

   $FDV(S:SITE) = (v \bullet v:Variables \mid v = \phi \wedge EDP(S) = \phi)$.

State variables defined in a site (DSV). DSV represents all the state variables defined in the site. DSV is formally defined as:

$$DSV(S:SITE) = \{v\bullet\ v:Variables;\ j:StatementNumber \mid (v \in SV \wedge (v_j,S,Definition) \in DPS(S)\ \}$$

State variables used in a site (USV). USV represents all state variables used in the site. USV is formally defined as:

$$USV(S:SITE) = \{v\bullet\ v:Variables;j:StatementNumber \mid (v \in SV \wedge (v_j,S,Use) \in UPS(S))$$

Final def state variable (FDSV). FDSV of a site S is the state variable defined last in S. If FDV of a site is a control variable or a local variable, then the site may have a different variable as FDSV than FDV. The FDSV is formally defined as:

$$FDSV(S:SITE) = (v\bullet\ v,w,y:Variables;j,k:StatementNumber \mid ((v \in SV \wedge v \in FDV(S)) \vee ((y \notin SV \wedge y \in FDV(S) \wedge w \in SV \wedge (v_j,S,Definition) \in EDDP(S) \wedge (w_k,S,Definition) \in EDDP(S) \wedge \forall k\bullet\ j > k\ )) \vee (v = \varnothing \wedge \forall w\bullet\ w \in SV \wedge w \notin FDV(S))$$

2. Identification of object operations (P). Each site in the StDG is a potential operation. A RSG site may define zero or more SVs. If more than one SV is defined in a site, then that site needs to be separated into different sites, with each site defining exactly one SV. However, in certain cases where SVs are interdependent, it may not be possible to separate sites. One simpler way to separate a site defining more than one SV is to re-restructure the StDG, with the list of SVs (this is referred to as RSG2 in figure 1.1). During sites merging using data compaction (Chapter 4, section 4.3.2), cases one and two, site s1 are not merged in s2 if s1 satisfies the following condition:

$$((v,s1,Definition) \in DPS(s1) \wedge v \in SV)$$

In the remainder of this work, RSG refers to the re-restructured StDG. A site in the RSG has one or more def ports, representing state variables, control variables, or local variables. Each of these sites is associated with a variable known as a final def variable. Sites and other definitions for the example in table 8.1 are presented in table 8.2. Procedure initStack has three sites and initQ has two sites; each of these sites will be considered as a different operation. All other procedures have one site each in the StDG of the example.

Table 8.2. Sites and other definitions for the example in table 8.1.

Procedures	Sites	FDV	FDSV	DSV	USV
initStack	1. base 2. sp 3. size	base sp size	base sp size	{base} {sp} {size}	∅ ∅ ∅
initQ	1. front 2. back	front back	front back	{front} {back}	∅ ∅
isEmptyStack		∅	∅	∅	{sp,base}
isEmptyQ		∅	∅	∅	{sp,base,front,back}
push		sp	sp	{sp}	{sp,size}
pop		sp	sp	{sp}	{sp}
enq		front	front	{sp,front}	{sp ,front }
deq		back	back	{front,back}	{front,back}

3. Identification of candidate objects. The following steps are followed to identify the objects. Let,

OV(o) - state of object o ($OV(o) \subseteq SV$).
OP(o) - operations in object o ($OP(o) \subseteq P$).
O - set of objects in the program ($o \in O$).

Step 1. For each data structure present or group of variables identified by the software engineer (d), add an object o_d to objects set. Members (v) of a group (d) are added to

their respective objects. That is, $(d \in DV \wedge O = O \cup o_d) \wedge (v{:}V \in d \wedge OV(o_d) = OV(o_d) \cup \{v\})$. If d is a DV then add o_d to O and if v is a member of d then add v to $OV(o_d)$.

Step 2. For each state variable v, a set P(v) of sites with v as FDSV is defined. P(v) and v are grouped into a candidate object o. If v belongs to one of the objects (already grouped), then P(v) is added to the object of v. That is, $(v \in OV(o) \wedge OP(o) = OP(o) \cup P(v)) \vee (\forall o{:}O \bullet v \notin OV(o) \wedge O = O \cup o \wedge OV(o) = OV(o) \cup \{v\} \wedge OP(o) = OP(o) \cup P(v))$, where $P(v{:}SV) = \{s{:}SITE \mid \forall s \bullet v \in FDSV(s)\}$. If v belongs to a candidate object o, then add P(v) to the object o. If v does not belong to any object, then add a new object (o), and add v and P(v) to the new object.

Step 3. Add operations with null FDSV as candidate objects. These candidate objects are: $\{s{:}SITE \mid FDSV(s) = \varnothing\}$.

Step 4. State Reference Graph (SRG) is built using candidate objects ({iv} and P(iv), where {iv} is the set of variables in a candidate object) as nodes and references of candidate objects of variables of other candidate objects as edges. Nodes in the SRG are placed in different levels; if a node A references node B (operations in A reference variables in B), then node B is placed at a lower level than node A. Root nodes at the bottom of the SRG represent objects that do not reference any variables. If nodes are circularly connected, then the nodes are merged into one node. SRG is a directed graph G(N,E) with nodes ≡ candidate objects (O) and edges ≡ $\{(c_1,c_2) \mid c_1, c_2 \in O \wedge v \in OV(c_1) \wedge P(v) \in OP(c_1) \wedge w \in OV(c_2) \wedge P(w) \in OP(c_2) \wedge w \in MUP(P(v))\}$. The SRG of the example in table 8.1 is shown in figure 8.1. In figure 8.1, the sp, base, and size nodes should be represented in one node as they belong to a data structure. Similarly, the front and back nodes should be represented in one node. However, to explain the

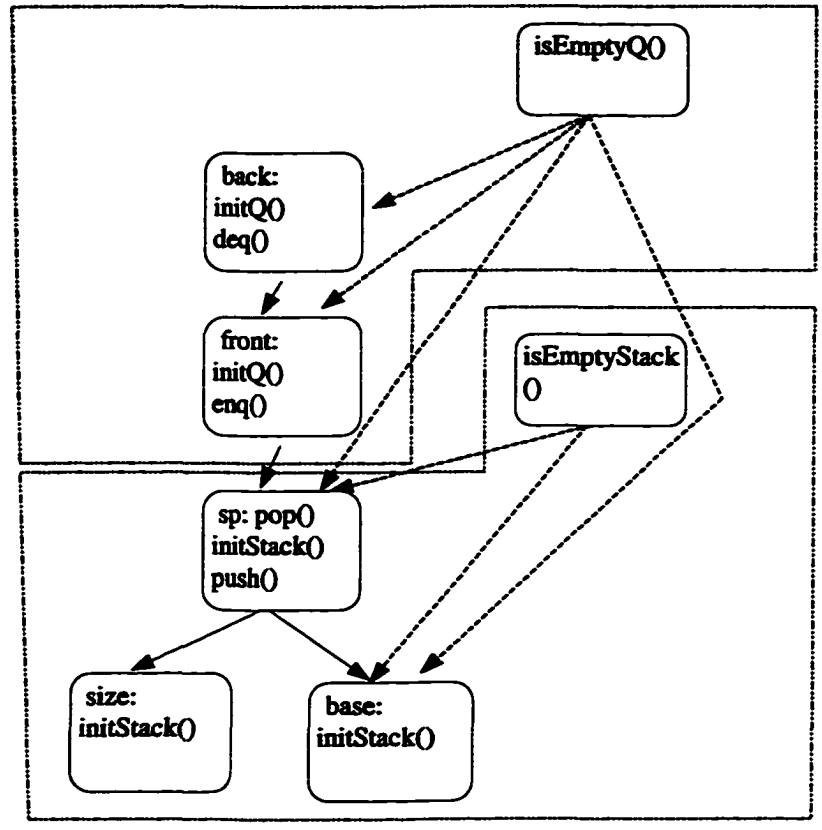


Figure 8.1. State reference graph

object identification process let us not consider the presence of data structures. In the figure, nodes with out-going dotted edges represent the sites with null FDSV (*isEmptyQ*, *isEmptyStack*).

Step 5. Connected nodes in the isolated sub-graphs of the SRG are merged with the help of the software engineer to fine-tune candidate objects. The SRG nodes can only be merged with the node at the highest level that is connected to it. Nodes of sites with null FDSV are eliminated from the graph if they are not merged with any other node. These eliminated nodes are due the presence of procedures that access several objects (e.g. main function in C programs). A node is merged with a lower level node with the help of the software engineer and with the use of the information related to the two nodes in

table 2, StDG, and SRG. Two groups are identified in the example, as shown with dotted borders in figure 8.1.

Nodes in each group are merged into one object: $c_1,c_2 \in O \wedge$ 'c_1 and c_2 are grouped' $\wedge P(v) \in OP(c_1) \wedge P(w) \in OP(c_2) \wedge v \in OV(c_1) \wedge w \in OV(c_2) \wedge OP(c_1) = OP(c_1) \cup OP(c_2) \wedge OV(c_1) = OV(c_1) \cup OV(c_2) \wedge O = O \setminus c_2$. If c_1 and c_2 are merged, add operations and variables in c_2 to c_1 and remove c_2 from O. We consider candidate objects in O (stack and queue in the example) as objects in the new system.

Step 6. If an operation of one object uses a variable of another object, then a selector operation is introduced in the second object. A selector operation is introduced in the object if these conditions are satisfied: $(c_1,c_2) \in E \wedge v \in OV(c_1) \wedge P(v) \in OP(c_1) \wedge w \in OV(c_2) \wedge P(w) \in OP(c_2) \wedge w \in USV(OP(c_1))$. That is, if there is an edge from c_1 to c_2 and an operation in c_1 uses a variable of c_2, a selector operation is introduced. Similarly, a modifier or an iterator operation is introduced if one object defines a variable of another. The condition for introducing a modifier (or iterator) operation: $(c_1,c_2) \in E \wedge v \in OV(c_1) \wedge P(v) \in OP(c_1) \wedge w \in OV(c_2) \wedge P(w) \in OP(c_2) \wedge w \in DSV(OP(c_1))$.

Step 7. In the StDG, if a site (s1) defining a control variable and sites (s2) connected to s1 are grouped into different objects, then duplicate s1 and add to each object that has a site connected to s1. The condition for site (operation) duplication: $(\tau_j,s1,Definition) \in DPS(s1) \wedge (\tau_j,s2,Use) \in UPS(s2) \wedge FDV(s1) = \tau_j \wedge s2 \in OP(o) \wedge s1 \notin OP(o) \wedge o \in O$. That is, s1 and s2 are connected by a control edge, and they belong to different objects. If this step results in site duplication, step 6 is repeated for the duplicated site.

8.3 Examples for Object Identification

8.3.1 Example 1

Table 8.3. Sample program for object identification

```
1.  int main(){
2.      long Fitems[MAX];
3.      long Bitems[MAX];
4.      int Fsp, Bsp, size=MAX,j;
5.      long it, jt;
6.      Fsp = Bsp = 0;
7.      for(j=0;j<size;j++){
8.              Fitems[j] = 0;
9.              Bitems[j] = 0;
10.     }
11.     ...
12.     if(Fsp < size)
13.             Fitems[Fsp++] = it;
14.     ...
15.     if(Fsp < size)
16.             Fitems[Fsp++] = it;
17.     ...
18.     if(Bsp)
19.             jt  = Bitems[Bsp-1];
20.     else if(Fsp)
21.             jt = Fitems[0];
22.     else jt = 0;
23.      ...
24.     if(Bsp)
25.             jt  = Bitems[--Bsp];
26.     else if(Fsp) {
27.             while(Fsp)
28.                     Bitems[Bsp++] = Fitems[--Fsp];
29.             jt = Bitems[--Bsp] ;
30.     }
31.     else jt = 0;
32.      ...
33.     if(Fsp < size)
34.             Fitems[Fsp++] = it;
35.     ....
36. }
```

To explain the object identification process we use the sample program shown in table 8.3. The RSG of the program is shown in figure 8.2. In the RSG, internal edges are not shown. Statements in the sites of the RSG are shown in table 8.4. From the RSG, we identify variables Fitems, Fsp, Bitems, and Bsp as candidates for state variables. Re-restructuring the StDG using the state variable information we have a RSG with statements in sites as shown in table 8.4.

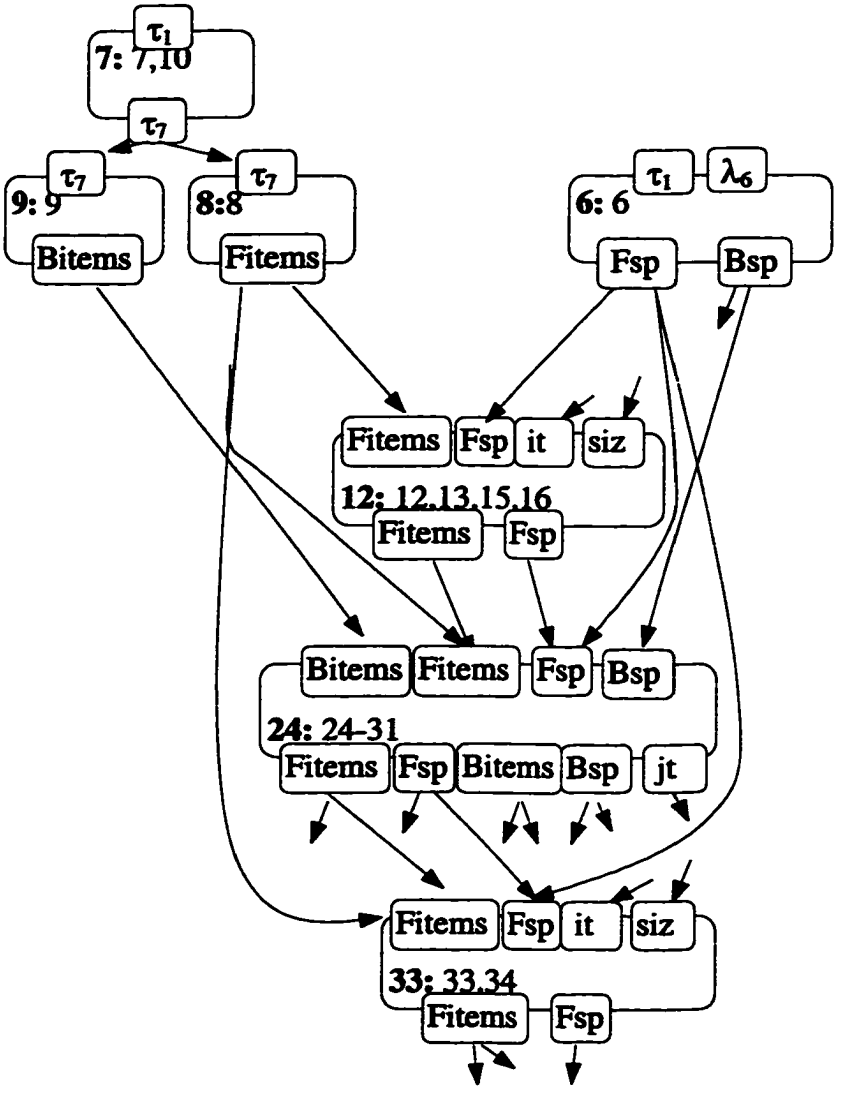


Figure 8.2. RSG of the program in table 8.3.

Table 8.4. Sites in the modified RSG.

```
Site 6:
6.  Fsp = Bsp = 0;
Site 7:
7.          for(j=0;j<size;j++){
10.         }
Site 8:
8.                  Fitems[j] = 0;
Site 9:
9.                  Bitems[j] = 0;
Site 12:
12. if(Fsp < size)
13.                 Fitems[Fsp++] = it;
Site 15:
15.         if(Fsp < size)
16.                 Fitems[Fsp++] = it;
Site 17:
17.         ...
18.         if(Bsp)
19.                 jt  = Bitems[Bsp-1];
20.         else if(Fsp)
21.                 jt = Fitems[0];
22.         else jt = 0;
23.     ...
Site 24
24.         if(Bsp)
Site 25:
25.                 jt  = Bitems[--Bsp];
Site 26:
26.         else if(Fsp) {
30.         }
Site 27:
27.                 while(Fsp)
28.                         Bitems[Bsp++] = Fitems[--Fsp];
Site 29:
29.                 jt = Bitems[--Bsp] ;
Site 31:
31.         else jt = 0;
Site 33:
33.         if(Fsp < size)
34.                 Fitems[Fsp++] = it;
```

State variables and operations in the program are:

SV = {Fitems, Fsp, Bitems, Bsp}
P = {6a, 6b, 7, 8, 9, 12, 15, 17, 24, 25, 26, 27, 29, 31, 33}
P(Fitems) = {8, 12, 15, 33}
P(Fsp) = {6a}
P(Bitems) = {9,27}
P(Bsp) = {6b, 25 ,29}

SRG of the program is shown in figure 8.3(a). Nodes in the SRG are merged as: {26, Fsp} (26 is a null FDSV site), {24, Bsp} (24 is a null FDSV site), {Bsp, Bitems} (connected nodes at the same level), and {Bsp, Bitems, 17} (17 is a null FDSV site). Nodes in the merged SRG are shown in figure 8.3(b). These nodes are further merged with the help of the software engineer. Assuming that the engineer merges nodes Fsp and Fitems, we have two objects. These objects and their operations are:

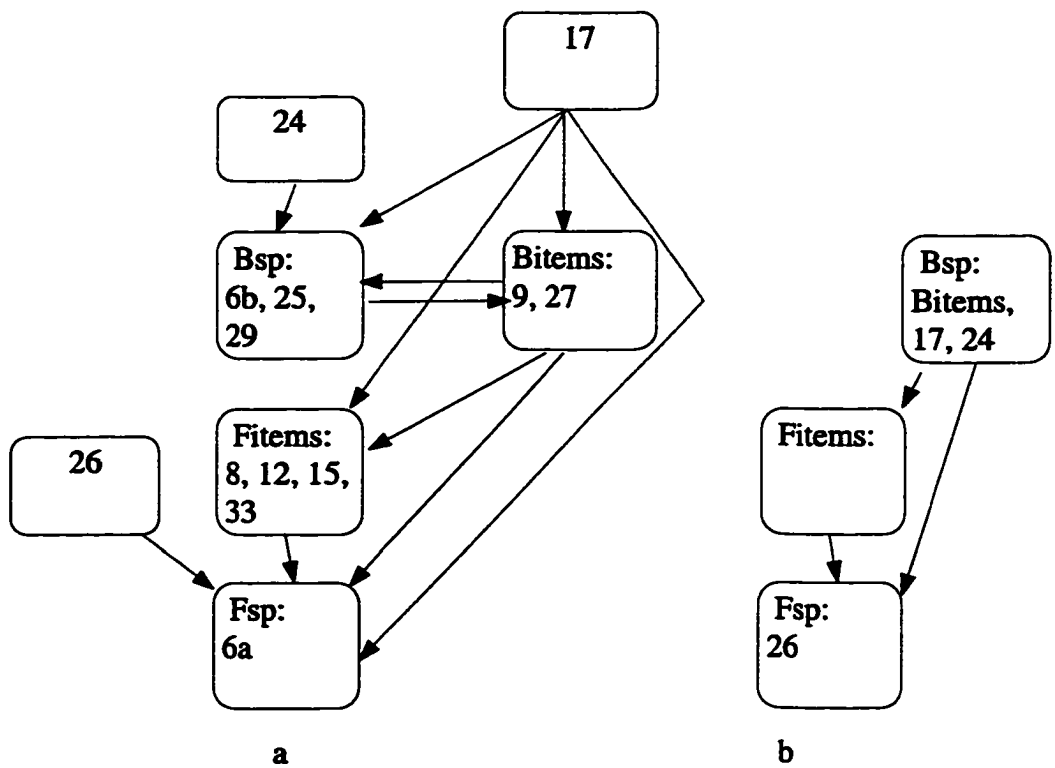


Figure 8.3. (a). SRG of the program. (b) Partially merged SRG.

Table 8.5. Sample program 2 for object identification.

```
1.  long Fitems[MAX];
2.  long Bitems[MAX];
3.  int Fsp, Bsp, size=0;
4.  push_item(long it){
5.  if(Fsp < size)
6.          Fitems[Fsp++] = it;
7.  }
8.  long next_item(){
9.   ling jt;
10. if(Bsp)
11.         jt  = Bitems[Bsp-1];
12. else if(Fsp)
13.         jt = Fitems[0];
14. else jt = 0;
15. return jt;
16. }
17. long get_next_item(){
18. long jt;
19. if(Bsp)
20.         jt  = Bitems[--Bsp];
21. else if(Fsp) {
22.         while(Fsp)
23.                 Bitems[Bsp++] = Fitems[--Fsp];
24.         jt = Bitems[--Bsp] ;
25. }
26. else jt = 0;
27. return jt;
28. }

29. int main(){
30.    int j;
31.    long it, jt;
32.    Fsp = Bsp = 0;
33. for(j=0;j<size;j++){
34.         Fitems[j] = 0;
35.         Bitems[j] = 0;
36. }
37.         ...
38. push_item(it);
39. ...
40. push_item(it);
41. ...
42. jt = next_item();
43.         ...
44. jt = get_next_item();
45. ...
46. push_item(it);
47. ....
48. }
```

OV(o1) = {Fitems, Fsp}
OP(o1) = {6a, 8, 12, 15, 26, 33}
OV(o2) = {Bitems, Bsp}
OP(o2) = {6b, 9, 17, 24, 27, 29}

8.3.2 Example 2

Let us consider the sample program in table 8.3 implemented differently, as shown in table 8.5. The RSG similar to the RSG in figure 8.2, with different site numbers. Applying the object identification process to the program results in identification of objects similar to the objects in example 2. Objects identified are shown in table 8.6.

8.4 Object Extraction

State variables and operations identified and grouped into objects are extracted by replacing the variables and operations by the objects using the following guidelines:

1. If the FDV of a modifier operation is a local variable, then add a return statement after each statement defining the local variable.
2. Operations are given appropriate names with the help of the programmer, and duplicate operations in each object are removed.
3. If an operation of one object uses a state variable of another object and the operations in the two objects are connected by an edge, then add the variable as a formal parameter. Or, if a variable used is defined in the operation, then declare the variable as a local variable.
4. For each site selected as an operation of an object, the site is replaced by a call statement to the operation.

Table 8.6. Operations in the objects identified

```
init_F(){/*Site 6 is separated into 6a & 6b */
6. Fsp = 0; /*as it has a multi-def variable*/
} /* site 6 – part */
init_B(){ /* site 6 – part */
6. Bsp = 0;
}
init_Fi(int j){ /*using - j */
7. for(j=0;j<size;j++){
8.              Fitems[j] = 0;
10. }
} /* site 8 */
init_Bi(int j){
7. for(j=0;j<size;j++){
9.              Bitems[j] = 0;
10. }
} /* site 9 */
push_F(long it){ /* using - it */
12.     if(Fsp < size)
13.             Fitems[Fsp++] = it;
}/* site 12, site 15, site 33 */
pop_B(){ /* defining non-SV */
        long jt; /* non-SV defined */
25.     jt = Bitems[--Bsp];
                return jt;
} /* site 25, site 29 */
F_to_B(){
27.             while(top_F())
28.                     Bitems[Bsp++] = pop_F();
}
pop_F(){
28.             return Fitems[--Fsp];
}
ptr_B(){ /* Bsp – used in sites not part of the object, also in 24 */
18. return Bsp;
} /* site 18, site 24 */
ptr_F(){ /* Fsp – used in sites not part of the object, also in 26 */
20. return Fsp;
} /* site 20, site 26 */
top_F(){
21.     return Fitems[0];
}/* site 21 */
top_B(){
19.     return Bitems[bsp-1];
}/* site 19 */
```

5. Statements in the operations are sorted in an ascending order.
6. Introduce constructors and destructors in each of the identified objects. Some of the modifier operations may initialize state variables; those initializations are separated and made constructor operation. For each object, a destructor operation may be introduced with the help of the programmer.
7. Sites with null FDSV, USV, and DSV are grouped with the sites that are connected to the sites under consideration. Site 31, in the example of table 8.4 is an example of a site with null USV and null DSV. If these sites are connected to sites that are grouped into several objects, then the sites are duplicated and used in all objects with connected sites. In table 8.4, site 7 is duplicated and used in both the objects. If these sites are not connected with any other site, then the sites are grouped with the sites that call these sites. The objects and their operations for the program in table 8.4 are:

   OV(o1) = {Fitems, Fsp}
   OP(o1) = {6a, 8, 12, 15, 26, 33, 7}
   OV(o2) = {Bitems, Bsp}
   OP(o2) = {6b, 9, 17, 24, 27, 29, 31, 7}

By applying the above suggestions to the sites in table 8.4 we get the operations shown in table 8.6. Operations are given appropriate names and duplicate operations are removed. The sites that an operation represents are indicated as comments in the operations. Constructors and destructors are not introduced. The objects and their operations are:

   OV(o1) = {Fitems, Fsp}
   OP(o1) = {init_F, init_Fi, push_F, pop_F, top_F, ptr_F}
   OV(o2) = {Bitems, Bsp}
   OP(o2) = {init_B, init_Bi, F_to_B, pop_B, top_B, ptr_B}

8.5 Case Studies

In this chapter, we present two case studies in which we apply the StDG to object recognition.

8.5.1 Bash

The legacy system we chose to study is Bourne Again SHell (bash), as it was also used by [Siff 99] and [Gira 97] for a similar purpose. It is a GNU project which is an interactive shell program with 38K lines of code. In table 8.7, we present the results of the analysis of a sub-system of bash (sbash).

Sbash uses four data structure: HASH_TABLE (HT), BUCKET_CONTENTS (BC), ASSOC (AS), and SHELL_VAR (SH). Sbash also declares a global variable ht of HT type. HT and BC have an aggregation relationship, and the HT contents are in the form of BCs. In the table '->' indicates the access (use or definition) of member variables of the type. OT and OT1 stands for other objects that are not part of sbash. The 'result' column indicates the object the procedure belongs to and objects in which a selector or modifier operation needs to be introduced. The 'miscellaneous' column indicates parameter and return types. The column also indicates objects that 'call' current procedure and the procedures of objects that current procedure 'called'. The SRG of the sbash is presented in figure 8.4. Four objects: HT, BC, AS, and ht are identified as part of sbash.

Comparison of results obtained using different approaches

a) Siff and Reps approach. Applying concept analysis approach [Siff 99] to sbash will result in identification of the four objects. But, object operations and

interfaces differ from our results. Object BC will have only 7 as its operation. Several objects access a data member of BC, hence they all must be declared as 'friend' in BC. AS will have 10, 11, 13, and 17 as operations, though 13 and 17 are call operations of HT. Hence, AS has to be a 'friend' of HT and HT a 'friend' of AS. Also, OT must be a 'friend' of HT.

Table 8.7. Analyses of sbash

File name	Procu dure #	Procedures		USV	DSV	Miscellaneous	Result
alias.c	1	initialize_aliases		ht	ht		ht
	2	find_alias		ht, BC->			ht, BC->
	3	get_alias_value		AS->		Call (ht)	ht, AS->
	4	add_alias	Site (1)	ht AS->	AS->	Call (ht)	ht AS->
	5		Site (2)	BC	BC-> AS->		BC AS->
	6	remove_alias		ht-> BC-> AS->			ht, HT-> BC-> AS->
	7	delete_alias_list		BC-> AS->		BC - parameter	BC AS->
	8	delete_all_aliases		ht->			ht HT->
	9	map_over_aliases		ht-> BC-> AS			ht HT-> BC->
	10	sort_aliases		AS		AS - parameter Call (AS)	AS
	11	qsort_alias_compare		AS->			AS
	12	all_aliases		ht, AS		Call (ht)	ht
	13	alias_expand_word		AS->		Call (ht)	ht AS->
	14	skipquotes				Called (ht)	ht
	15	skipws				Called (ht)	ht
	16	rd_token				Called (ht)	ht
	17	alias_expand		AS->		Call (ht)	ht AS->
Bashli ne.c	18	Command_word_co mpletion_function		OT1 AS->		Call, data	OT1 AS->

Table 8.7. Continued

File name	Procu dure #	Procedures	USV	DSV	Miscellaneous	Result
hash.c	19	intialize_hash_table	HT->	HT->	HT - parameter	HT
	20	make_hash_table		HT->	HT – parameter HT - return	HT
	21	xmalloc			call	HT
	22	hash_string	HT->			HT
	23	find_hash_item	HT-> BC->	BC->	HT – parameter BC - return	HT BC->
	24	remove_hash_item		HT-> BC->	HT – parameter BC - return	HT BC->
	25	add_hash_item		HT-> BC->	HT – parameter BC - return	HT BC->
	26	get_hash_bucket	HT->		HT – parameter BC - return	HT
variable.c	27	map_over	HT-> BC-> SV		HT – parameter SV - return call	HT BC->
	28	all_vars	OT HT SV		HT – parameter SV - return call	OT
	29	var_lookup	OT HT BC-> SV		HT – parameter SV - return call	OT BC->
	30	makunbound	OT HT BC-> SV		HT – parameter SV - return call	OT BC->
	31	kill_all_local_variables	OT HT SV->		HT – local SV - return Call, data	OT SV->
	32	delete_all_variables	OT HT-> BC-> SV	BC->	HT – parameter SV - return call	OT BC->
	33	make_var_array	OT HT SV->		HT – parameter SV - return call	OT SV->

b) CCM approach. The CCM technique [Canf 96] fails to identify AS, SV, and BC. AS and ht are merged into one object, and HT and BC are merged into one. Procedures 14, 15, 16, and 21 are not included in any object.

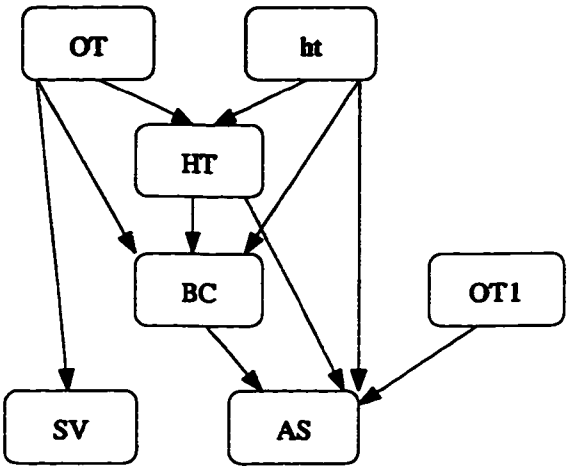


Figure 8.4. State reference graph of the sbash

c) Dunn and Knight, Liu and Wilde approaches [Liu 90][Dunn 93]. These approaches merge all procedures into one object. Only with the help of the software engineer can these approaches identify the objects.

In the bash sub-system there are seven objects: HT, BC, AS, ht, OT, OT1, and SV. Of these seven objects, HT, BC, AS, and ht are part of Sbash. As we can see in table 8.7, ht references HT, BC, and AS; HT references BC, and OT references HT, BC, and SV. If we use 'references' relationship to identify objects as used in the CCM, Dunn and Knight, and Liu and Wilde approaches all seven objects will be grouped into one object. The CCM approach will give better results by not recognizing AS, BC, and SV as candidate objects. In the absence of these objects the internal connectivity of HT and ht is greater than the connectivity between HT and ht; hence, HT and ht are recognized.

The Siff and Reps approach works using the information in the 'Miscellaneous' column. It uses information like has a parameter of type HT, uses HT, has a return type HT, and does not have a parameter of type HT. Information required for this approach can only be obtained through trial and error. Once the information is obtained, this

approach works better for recognizing objects. However, it fails to recognize the correct object operations, and the process is not comprehensive.

The objects and their operation identified by the StDG approach are shown in the 'Result' column of table 8.7. The object names followed by '->' indicate the violation of the object encapsulation and require introduction of a selector or a modifier operation in the respective objects. Slicing, proposed by several approaches, fails to recognize these violations properly. There are some procedures (e.g. 14-17) that do not reference any state variables. This problem is not addressed by any other approach. Procedure 17 calls a procedure that is grouped with ht, hence it is grouped with ht as this grouping will not violate any object encapsulation. Procedures 14 – 15 are called by ht and hence are grouped in it. This grouping may not be the most appropriate thing to do. A closer look at these procedures reveals that they manipulate strings.

8.5.2 Chull

Chull is a program taken from a computational geometry library that computes the convex hull of a set of vertices in the plane. It has three data structures: tVertex, tEdge, and tFace. *Chull* is also used in [Siff 99] for modularization. As in [Siff 99], we identify three objects, one for each data type. However, we differ with [Siff 99] in three ways. First, procedures 16 and 17 are identified with tVertex; whereas, we identified them with tFace. Second, procedure 20 is identified with tEdge, and we identified it with tFace. The third and most important difference is that [Siff 99] declares every object a 'friend' of every other object. Object tFace in [Siff 99] has only two operations, a constructor and a destructor. And the other two objects are also incomplete. The object operations we identified are shown in table 8.8. Objects identified in the StDG approach

represent true abstractions as all the operations required for the state changes are identified and encapsulated in the respective objects.

Table 8.8. Analyses of Chull.c

Procu dure #	Procedures	USV	DSV	Result
34	main			main
35	CleanUp			main
36	CheckEuler			main
37	PrintOut			main
38	MakeVertex		tVertex->	tVertex->
39	ReadVertices		tVertex->	tVertex->
40	Collinear	tVertex->		tVertex->
41	ConstructHull	tVertex->	tFace	tVertex
42	PrintPoint	tVertex->		tVertex
43	PrintVertices	tVertex->		tVertex
44	MakeEdge		tEdge->	tEdge
45	CleanFaces		tFace->	tFace
46	MakeFace		tFace->	tFace
47	CleanVertices		tVertex-> tEdge->	tVertex tEdge->
48	PrintEdges	tEdge-> tVertex->		tEdge
49	Volume6	tVertex-> tFace->		tFace tVertex->
50	Volumed	tVertex-> tFace->		tFace tVertex->
51	Convexity	tvertex-> tEdge->	tvertex-> tEdge->	tVertex tEdge->
52	PrintFaces	tvertex-> tFace->		tFace tvertex->
53	MakeCcw	tEdge->	tFace->	tEdge tFace->
54	CleanEdges		tEdge->	tEdge
55	Consistency	tEdge-> tFace->	tEdge	tEdge tFace->
56	Print	tVertex-> tEdge-> tFace->		tVertex-> tEdge-> tFace->
57	Tetrahedron		tVertex-> tEdge-> tFace->	tvertex tedge tFace
58 59	AddOne MakeStructs		tVertex-> tEdge-> tFace->	tVertex tEdge-> tFace->
60	Checks	tVertex-> tEdge-> tFace->		tVertex tEdge-> tFace->

8.6 Summary

In this chapter, we presented a new approach to object identification. We use the RSG for object state selection, the modified RSG for identifying object operations, and the SRG for identifying the objects. The SRG is a layered graph; nodes in the SRG are grouped into objects by merging the nodes in adjacent layers. The StDG approach stays within the system; as opposed to applying ad hoc methods [Siff 97]. The help of an engineer is necessary to control object granularity. Further, it is generally agreed that the reengineering process cannot be fully automated as object identification reflects a design decision that is inherently subjective.

Objects identified by the StDG approach are finer-grained than the approaches presented in Chapter 7. The work in [Canf 96][Liu 90][Newc 95][Siff 97][Yeh 95] represent the system as a graph, with functions, global variables, or function attributes as nodes. The StDG approach follows a similar approach, but it extends the graph nodes by including key local variables and program slices. Moreover, other approaches are only applicable to certain kinds of systems, unlike the StDG approach. The type of objects identified depends on the information present in the code. Hence the objects that are identified may require fine-tuning, like other approaches. Programmer knowledge of the problem can be easily incorporated into the approach.

Chapter 9

ReArchitect

9.1 Overview

Adequate system modifications and extensions depend on our ability to handle the system properties embedded in the source code. Other artifacts, such as design documents and users manuals, may be sufficient for a general understanding of the high-level system concepts; however, they provide insufficient details for actually changing the code. Instead, engineers rely on various representations and abstractions of the real system to understand the program, apply modifications, and analyze the effects of the changes. This research addresses these issues by defining a process based on a graph representation of the system. The software system ReArchitect was developed to automate the process. We present an overview of ReArchitect. Important features of ReArchitect include:

i. Program representation. ReArchitect uses the statement dependence graph (StDG) representation.

ii. Design extraction. ReArchitect extracts program control flow, data flow, and flow information. It then represents this information in the StDG. Other details about the program such as program variables and procedure calls are also extracted.

iii. Design restructuring. Cohesive components of the program must be considered in union in any maintenance application. ReArchitect restructures the StDG to

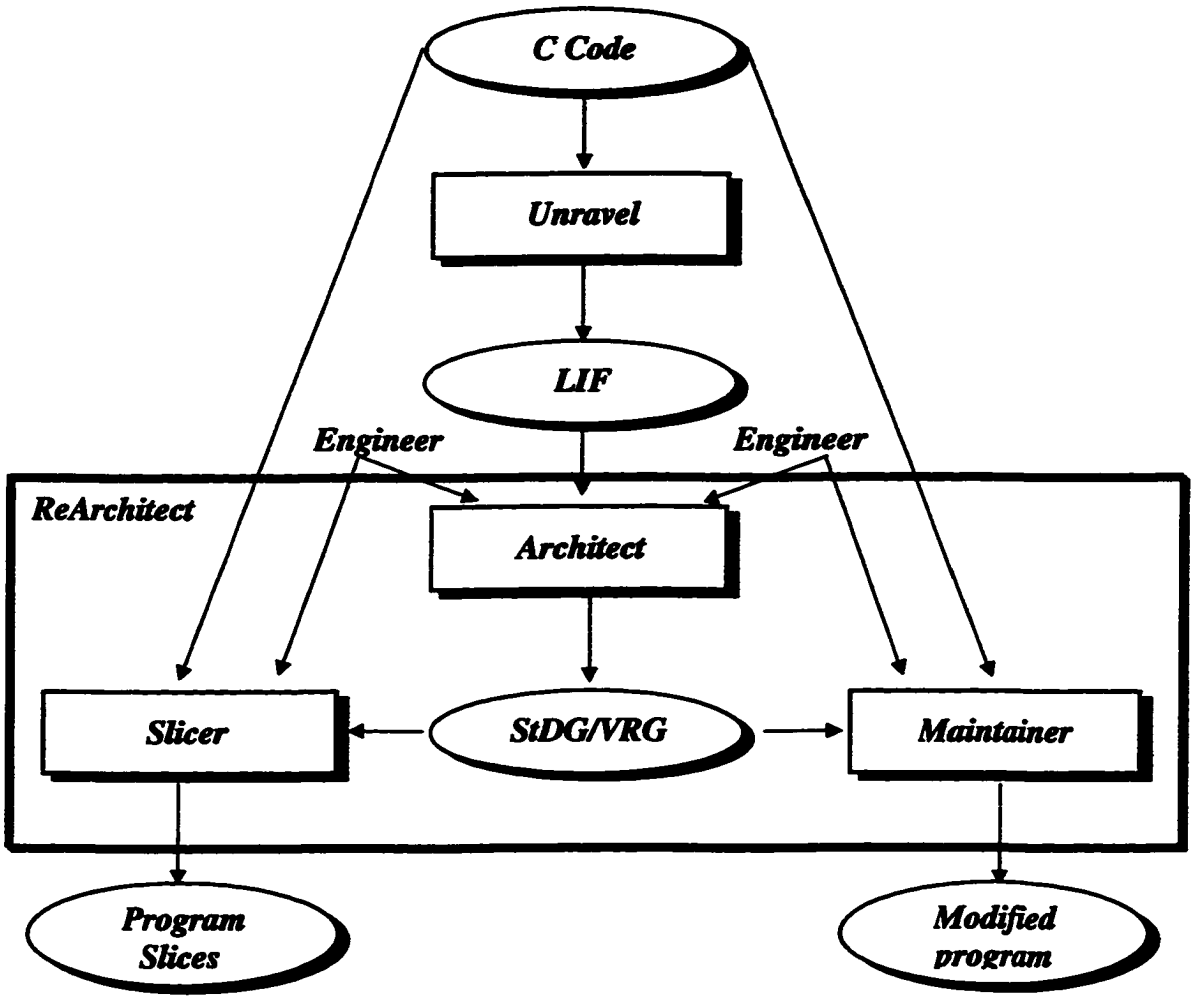


Figure 9.1. ReArchitect components and interactions

identify and merge these cohesive components. The restructured graph of the ReArchitect is similar to a type 2 RSGS.

iv. Design queries. The user can obtain the design information by interacting with the tool.

v. Applications. ReArchitect can be applied to several maintenance domains such as slicing, maintenance, and reengineering.

The main components of ReArchitect are shown in figure 9.1. Input to the ReArchitect is a C program in language independent format (LIF). LIF is an

intermediate representation generated by the Unravel tool (see Chapter 6). Unravel represents a C program as a flow graph in LIF format. An LIF representation of the wc program (figure 3.1) is presented in Appendix A with explanations of the codes used. ReArchitect converts the LIF representation of programs to StDG. A site in the StDG has a set of use ports and a set of def ports. A port has a set of use ports and a set of def ports connected to it. Sites in the StDG of the wc program, generated by the ReArchitect, are presented in Appendix A. A site has the following format.

```
Site 11:S:11,|
  Port(11,9,0,)TaU U{UPP1:TaU 9;DPP1-nw 11;}
  Port(11,11,0,)nw U{UPP1:nw 23;DPP1-nw 11;}
  Port(11,11,0,)nw D{UPP1:nw 11;UPP2:TaU 11;DPP1-nw 24;}
```

The site number of the site shown above is 11. The type of the site is given after the number. The type indicates whether the site is a block begin, a block end, a final use, or a loop statement. Sites represented by the site (in RSG) are listed following the 'S'. Ports in the site are indicated using the word 'Port', followed by the port information and its connections (use and def ports). Port information includes the site number of the port, the statement number, and level of indirection of a pointer variable (these are presented within the parenthesis). A use port is indicated with a 'U' and a def port with a 'D'. Use (UPP) and def (DPP) ports connected to the port are presented within the braces. UPP and DPP are give sequential numbers, and the variables these ports represent along with the site number of these ports are also indicated. We present design and implementation details of the ReArchitect in sections 9.2 and 9.3, respectively.

9.2 Design

ReArchitect consists of three main components: architect, slicer, and maintainer. The *architect* takes the LIF and generates the StDG. This generation involves extraction of dependence information in the form of control flow, data flow, and flow dependences. Extracted dependence information is incorporated in the StDG as edges. From the StDG, RSG is computed by restructuring it using structure compaction. The engineer can interact with the architect to obtain call graphs, lists of local and global variables, the lists of sites in the procedures, and summary sites of the procedures.

The *slicer* uses the RSG of the program to compute program slices. The slicer has a GUI interface for user interaction. The slicer can be used to obtain forward, backward, or modular slices. Slice criteria is specified as a group of variables and their site numbers. A forward or a backward slice includes the union of slices of each variable. A list of statements in the slice is given as output. For a modular slice, a set of ports as source and a set of ports as sinks form the slice criteria.

The *maintainer* uses the RSG of the program for maintenance activities such as code additions, changes, deletions, and code movement. The maintainer has a GUI interface for user interaction. For deleting a statement, its number is given as input to the maintainer. A list of statements that need to be deleted is the output. Input for code movement is a statement number. The maintainer lists the choices for moving the statement. A query for code additions is in the form of a list of variables (existing variables defined in the new statements) and the statement number where the new statements will be added. The result will indicate whether the new statements can be added at that location.

We used the Unified Modeling Language (UML) [Lee 97] for the representation of the design. An object-oriented analysis resulted in the identification of the objects and classes as shown in figure 9.2 and figure 9.3. Figure 9.2 includes generalization,

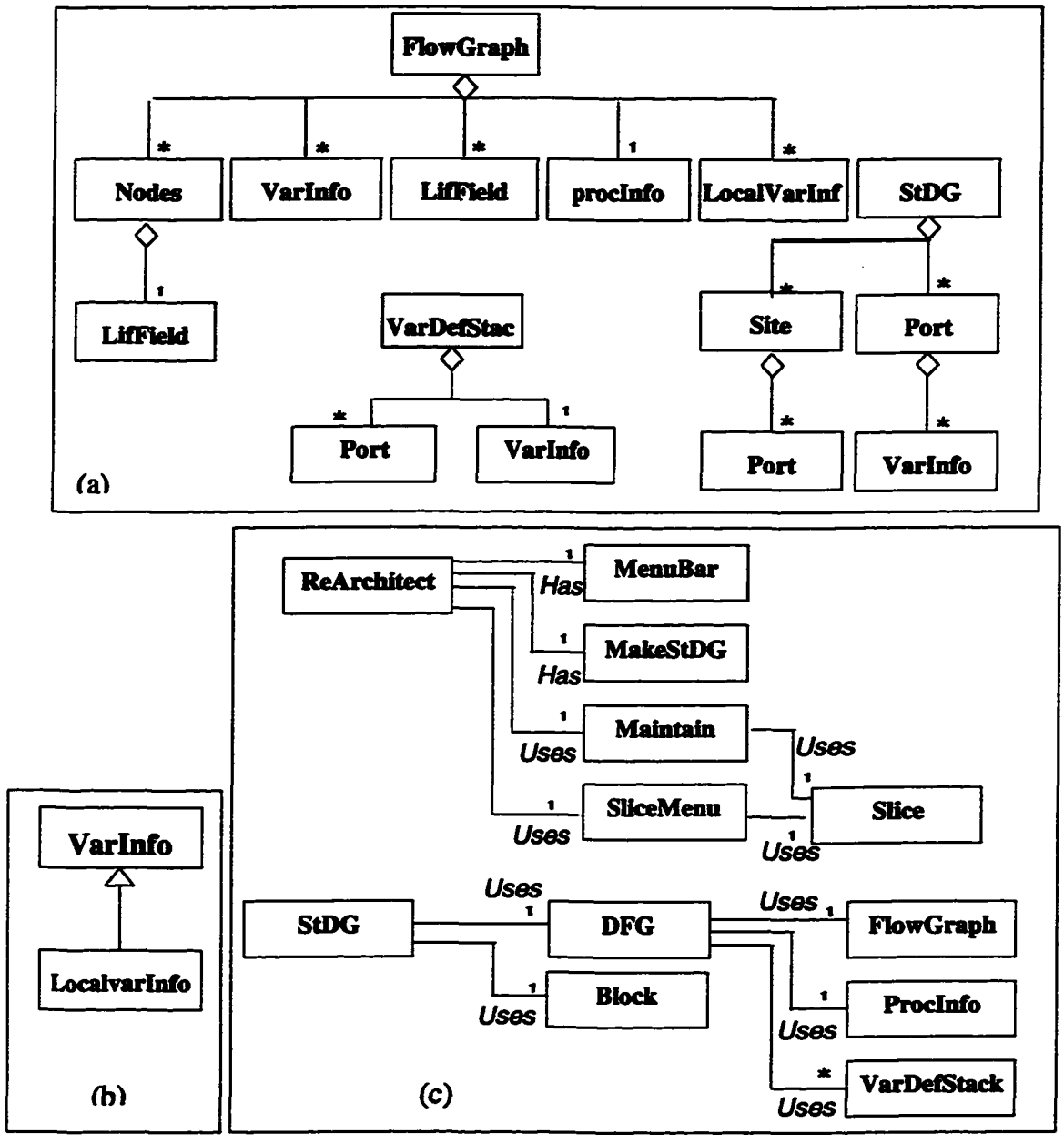
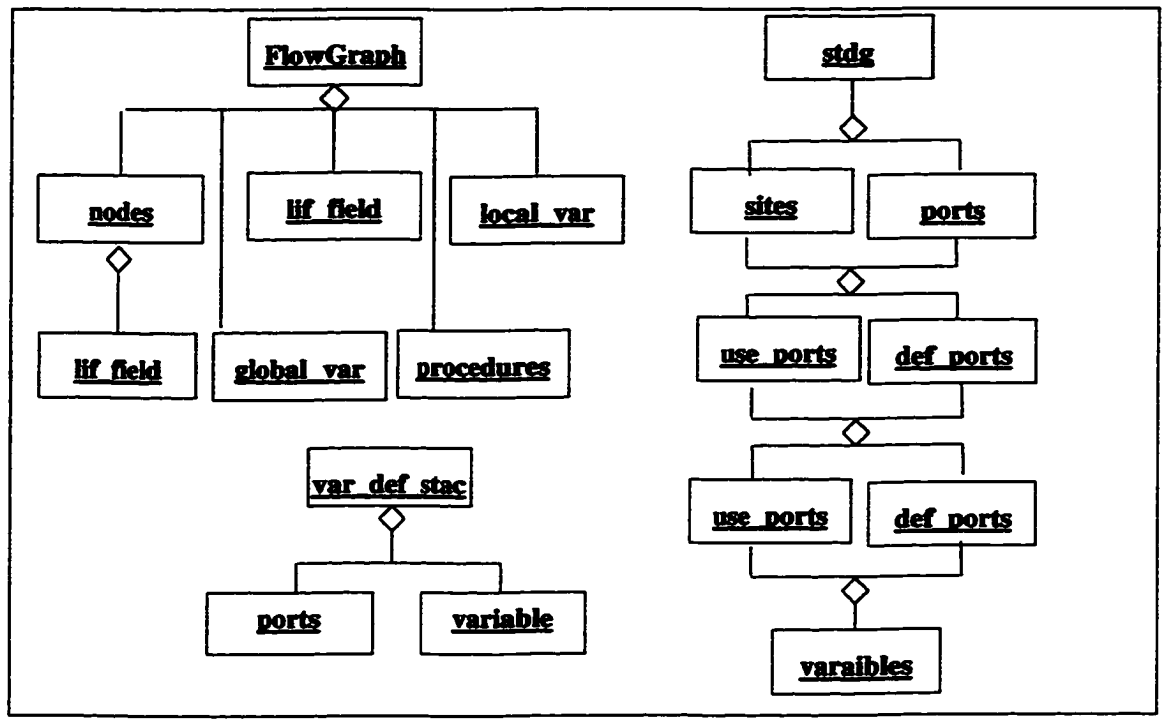


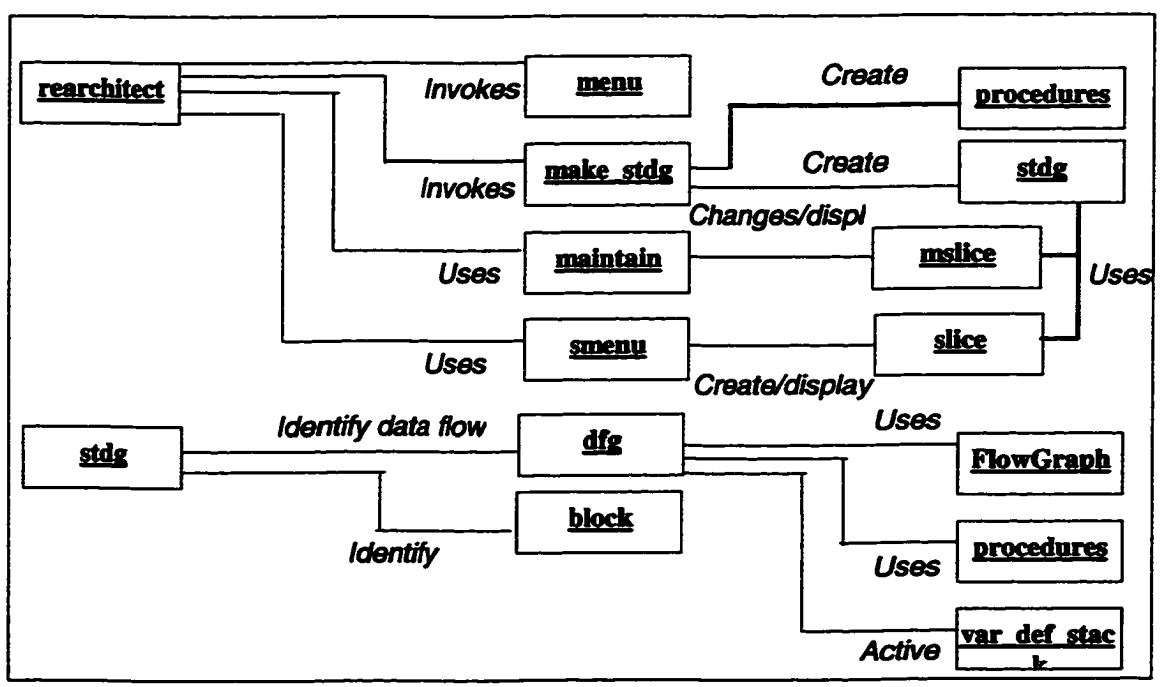
Figure 9.2. (a) ReArchitect class aggregation diagram
(b) Class generalization diagram (c) Class association diagram

aggregation, and association diagrams of the classes in ReArchitect. Figure 9.3 shows

object aggregations and association diagrams. These diagrams give an overview of ReArchitect system objects and their interaction.



(a)



(b)

Figure 9.3. (a) ReArchitect object aggregation diagram (b) association diagram

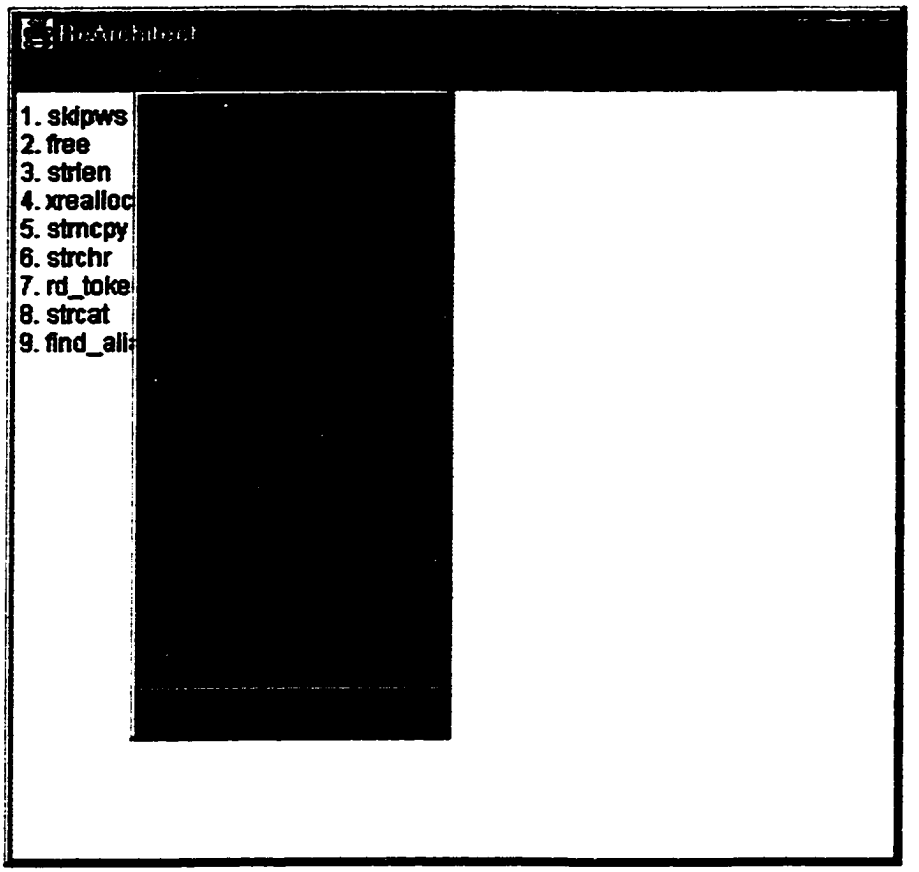


Figure 9.4. ReArchitect interface

9.3 Implementation

ReArchitect was developed on a Pentium machine running under the Windows NT operating system. The object-oriented programming language Java was used for coding. Figure 9.4 shows the ReArchitect graphical interface. The figure also shows the procedures called by the 'alias_expand' procedure (procedure selection is shown with a mark in the 'Process' sub-menu). All the procedures in the program are listed in the 'Process" sub-menu. Procedure summary sites are displayed as site specifications. In figure 9.5, the summary site of the procedure `wordCount` of `wc` program is given. It has three sets of ports, indicating the three internal edges in the summary site. In figure

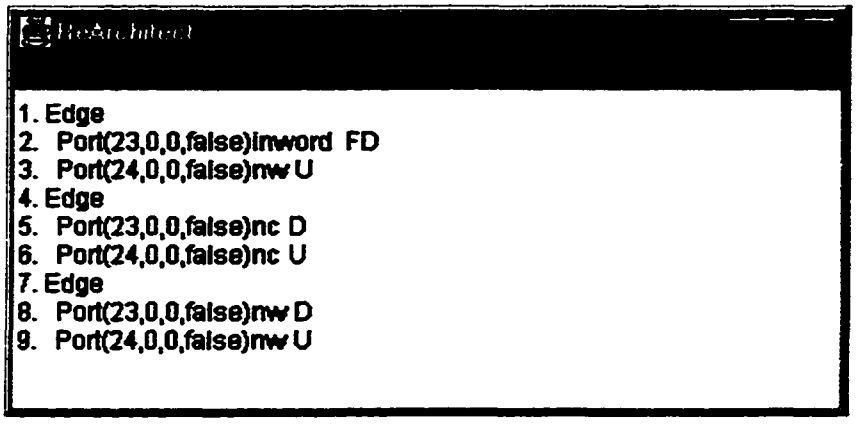


Figure 9.5. Summary site of procedure `wordCount`

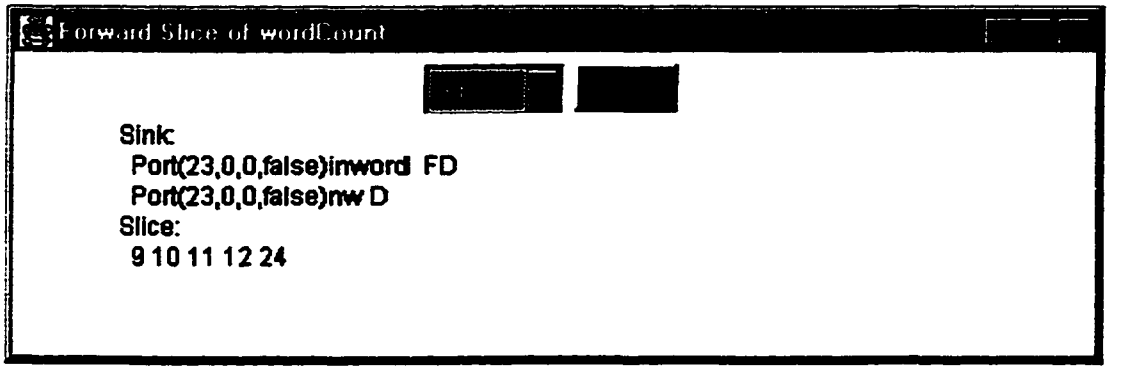


Figure 9.6. A forward slice on `nw` and `inword` at enter site

9.6, a forward slice on `nw` and `inword` at enter site (site 23) is shown. Site number 24 in the figure is the exit site of the procedure.

9.4 Discussion

In this section, we analyze the ReArchitect with respect to questions of scale, use in interactive environments, and limitations of static analysis. Scalability is an important property for any tool architecture. To be scalable, the internal data structure of the tool must grow (approximately) linearly with the size of the program being manipulated. A simple, small data structure enhances the architecture of any software application [Morg 97]. ReArchitect satisfies these conditions.

The data structure used in the StDG is based on the PDG models. Scalability aspects of these models have been shown in the literature, motivated by their use in compiler technologies. A comparison of StDG to PDG models is presented in section 5.2.4. StDG is a unified and discard type representation. It discards the LIF after building the StDG, thus eliminating the need for mapping functions as required in multiple and non-discard type representations.

Unlike the tools that use representations such as AST and some PDG models, StDG analyzes the independently compilable units of a system separately, thus dramatically reducing the memory requirements. Moreover, StDG uses a modular representation (summary site) for procedures. Hence, a procedure needs to be analyzed only once and can be stored in persistent data. A call to the procedure is replaced by its summary site. Hence, technically only one procedure StDG needs to be in memory for slicing and maintenance applications.

An interactive tool needs to conserve memory resources and yet quickly determine needed information. StDG is more than an all-inclusive representation. All-inclusive representations include all the dependence information in a representation, as opposed to the representations (e.g. AST) that derive the information when needed. StDG also includes the information on cohesive statements thus eliminating the need for individually analyzing these statements. Furthermore, the summary site representation eliminates the need for analyzing the procedures at every call statement.

Static analysis can only conservatively approximate data and control flow dependences between different program components. Though this approximation is not a major impediment to slicing and restructuring of programs, the quality of results

obtained can be improved with better analysis. Moreover, these limitations severely affect reengineering tools based on static analysis. StDG provides several features to help address this deficiency and to improve the quality of the results obtained. StDG is editable. Incorrect dependences in the StDG can be corrected manually. StDG can be made a component of the deliverable products which are handed off from the development team to the maintenance team.

Several approaches either assume the effects on data flow of calls to library or unknown functions, or they require dummy functions. Our approach requires the summary sites of these functions. Algorithms based on PDG must be modified to handle the individual language constructs, such as short circuiting in C. ReArchitect achieves language independence through the use of LIF, an intermediate representation. All language dependencies are handled during the program to LIF translation.

Some of the important features of the ReArchitect that distinguishes it from other such tools include:

- We know of no tool that allows constraints based changes to the program using a representation.
- Maintenance activities require changes to the representation to complete the process and for further processing. The representations generally must be re-derived from the program when the program is changed. The RSG based maintenance model is a semantically constrained maintenance process model that allows simultaneous updates to both the representation model and the program.

- General code move queries involve a boolean answer with origin and destinations of the move specified. In addition, ReArchitect can provide a range of choices where the code can be moved without changing the program meaning.
- In addition to the list of maintenance activities presented in section 5.3, other activities such as breaking a compound statement into two or merging two compound statements into one can be performed. Selected block (compound statement) members can be moved out of the block by duplicating the structure part (one site) of the block and by making the selected statements control dependent on the new site.
- Intelligent use of the statement numbering scheme facilitates the execution of all maintenance activities and restructuring transformations by comparing the site and port numbers of the statement under consideration, thus requiring no further processing.
- The statement numbering eliminates the need for any annotations needed for mapping the sites to the program, unlike other representations.

Chapter 10

Conclusions

10.1 Summary

The process of maintenance and enhancement of legacy software systems is a laborious and unavoidable task. This research was undertaken to seek systematic solutions to the maintenance problem and to provide automated support for the maintenance process. This research encompasses four major areas: program analysis and representation, program slicing, program maintenance, and reengineering.

We described a new representation known as Statement Dependence Graph (StDG). The StDG is a fine-grained PDG based discard type multiple representation with modular representation for functions and slices. The analysis approach used is exhaustive. Alias information is incorporated in the graph. We also provide algorithm for building the graph from C programs.

Cohesive components in the graph are merged using compactions. We define three types of compactions for merging the graph. These are data, structure, and edge compactions. Algorithms for identifying the candidate components for compaction using the graph connections and for merging the graph so as to reduce the size of the graph by eliminating duplicate parts are described.

The restructured graph RSG is useful in several maintenance domains such as slicing, maintenance and reengineering. Several different versions of slices are described in the literature. We describe three ways of restructuring the StDG to obtain different kinds of slices. The benefits of using different types of RSGs are also

presented. We also presented the application of RSG for maintenance activities using a constraints based approach. Using the RSG, we separate the program into change dependent and change independent parts. The change independent part is isolated, thus restricting the maintainer's attention to certain parts of the program.

Research in reengineering is mainly focused on clustering techniques that group procedures in a legacy system into candidate objects. We described a comprehensive and systematic process for reengineering legacy systems. Reengineering involves reverse engineering of the design, making changes to the design, and forward engineering.

An intermediate representation known as the language independent format (LIF) is used by the Unravel CASE tool for slicing. We use Unravel for converting C programs to LIF. We presented algorithms for converting LIF to StDG. The conversion process involves derivation of data, control, and flow dependences from the LIF, extraction of call graph, and other variable usage information. We also showed how the reverse engineered design is presented to the user. In place of a procedure call, its modular representation (summary site) is used in the representation. We give an algorithm for computing the summary site.

Code in procedural programs exists as interleaved code. We show how the interleaved can be localized and used in identifying object operations and state variables. Candidate operations, state variables, and data structures present in the system along with a call graph and the RSG are used to build a state reference graph (SRG) for object identification. Objects are identified and extracted. The procedural code can then be translated into object-oriented code. The help of the software engineer is elicited

when necessary during object identification. We described various object identification techniques available in the literature and discussed the pros and cons of these techniques. These reengineering techniques were compared using a case study, and the results were presented.

The reengineering process we presented is incorporated into a tool, ReArchitect. ReArchitect has six key components: static analysis and design extraction, program and design representation, design change, slicing, maintenance, and engineer interactions. The tool extracts program design from the LIF representation and represented it as StDG. StDG is restructured using structure compaction. The restructured graph (RSG) is used in slicing and maintenance activities. The tool performs favorably to other slicers as the RSG used by the tool requires no dependence analysis during slicing. Maintenance activities are unique to the ReArchitect. We analyzed the ReArchitect with respect to questions of scale, use in interactive environments, and how it overcomes the limitations of static analysis.

10.2 Contributions

This research encompasses four major areas: program analysis and representation, program slicing, program maintenance, and reengineering. It makes significant contributions to each of the four areas. This research:

- introduces a new program representation StDG that addresses several issues neglected to date. Salient features of this representation include:
    - ✓ Understandability – static analysis can only conservatively approximate dependences; the engineer can correct the dependences.

- ✓ Editability – program changes can be incorporated in the representation, thus eliminating the need for re-derivation of the representation whenever the program is changed.
- ✓ Hybrid analysis – dependences among cohesive statements are discarded and are computed on demand when and if needed. This feature eliminates the need for caching techniques used by other algorithms to manage memory resources.
- ✓ Alias information – such information can be incorporated into the StDG. Other tools use a separate representation for presenting alias information.
- ✓ Modular representation – this representation allows merging of different components.

- defines algorithms for converting C to StDG and LIF to StDG. The use of LIF as an intermediate representation gives the StDG applications language independence, as a program written in any language can be transformed to the LIF format.
- describes methods for identifying and merging cohesive components of the graph, simultaneously reducing the graph size. We know of no other analysis process that employs this innovative idea.
- presents three types of restructurings for obtaining different kinds of slices. Several kinds of slices are defined in the literature, each of these are computed in a different way. By restructuring the graph differently for different slices we can improve the slicing process.

- obtains modular slices, where slice criteria can be given as a function signature. Modular slicing is beneficial in building reuse libraries.
- defines a constraints based maintenance model for efficient program maintenance and restructuring transformations. The maintenance process is often done in an ad hoc manner. This research provides a systematic approach to making maintenance and structural changes to programs. Moreover, the process is scalable and can be performed at interactive speeds.
- defines a comprehensive method for identifying and extracting objects. The definition of the processes and the associated algorithms advance the state of reengineering research. Unlike approaches that group procedures into objects, this research presents a systematic process that identifies or introduces object-oriented principles in procedural programs. Systems are analyzed at the statement level.
- facilitates the derivation of the up to date design documents.
- includes an automated tool, called ReArchitect, that demonstrates the feasibility of the StDG for slicing and maintenance applications.

10.3 Future Research

Several extensions can be incorporated into the work. Some of the important areas are:

- The LIF to StDG translation process spends a significant time inferring the nodes that start a loop in the flow graph. It would be more efficient to change

the Unravel parser YACC grammar to mark the loop nodes for future identification. Unravel code is freely distributed by the system developers.

- Sometimes it is necessary to build the StDG of a select group of statements, for instance, to find the internal dependencies among statements that are merged during compaction. A process for building an StDG selectively is desirable. Moreover, if such a process were available we could be less conservative during compaction.
- During object identification, the StDG is re-restructured with the state variable information. It would be more efficient to selectively restructure the statements of the sites that are affected by the new information instead of restructuring the entire StDG.
- ReArchitect can be improved. It needs to be extended to incorporate facilities to build the state reference graph and to incorporate the object identification process. We have presented guidelines for object extraction by incorporating these guidelines. The ReArchitect can be extended to include automatic translation of code to object-oriented code.
- Finally, an all-inclusive case tool with slicing, maintenance, and reengineering capabilities, along with data management and storage facilities would be a valuable maintenance aid.

Bibliography

[Ache 95] Achee, B.L, Carver, D.L., "Identification and Extraction of Objects from legacy Code," In Proceedings of the Conference on Aerospace Applications, Colorado, USA (February 1995), pp. 181-190.

[Aho 86] Aho, A.V., Sethi, R, Ullman, J.D., *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

[Atki 96] Atkinson, D.C., Griswold, W., "The Design of Whole-Program Analysis Tools," In Proceedings of the 18th International Conference on Software Engineering, (1996), pp. 16-27.

[Benn 92] Bennett., K., Bull, T., Yang, H., "A Transformation System for Maintenance – Turning Theory into Practice," In Proceedings of the Conference on Software Maintenance, Orlando, FL (November 1992), pp. 146-155.

[Bigg 94] Biggerstaff, T.J., Mitbander, B.G., Webster, D.E., "Program Understanding and the Concept Assignment Problem," Communications of the ACM, Vol. 37, No.5 (May 1994), pp. 72-83.

[Booc 94] Booch, G., *Object Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Bran 96] Brand, M., Klint, P., Verhoef, C., "Re-engineering needs Generic Programming Language Technology," SIGPLAN Notices, Vol. 32, No. 2 (February 1997), pp. 54-61.

[Canf 93] Canfora, G., Cimitile, A., Munro, M., "Extracting Abstract Data Types from C Programs: A Case Study," In Proceedings of the Conference on Software Maintenance, Montreal, Canada (September 1993), pp. 200-209.

[Canf 96] Canfora, G., Cimitile, A., Munro, M., "An Improved Algorithm for Identifying Objects in Code," Software-Practice and Experience, Vol. 26(1) (January 1996), pp. 25-48.

[Chen 90] Chen, Y., Nishimoto, Y., Ramamoorthy, C.V., "The C information Abstraction," IEEE, pp. 325-334.

[Chik 90] Chikofsky, E.J., Cross II, J.H., "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, Vol. 13, No. 1 (January 1990), pp.13-17.

[Choi 90] Choi, S.C., Scacchi, W., "Extracting and Restructuring the Design of Large Systems," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 66-71.

[Chu 92] Chu., W.C., Patel, S., "Software Restructuring by Enforcing Localization and Information Hiding," In Proceedings of the Conference on Software Maintenance, Orlando, FL (November 1992), pp. 165-171.

[Dues 95] Duesterwald., E., Gupta, R., Soffa, M.L., "Demand-driven Computation of Interprocedural Data Low," Symposium on Principles of Programming Languages, San Francisco, California (January 1995).

[Dunn 93] Dunn, M.F., Knight, J.C., "Automating the Detection of Reusable Parts in Existing Software," In Proceedings of the International Conference on Software Engineering, Baltimore, Maryland, (1993), pp. 381-390.

[Ferr 87] Ferrante, J., Ottenstein, J.K., Warren, J.D., "The Program Dependence Graph and its Use in Optimization," ACM Transactions on Programming Languages and Systems, 9(3), (July 1987), pp. 319—349.

[Gall 95] Gall, H., Klosch, R., "Finding Objects in Procedural Programs: An Alternate Approach," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 208-216.

[Gall 91] Gallagher, K.B., Lyle, J.R., "Using Program Slicing in Software Maintenance," IEEE Trans. On Software Engineering, Vol 21, No 4 (August1991), pp. 751-761.

[Glen 92] Glenn, O., "Chembench: Redesign of a Large Commercial Application Using Object Oriented Techniques," OOPSLA, Vancouver, Canada (October 1992), pp. 13-16.

[Gris 91] Griswold, W.G., "Program Restructuring to Aid Software Maintenance," Ph.D. dissertation, University of Washington, Dept. of Computer Science & Engineering (1991) Tech. Report No. 91-08-04.

[Gris 93] Griswold, W.G., "Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool," In ACM SIGSOFT Symposium on the Foundations of Software Engineering, Software Engineering Notes (December 1993), 18(5), pp. 42--55.

[Gris 95] Griswold, W.G., Notkin, D., "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool," IEEE Trans. On Software Engineering, Vol 21, No 4 (April 1995), pp. 275-287.

[Harr 95] Harrold, M.J., Soffa, M.L., "Efficient Computation of Interprocedural Definition-Use Chains," ACM Transactions On Programming Languages and Systems, Vol. 16, No 2 (March 1994), Pages 175-204.

[Harr 93] Harrold, M.J., malloy, B., "A Unified Interprocedural Program Representation for a Maintenance Environment," IEEE Transactions On Software Engineering, Vol 19, No 6 (June 1993), pp. 584-593.

[Horw 90] Horwitz, S., Reps, T., Binkley, D., "Interprocedural Slicing Using Dependence Graphs," ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1 (January 1990), pp. 26-60.

[Horw 88] Horwitz, S., Prins, J., Reps, T., "Integrating non-interfering Versions of Programs," Fifth ACM Symposium on Principles of Programming Languages, San Diego, California, (January 1988), pp. 133-145.

[Huch 85] Huchens, D.H., Basili, V.R., "System Structure Analysis: Clustering with Data Bindings," IEEE Trans. On Software Engineering, Vol 11, No 8 (August 1985), pp. 749-757.

[Jack 94] Jackson, D., Rollins, E.J., "A New Model of Program Dependences for Reverse Engineering," Software Engineering Notes, SIGSOFT, ACM Press, Vol. 19, No. 5 (December 1994), pp. 2-10.

[Jack 97] Jacky, J., *The Way of Z Practical Programming with Formal Methods*, Cambridge University Press.

[Jaco 91] Jacobson, I., Lindstrom, F., "Re-engineering of Old Systems to an Object-Oriented Architecture," OOPSLA, (1991), pp. 340-350.

[Jarz 95] Jarzabek, S., Keam, T.P., "Design of a Generic Reverse Engineering Assistant Tool," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 61-70.

[Jian 91] Jiang, J., Zhou, X., Robinson, D.J., "Program Slicing for C - The Problems in Implementation," IEEE Conference On Software Maintenance, Sorrento, Italy, (October 1991), pp. 182-189.

[John 85] Johnson, W.L., Soloway, E., "PROUST: Knowledge-Based Program Understanding," IEEE Trans. On Software Engineering, Vol 11, No 3 (September 1985).

[Kern 88] Kernighan, R., Ritchie, D., *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice hall (1988).

[Kinl 94] Kinloch, D.A., Munro, M., "Understanding C Program Using Combined C Graph Representation," In Proceedings of the Conference on Software Maintenance, Victoria, Canada, (September 1994), pp. 172-180.

[Land 92] Landi, W., "Undecidability of Static Analysis," ACM Letters on Programming Languages and Systems, Vol. 1, No. 4 (December 1992), pp. 323-337.

[Lanu 93] Lanubile, F., Visaggio, G., "Function Recovery based on Program Slicing," In Proceedings of the Conference on Software Maintenance, Montreal, Canada (September 1993), pp. 396-404.

[Lee 97] Lee, C.R., *UML and C++ a Practical Guide to Object-Oriented Development*, Prentice Hall, New Jersey, and (1997).

[Lehm 85] Lehman, M.M., Belady, L., *Program Evolution Process of Software Change*, Academic Press, London, (1985).

[Lind 97] Linding, C., Snelting, G., "Assesing Modular Structure of Legacy Code Based on Mathematical Concept Analysis," In Proceedings of the 19th International Conference On Software Engineering, (1997), pp. 349-359.

[Liva] Livandas, E.L., "System Dependence Graphs Based on Parse Trees and their Use in Software Maintenance", Web site of Univ. of Florida, pp. 1-24.

[Liu 90] Liu, S., Wilde, N., "Identifying objects in a Conventional Procedural Language: An Example of Data Design Recovery," In Proceedings of the Conference on Software Maintenance, (November 1990), pp. 266-271.

[Lyle 84] Lyle, J.R., "Evaluating Variations of Program Slicing", Ph.D. Dissertation, Univ. of Maryland, College Park, (December 1984).

[Lyle 86] Lyle, J.R., Weiser, M.D., "Automatic Program Bug Location by Program Slicing", In Proceedings of the 2nd International Conference On Computers and Applications (Peking, China), June 1987, pp. 877-882.

[Lyle 95] Lyle, J.R., Wallace, D.R., Graham, J.R., Gallagher, K.B., Poole, J.P., Binkley, D.W., "Unravel: A CASE Tool to Assist Evaluation of High Integrity Software", U.S. Deprt. Of Commerce, Technology Administration, NIIT, (1995).

[Morg 97] Morgenthaler, J.D., "Static Analysis for a Software Transformation Tool", Ph.D. Dissertation, University of California, San Diego (1997).

[Newc 95] Newcomb, P., Kotik, G., "Reengineering Procedural into Object-Oriented Systems," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 237-249.

[Newc 95] Newcomb, P., "Legacy System Cataloging Facility," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 52-60.

[Ning 94] Ning, J. Q., Engberts, A., Kozaczynski, W., "Automated Support for Legacy Code Understanding," Communications of the ACM, Vol. 37, No. 5 (May 1994), pp. 50-57.

[Opdy 92] William F.O., "Refactoring Object-Oriented Frameworks". Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, (1992).

[Osbo 90] Osborne, W.M., Chikofsky, E.J., "Fitting Pieces to the Maintenance Puzzle," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 11-12.

[Ott 92] Linda, M.O., Jeffrey, J.T., "The Relationship between Slices and Module Cohesion," Communications of the ACM, Vol. 37, No. 5 (May 1989), pp. 198-204.

[Ott 89] Ott, M.L., Thuss, K., Wills, L.M., "The Relationship between Slices and Module Cohesion." Proceedings of the 11th International Conference on Software Engineering, Singapore, (May 1989), pp. 198-204.

[Otte 84] Ottenstein, K.J., Ottenstein, L.M., "The Program Dependence Graph in a Software Development Environment." In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (April 1984) pp. 177-184.

[Pent 98] Penteado, R., Masiero, P.C., Padro, A.F., Braga, R.T., "Reengineering of Legacy Systems based on Transformation Using the Object-Oriented

Paradigm," In Proceedings of the Fifth Working Conference on Reverse Engineering, Honolulu, USA (October 1998), pp. 144-153.

[Plat 91] Platff, M., Wagner, M., "An Integrated Program Representation and Toolkit for the Maintenance of C programs," International Conference on Software Maintenance, Sorrento, Italy, (October 1991), pp. 129-136.

[Pres 92] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill , Inc., Third Edition, (1992).

[Quil 94] Quilici, A., "A Memory-Based Approach," Communications of the ACM, Vol. 37, No. 5 (May 1994), pp. 84-93.

[Reps 95] Reps, T., Horwitz, S., Sagiv, M., "Precise Interprocedural Dataflow Analysis via Graph Reachability". In Conference Record of the 22nd ACM Symposium on Principles of Programming Languages, (1995) pp. 49-61.

[Rich 90] Rich, C., Wills, L.M., "Recognizing a Program's Design: A Graph-Parsing Approach," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 82-89.

[Ruga 90] Rugaber, S., Ornburn, S.B., LeBlanc, Jr., R.J., "Recognizing Design Decisions in Programs," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 46-54.

[Ruga 95] Rugaber, S., Stirewalt, K., and Wills, L.M., "The Interleaving Problem in Program Understanding." International Conference on Software Maintenance, Nice, France (October 1995), pp. 265-274.

[Rumb 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy., F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall, (1991).

[Sahr 97] Sahraoui, H.A., "Applying Concept Formation Methods to Object Identification in Procedural Code," Technical Report CRIM-97/05-77, CRIM, 1997.

[Scha 96] Schach, S.R., *Classical and Object-Oriented Software Engineering*, 3rd Edition, Irwin, 1996.

[Seba 93] Sebesta, R.W., *Concepts of Programming Languages*, Benjamin/Cummings, Second Edition, 1993.

[Siff 96] Siff, M., Reps, T., "Program Generalization for Software Reuse: From C to C++," Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, California, (October 1996), pp. 135-146.

[Siff 97] Siff, M., Reps, T., "Identifying Modules Via Concept Analysis," International Conference on Software Maintenance, Bary, Italy, (October 1997)

[Snee 95] Sneed, M.S., Nyary, E., "Extracting Object-Oriented Specification from Procedurally Oriented Programs" In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 217-226.

[Snee 98] Sneed, M.H., "Architecture and Function of a Commercial Software Reengineering Workbench" In Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy (1998), pp. 2-10.

[Stee 96] Steensgaard, B., "Points-to Analysis in Almost Linear Time," In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1996), pp. 32-41.

[Snel 96] Snelting, G., "Reengineering of Configurations based on Mathematical Concept Analysis," ACM Transactions on Software Engineering and Methodology, Vol 5, No 2 (April 1996), pp. 146-189.

[Thom 84] Thomas, A.S., "An Essay on Software Reuse," IEEE Trans. On Software Engineering, Vol 10, No 5 (September 1984), pp. 494-497.

[Vala 98] Valasareddi, R.R, Carver, D.L., "A Graph-Based Object Identification Process for Procedural Programs," In Proceedings of the Fifth Working Conference on Reverse Engineering, Honolulu, USA (October 1998), pp. 50-58.

[Vala 99] Valasareddi, R.R, Carver, D.L., "A Representation Model for Procedural Program Maintenance," In Proceedings of the ACM Symposium on Applied Computing, San Antonio, USA (February 1999), pp. 580-585.

[Vane 89] Vanek, L.I., "Static Analysis of Program Source Code Using EDSA," In Proceedings of the Conference on Software Maintenance, (1989), pp. 192-199.

[Venk 95] Venkatesh, G.A., "Experimental Results from Dynamic Slicing of C Programs," ACM Transaction on programming languages and systems, Vol. 17, No. 2 (March 1995), pp. 197-216.

[Ward 93] Ward, M.P., Bennett, K.H., "A practical Program Transformation System for reverse engineering," In Proceedings of the Conference on Software Maintenance, Montreal, Canada (September 1993), pp. 212-221.

[Wate 94] Waters, R.C., Chikofsky, E., "Reverse Engineering: Progress Along Many Dimensions," Communications of the ACM, Vol. 37, No. 5 (May 1994), pp. 22-25.

[Weis 84] Weiser, M., "Program Slicing," IEEE Transactions on Software Engineering," Vol. SE-10 No. 4 (July 1984), pp. 352-357.

[Weis 94] Weise, D., Crew, R.F., Ernst, M., Steensgaard, B., "Value Dependence Graphs: Representation without Taxation." Technical Report MSR-TR-94-03, Microsoft research, Redmond, WA, (April 94).

[Wigg 97] Wiggerts, T.A., Baxter, I., Quilici, A., Verhoef, C., "Using Clustering Algorithms in Legacy Systems Remodularization," In Proceedings of the Fourth Working Conference on Reverse Engineering, Amsterdam, Netherlands (October 1997), pp. 33-43.

[Wood 95] Woods, S., Yang, Q., "Program Understanding as Constraint Satisfaction" In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 314-323.

[Wood 98] Woods, S.G, Quilici, A.E., Yang, Q., *Constraint-Based Design Recovery for Software Reengineering*, Kluwer Academic Publishers, Boston (1998).

[Yeh 95] Yeh, A.S., Harris, D.R., Reubenstein, H.B., "Recovering Abstract Data Types and Object Instances from a Conventional Procedural Languages," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 227-236.

Appendix

Program Representations in ReArchitect

A.1 wc program formatted for StDG analysis

```
1  int nw, nc;
2  void wordCount(int inword) {
3    int c;
4    c = getchar ();
5    while ( c != EOF ) {
6      nc = nc + 1;
7      if ( c == ' '||c == '\n'||c == '\t' )
8        inword = 0;
9      else if ( inword == 0 ) {
10       inword = 1;
11       nw = nw + 1;
12     }
13   c = getchar ();
14   }
15 }
16 main() {
17   int inw;
18   inw = nc = nw = 0;
19   wordCount(inw);
20   printf ("\n") ;
21   printf (" %d %d\n",nc,nw ) ;
22 }
```

A.2.2 LIF representation of the wc program with explanations

```
18(1,2,6,2,14)   source for node 1 line 2 cols 6-14
1(1,1,wordCount)  Function wordCount 1 entry at 1
8(2,2)  local def to c (2) at stmt 2 on line 4
11(2,2)   call to getchar at 2
13   end call to getchar at 2
18(2,4,4,4,18)   source for node 2 line 4 cols 4-18
10(3,2)  global def to nc (2) at stmt 3 on line 6
9(3,2)  global ref to nc (2) at stmt 3 on line 6
18(3,6,8,6,19)   source for node 3 line 6 cols 8-19
8(4,1)  local def to inword (1) at stmt 4 on line 8
18(4,8,11,8,21)   source for node 4 line 8 cols 11-21
8(5,1)  local def to inword (1) at stmt 5 on line 10
18(5,10,10,10,20)   source for node 5 line 10 cols 10-20
```

10(6,1) global def to nw (1) at stmt 6 on line 11
9(6,1) global ref to nw (1) at stmt 6 on line 11
18(6,11,10,11,21) source for node 6 line 11 cols 10-21
16(5,6) connect from 5 to 6
18(7,9,32,9,32) source for node 7 line 9 cols 32-32
18(8,12,7,12,7) source for node 8 line 12 cols 7-7
16(7,5) connect from 7 to 5
16(6,8) connect from 6 to 8
17(7,5,6) nodes 5-6 require node 7
17(8,7) nodes 7-7 require node 8
7(9,1) local ref to inword (1) at stmt 9 on line 9
17(9,10) nodes 10-10 require node 9
17(10,8) nodes 8-8 require node 10
17(10,6) nodes 6-6 require node 10
16(8,10) connect from 8 to 10
16(9,7) connect from 9 to 7
16(9,10) connect from 9 to 10
18(9,9,13,9,30) source for node 9 line 9 cols 13-30
18(10,9,30,9,30) source for node 10 line 9 cols 30-30
7(11,2) local ref to c (2) at stmt 11 on line 7
7(11,2) local ref to c (2) at stmt 11 on line 7
7(11,2) local ref to c (2) at stmt 11 on line 7
17(11,13) nodes 13-13 require node 11
17(13,4) nodes 4-4 require node 13
16(4,13) connect from 4 to 13
16(11,4) connect from 11 to 4
16(12,9) connect from 12 to 9
16(10,13) connect from 10 to 13
16(11,12) connect from 11 to 12
18(12,9,8,9,11) source for node 12 line 9 cols 8-11
17(13,12) nodes 12-12 require node 13
17(13,9,10) nodes 9-10 require node 13
17(12,9,10) nodes 9-10 require node 12
18(11,7,8,7,44) source for node 11 line 7 cols 8-44
18(13,7,44,7,44) source for node 13 line 7 cols 44-44
16(3,11) connect from 3 to 11
8(14,2) local def to c (2) at stmt 14 on line 13
11(14,2) call to getchar at 14
13 end call to getchar at 14
18(14,13,5,13,19) source for node 14 line 13 cols 5-19
16(13,14) connect from 13 to 14
18(15,5,23,5,23) source for node 15 line 5 cols 23-23
18(16,14,5,14,5) source for node 16 line 14 cols 5-5
16(15,3) connect from 15 to 3
16(14,16) connect from 14 to 16

17(15,3) nodes 3-3 require node 15
17(15,11,14) nodes 11-14 require node 15
17(16,15) nodes 15-15 require node 16
7(17,2) local ref to c (2) at stmt 17 on line 5
18(17,5,4,5,21) source for node 17 line 5 cols 4-21
18(18,5,21,5,21) source for node 18 line 5 cols 21-21
16(17,15) connect from 17 to 15
16(17,18) connect from 17 to 18
16(16,17) connect from 16 to 17
17(17,18) nodes 18-18 require node 17
17(18,16) nodes 16-16 require node 18
17(18,11,14) nodes 11-14 require node 18
16(2,17) connect from 2 to 17
4(2,c) local id
18(19,2,28,2,28) source for node 19 line 2 cols 28-28
18(20,15,1,15,1) source for node 20 line 15 cols 1-1
16(19,2) connect from 19 to 2
16(18,20) connect from 18 to 20
17(19,2) nodes 2-2 require node 19
17(19,17,18) nodes 17-18 require node 19
17(20,19) nodes 19-19 require node 20
16(1,19) connect from 1 to 19
17(19,1) nodes 1-1 require node 19
3(1,inword) Formal parm
2(20) end function wordCount
18(21,16,1,16,4) source for node 21 line 16 cols 1-4
1(21,3,main) Function main 3 entry at 21
8(22,1) local def to inw (1) at stmt 22 on line 18
10(22,2) global def to nc (2) at stmt 22 on line 18
10(22,1) global def to nw (1) at stmt 22 on line 18
18(22,18,5,18,22) source for node 22 line 18 cols 5-22
11(23,1) call to wordCount at 23
7(23,1) local ref to inw (1) at stmt 23 on line 19
13 end call to wordCount at 23
18(23,19,5,19,19) source for node 23 line 19 cols 5-19
16(22,23) connect from 22 to 23
11(24,4) call to printf at 24
13 end call to printf at 24
18(24,20,5,20,19) source for node 24 line 20 cols 5-19
16(23,24) connect from 23 to 24
11(25,4) call to printf at 25
12 Actual seperator
9(25,2) global ref to nc (2) at stmt 25 on line 21
12 Actual seperator
9(25,1) global ref to nw (1) at stmt 25 on line 21

```
13   end call to printf at 25
18(25,21,5,21,32)   source for node 25 line 21 cols 5-32
16(24,25)  connect from 24 to 25
4(1,inw)   local id
18(26,16,8,16,8)   source for node 26 line 16 cols 8-8
18(27,22,1,22,1)   source for node 27 line 22 cols 1-1
16(26,22)  connect from 26 to 22
16(25,27)  connect from 25 to 27
17(26,22,25)    nodes 22-25 require node 26
17(27,26)    nodes 26-26 require node 27
16(21,26)  connect from 21 to 26
17(26,21)    nodes 21-21 require node 26
2(27)   end function main
5(2,nc)   Global id
5(1,nw)   Global id
```

A.3 StDG representation (Sites) of *wc* program generated by the ReArchitect

```
Site 2:B S:2,|
 Port(2,0,0,)TaU U{UPP1:PhI 15;DPP1-TaU 2;}
 Port(2,2,0,)TaU D{UPP1:TaU 2;DPP1-TaU 4;DPP2-TaU 5;DPP3-TaU 15;}
Site 4:F,S:4,|
 Port(4,2,0,)TaU U{UPP1:TaU 2;DPP1-PhI 4;DPP2-c 4;}
 Port(4,4,0,)PhI D{UPP1:TaU 4;}
 Port(4,4,0,)c D{UPP1:TaU 4;DPP1-c 5;DPP2-c 7;}
Site 5:B L S:5,|
 Port(5,2,0,)TaU U{UPP1:TaU 2;UPP2:PhI 14;DPP1-TaU 5;}
 Port(5,5,0,)c U{UPP1:c 4;UPP2:c 13;DPP1-TaU 5;}
 Port(5,5,0,)TaU D{UPP1:TaU 5;UPP2:c 5;DPP1-TaU 6;DPP2-TaU 7;DPP3-TaU
13;DPP4-TaU 14;}
Site 6:S:6,|
 Port(6,5,0,)TaU U{UPP1:TaU 5;DPP1-nc 6;}
 Port(6,6,0,)nc U{UPP1:nc 23;UPP2:nc 6;DPP1-nc 6;}
 Port(6,6,0,)nc D{UPP1:nc 6;UPP2:TaU 6;DPP1-nc 24;DPP2-nc 6;}
Site 7:B S:7,|
 Port(7,5,0,)TaU U{UPP1:TaU 5;DPP1-TaU 7;}
 Port(7,7,0,)c U{UPP1:c 4;UPP2:c 13;DPP1-TaU 7;}
 Port(7,7,0,)TaU D{UPP1:TaU 7;UPP2:c 7;DPP1-TaU 8;DPP2-TaU 9;}
Site 8:S:8,|
 Port(8,7,0,)TaU U{UPP1:TaU 7;DPP1-inword 8;}
 Port(8,8,0,)inword  F D{UPP1:TaU 8;}
Site 9:B S:9,|
 Port(9,7,0,)TaU U{UPP1:TaU 7;UPP2:PhI 12;DPP1-TaU 9;}
 Port(9,9,0,)inword  F U{UPP1:inword 23;DPP1-TaU 9;}
```

```
  Port(9,9,0,)TaU D{UPP1:TaU 9;UPP2:inword 9;DPP1-TaU 10;DPP2-TaU
11;DPP3-TaU 12;}
Site 10:S:10,|
  Port(10,9,0,)TaU U{UPP1:TaU 9;DPP1-inword 10;}
  Port(10,10,0,)inword  F D{UPP1:TaU 10;}
Site 11:S:11,|
  Port(11,9,0,)TaU U{UPP1:TaU 9;DPP1-nw 11;}
  Port(11,11,0,)nw U{UPP1:nw 23;DPP1-nw 11;}
  Port(11,11,0,)nw D{UPP1:nw 11;UPP2:TaU 11;DPP1-nw 24;}
Site 12:E S:12,|
  Port(12,9,0,)TaU U{UPP1:TaU 9;DPP1-PhI 12;}
  Port(12,12,0,)PhI D{UPP1:TaU 12;DPP1-TaU 9;}
Site 13:F,S:13,|
  Port(13,5,0,)TaU U{UPP1:TaU 5;DPP1-PhI 13;DPP2-c 13;}
  Port(13,13,0,)PhI D{UPP1:TaU 13;}
  Port(13,13,0,)c D{UPP1:TaU 13;DPP1-c 5;DPP2-c 7;}
Site 14:E S:14,|
  Port(14,5,0,)TaU U{UPP1:TaU 5;DPP1-PhI 14;}
  Port(14,14,0,)PhI D{UPP1:TaU 14;DPP1-TaU 5;}
Site 15:E S:15,|
  Port(15,2,0,)TaU U{UPP1:TaU 2;DPP1-PhI 15;}
  Port(15,15,0,)PhI D{UPP1:TaU 15;DPP1-TaU 2;}
Site 23:B S:-1,|
  Port(23,0,0,)inword  F D{DPP1-inword 9;}
  Port(23,0,0,)nc D{DPP1-nc 6;}
  Port(23,0,0,)nw D{DPP1-nw 11;}
Site 24:E S:-1,|
  Port(24,0,0,)nc U{UPP1:nc 6;}
  Port(24,0,0,)nw U{UPP1:nw 11;}
Site 16:B S:16,|
  Port(16,0,0,)TaU U{UPP1:PhI 22;DPP1-TaU 16;}
  Port(16,16,0,)TaU D{UPP1:TaU 16;DPP1-TaU 18;DPP2-TaU 19;DPP3-TaU
20;DPP4-TaU 21;DPP5-TaU 22;}
Site 18:S:18,|
  Port(18,16,0,)TaU U{UPP1:TaU 16;DPP1-nw 18;}
  Port(18,18,0,)nw D{UPP1:TaU 18;DPP1-nw 21;DPP2-nw 26;}
Site 19:F,S:19,|
  Port(19,16,0,)TaU U{UPP1:TaU 16;DPP1-PhI 19;}
  Port(19,19,0,)inw U{DPP1-PhI 19;}
  Port(19,19,0,)PhI D{UPP1:TaU 19;UPP2:inw 19;}
Site 20:F,S:20,|
  Port(20,16,0,)TaU U{UPP1:TaU 16;DPP1-PhI 20;}
  Port(20,20,0,)PhI D{UPP1:TaU 20;}
Site 21:F,S:21,|
  Port(21,16,0,)TaU U{UPP1:TaU 16;DPP1-PhI 21;}
```

```
 Port(21,21,0,)nc U{UPP1:nc 25;DPP1-PhI 21;}
 Port(21,21,0,)nw U{UPP1:nw 18;DPP1-PhI 21;}
 Port(21,21,0,)PhI D{UPP1:TaU 21;UPP2:nc 21;UPP3:nw 21;}
Site 22:E S:22,|
 Port(22,16,0,)TaU U{UPP1:TaU 16;DPP1-PhI 22;}
 Port(22,22,0,)PhI D{UPP1:TaU 22;DPP1-TaU 16;}
Site 25:B S:-2,|
 Port(25,0,0,)nc D{DPP1-nc 21;DPP2-nc 26;}
Site 26:E S:-2,|
 Port(26,0,0,)nw U{UPP1:nw 18;}
 Port(26,0,0,)nc U{UPP1:nc 25;}
```

Vita

Ramachenga R. Valasareddi received Bachelor of Technology degree in Civil Engineering from Sri Venkateswara University, India. He received Master of Technology in Computer Science from Hyderabad Central University, India. He worked for Kollpuri Amma Foundries in Bangalore, India before attending the University of Southwestern Louisiana, where he received Master of Science degree in Computer Science. He worked for Fontenote Insurance Agency, Lafayette, Louisiana, and River City Medical, Inc., Baton Rouge, Louisiana, as a software engineer.

Valasareddi pursued his doctoral research under the supervision of Prof. Doris Carver. He received Doctor of Philosophy degree in May 1999 form the Louisiana State University. Valasareddi is currently with Chiptrek, Inc., New York. His research interests include software maintenance, restructuring, reengineering, object-oriented programming, and program models. He is a member of IEEE and ACM.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Ramachenga Reddy Valasareddi

Major Field: Computer Science

Title of Dissertation: A Transformational Reengineering System that Supports Software Maintenance using a Graph Representation for the Identification of an Object-Oriented Software Architecture

Approved:

Major Professor and Chairman

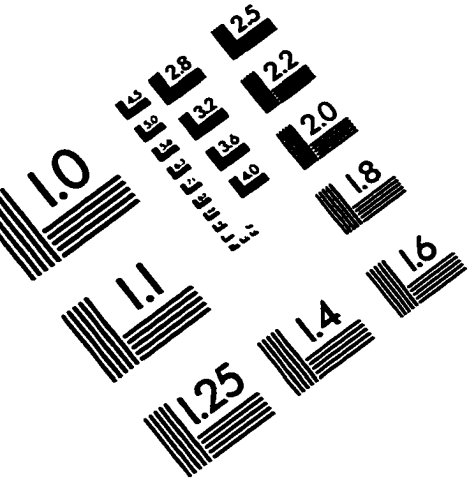
Dean of the Graduate School

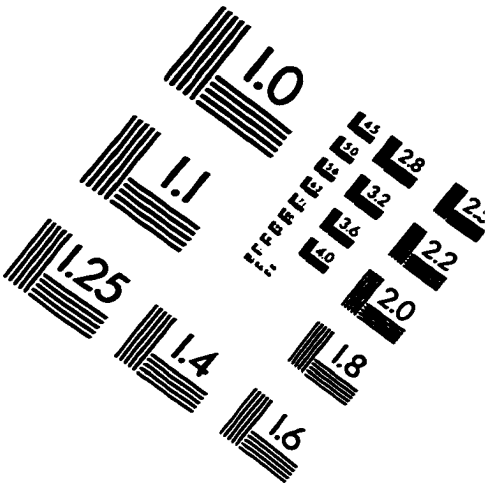
EXAMINING COMMITTEE:

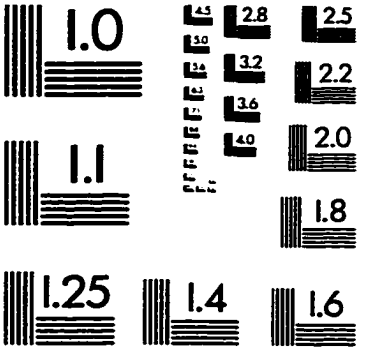
Date of Examination:

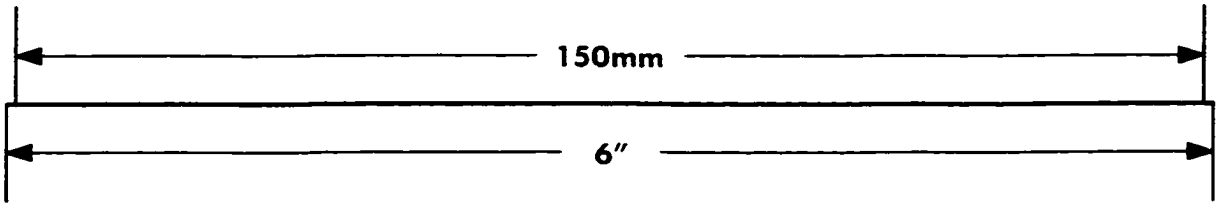
April 7, 1999

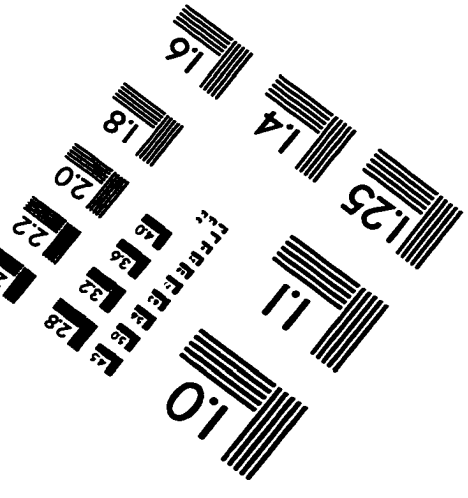
IMAGE EVALUATION TEST TARGET (QA-3)

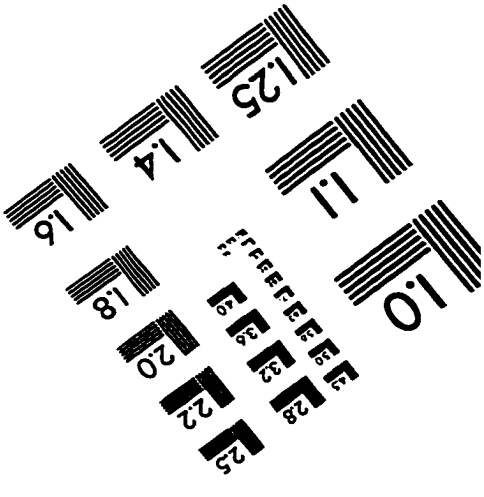












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989