2006

A Sparse Program Dependence Graph For Object Oriented Programming Languages

Keith Garfield
University of Central Florida

A SPARSE PROGRAM DEPENDENCE GRAPH FOR OBJECT ORIENTED PROGRAMMING LANGUAGES

by

KEITH GARFIELD
B.S. Embry-Riddle Aeronautical University, 1983
M.S. University of Central Florida, 2005

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2006

Major Professor: Charles E. Hughes

ABSTRACT

The Program Dependence Graph (PDG) has achieved widespread acceptance as a useful tool for software engineering, program analysis, and automated compiler optimizations. This thesis presents the Sparse Object Oriented Program Dependence Graph (SOOPDG), a formalism that contains elements of traditional PDGs adapted to compactly represent programs written in object-oriented languages such as Java. This formalism is called sparse because, in contrast to other OO and Java-specific adaptations of PDGs, it introduces few node types and no new edge types beyond those used in traditional dependence-based representations. This results in correct program representations using smaller graph structures and simpler semantics when compared to other OO formalisms.

We introduce the Single Flow to Use (SFU) property which requires that exactly one definition of each variable be available for each use. We demonstrate that the SOOPDG, with its support for the SFU property coupled with a higher order rewriting semantics, is sufficient to represent static Java-like programs and dynamic program behavior. We present algorithms for creating SOOPDG representations from program text, and describe graph rewriting semantics. We also present algorithms for common static analysis techniques such as program slicing, inheritance analysis, and call chain analysis.

We contrast the SOOPDG with two previously published OO graph structures, the Java System Dependence Graph and the Java Software Dependence Graph. The SOOPDG results in comparatively smaller static representations

of programs, cleaner graph semantics, and potentially more accurate program analysis.

Finally, we introduce the Simulation Dependence Graph (SDG). The SDG is a related representation that is developed specifically to represent simulation systems, but is extensible to more general component-based software design paradigms. The SDG allows formal reasoning about issues such as component composition, a property critical to the creation and analysis of complex simulation systems and component-based design systems.

To my patient, patient wife, Linda,

and my son, Nicholas.

I love you both.

ACKOWLEDGMENTS

There are many, many people who contribute to these things and helped me to complete this thesis. First, I would like to thank Tiffani Williams and Marc Smith for providing so many stimulating discussions and ideas as we each began our research paths. Tiffani was especially helpful by being the first in our group to provide an existence proof that the degree is attainable, and Marc has been an endless source of encouragement and determination.

I would like to thank my co-workers at the Institute for Simulation and Training (IST), who have always been supportive of the attempt to complete the degree. The cross-fertilization of ideas between my IST research topics and thesis topic benefitted both worlds.

Thank you Mom and Dad for giving me the gift of curiosity, and my sisters Colleen, Denise, Terry, and Wanda for your encouragement and support.

I would like to thank my committee members. Dr. Dutton provided a kind of symmetry by admitting me in the graduate program and participating in the defense that got me out. Dr. DeMara and Dr. Wu have been guides acedemically and professionally. I cannot say enough about the patience and effort my long-suffering advisors have consistently shown. Dr. Hughes juggled an astounding workload while Dr. Parsons juggled an astounding intercontinental travel schedule to provide their guidance. Thank you for giving the benefit of your broad experience, unique expertise, and high academic standards. Also thank you for pushing when it was productive to do so and for not pushing when it was more beneficial to not push.

Lastly, thank you to my wife Linda and son Nicholas. Nicholas has shown more patience at my preoccupation than young people often do, and has matured into a fine young man over the last few years. Linda has been a “thesis widow” for far too long as so much of my time and energies were devoted to this. Thank you for your patience. Thank you for your belief. Thank you for your presence by my side.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

This thesis presents the Sparse Object Oriented Program Dependence Graph (SOOPDG), a new program representation capable of supporting a Java-like language. In this chapter we outline the motivation for development of this representation by presenting a summary sketch of the limitations of existing representations and the advantages of the SOOPDG. We also present an overview of the target language used for discussing and developing the SOOPDG. These topics are fully developed in later chapters.

The Program Dependence Graph (PDG) is a dependence-based intermediate program representation used widely in software engineering and analysis applications [FOW87, LMP99, RWF03]. Software engineering activities supported by the PDG have grown since its first recognition as a software development tool in 1984 [OO84] to include program slicing, differencing, integration, debugging, testing, maintenance, complexity analysis, and semantic evaluation [HR92, Hor90, Par92, Zha98]. The PDG was originally introduced for single-threaded imperative programs, although the original forms inherently supported parallel computation [CF89, Par92]. Horwitz et al [HRB90] extended the PDG to represent multi-procedural programs, and thus explicitly multi-threaded programs, with the System Dependence Graph (SDG). The introduction of the PDG predates widespread acceptance of Object Oriented (OO) programming languages. With the growth of popularity of Object Oriented (OO) languages, variations of the PDG and SDG for OO programs have been presented in the lit-

erature [MMK94, CY96, LH98, CX01]. Zhao [Zha98] presented and Walkinshaw et al [WRW03] extended representations specifically supporting Java. These representations allow for static program representation only and have inefficiencies described more fully in Chapter 6. There is a need for a representation modeling static programs more efficiently, and capable of representing dynamic program behavior.

1.1 Need for the Work

Dependence based program representations supporting Java exist in the form of the Java System Dependence Graph [Zha98] and the Java Software Dependence Graph [WRW03]. These representations extend the basic PDG structure through the introduction of multiple special purpose node and edge types to support OO features. These special purpose graph elements complicate graph structure and increase the graph size. The bloat in the program representation can result in inefficiencies in program analysis and in development of executable code (or bytecode). Dynamic binding of polymorphic methods is supported in an inefficient manner through duplication of portions of the graph for each possible binding that may occur.

In addition, the authors do not present an underlying rewriting semantics for their representations, which limits program analysis to static cases. Analysis has been performed for non-OO languages and representations proving that translations from textual to graphical representations, and potential transformations performed on the graphical representation, preserve program semantics [CF89, Sel90b]. The limitation to static analysis prevents a similar formal analysis to be performed for Java programs.

1.2 The Sparse Object Oriented Program Dependence Graph

The representation presented within this thesis has been named the Sparse Object Oriented Program Dependence Graph because it introduces a small number of new node and requires no new edge types beyond those used in traditional PDGs, and results in smaller static program representations than achieved in compatible models. The reliance on traditional PDG elements maintains the attractive qualities of the original PDG forms. Primarily, the introduction of no new dependence edges beyond those used in traditional PDGs maintains the core of program semantics intended by traditional PDGs [KKP81, FOW87, Hor90], while allowing OO features to be represented. Additional qualities include the notion of a local store, compositional semantics, and inherent support for parallel computation. The SOOPDG utilizes higher order rewriting semantics. This will be shown to result in improved support for dynamic binding through straightforward resolution of method execution at specific calling sites, and smaller graphs than existing forms. These improvements in the representation result in improvements in the efficiency and accuracy of program analysis.

1.3 Limitations Imposed in the Language

The target language, J, does not have all features of Java. Specifically, we consider only thread safe applications, and do not allow unstructured control flow. We do not allow shadowing. We do not provide specific support for arrays or pointers, though future research may accomplish this by leveraging previous work [Par92] performed on traditional PDGs. We do not explicitly represent input-output forms, but make provisions for data values to be established at run time, and

program results to be returned to an external operating environment upon termination of the computation. These restrictions are typical of other formalisms presented in the literature [KKP81, TGH92, Zha99, WRW03, AH03]. The language J is more fully defined in Chapter 2.

1.4 Contributions of this Thesis

This thesis contributes to the fields of Computer Science, and Modeling and Simulation in the following ways.

1. Definition of the SOOPDG, a dependence based program representation targeting a Java-like language (and more generally, OO constructs).

2. Presentation of the first program representation supporting Java-like languages (the SOOPDG) that employs only the dependences required to maintain program correctness (flow, control, and def-order, defined in Chapter 2).

3. Definition of a rewriting semantics that allows the SOOPDG to represent computations in Java-like languages, and also permits analysis of dynamic program events as opposed to static analysis on the program structure. We could find no other rewriting semantics for dependence graphs representing as full a range of OO features as the SOOPDG supports.

4. Definition of the Single Flow to Use (SFU) property of programs, which is a new property used to disambiguate among multiple sources of values for variables during program execution and which makes analyses more precise.

5. Initial development and presentation of a related representation supporting analysis of modeling and simulation systems, the Simulation Dependence Graph.

1.5 Organization of this Thesis

The remainder of this document is structured as follows. Chapter 2 presents pertinent background information and definitions regarding the language J, program properties, and dependence based program representations. Chapter 3 introduces the formal definition of the Sparse OO PDG. Algorithms for constructing SOOPDG graphs from program text and an associated graph rewriting scheme are presented in Chapter 4. Program analysis algorithms are presented in Chapter 5, while comparisons with selected dependence based graph representations that support Java are given in Chapter 6. Chapter 7 presents the Simulation Dependence Graph, a related representation applicable to simulation systems. Future research opportunities and applications of the SOOPDG extending beyond the present discussion are given in Chapter 8. Conclusions are given in Chapter 9.

2

BACKGROUND

This section presents background information related to the topic of this thesis. We discuss an underlying semantics for computation, the target Java-like language being supported, program dependences and pertinent program properties. We present the program dependence graph (PDG), which forms the basis of the SOOPDG. Finally, we define specific structures found in the program dependence graph that will be used in future chapters when proving properties of the SOOPDG.

2.1 Informal Semantics of Computation

We consider program behavior through observable input-output behavior. A program performs transformations from an input to an output memory state, called a *program store* (We extend this notion slightly in the context of the target language, J, in Section 2.2). The program store is the mathematical model of physical memory, and consists of a mapping from program variables to values [CF89, Par92]. Programs perform computations by receiving input data from an input store, operating on the data, and producing an output store. In this sense programs define a mapping from input to output stores. This mapping is well defined for programs that terminate normally; the symbol of "bottom", $\perp$, is reserved to represent the output of programs that do not terminate, or

do not terminate normally [CF89, Par92]. Bottom is formally defined below in Definition 1. A program's mapping from input to output store provides an objective and non-ambiguous specification of program meaning [Mog91], allowing comparisons of program behaviors. Specifically, two programs are said to have the same meaning if they have the same input-output mapping for all input stores leading to normal termination for both programs. No correlation is made if either program terminates abnormally or if either program does not terminate.

Definition 1 (Bottom) *Bottom is a special value included in the range of values allowed in a computation that represents an undefined value. The result of any undefined or non-terminating computation is bottom. The result of any computation containing bottom is bottom. In this sense, bottom is a "sticky" value.*

Just as programs may be represented in various ways, various methods may be used to evaluate them. Evaluation methods differ in whether execution sequences emphasize program structure or results, whether a first order or higher order semantics is in effect, and how program termination is defined. *Sequential* (also called *imperative*) execution follows program structure as written by a programmer in a top down fashion. *Lazy* evaluation executes only program statements contributing to program results. These concepts are formally defined in Definitions 2 and 3, respectively. Lazy evaluation is also known as *call by need* or *demand evaluation*, as the evaluation begins at the statement defining program output and a need, or demand, for values is propagated backwards through the program representation. The execution of statements required to satisfy the demand then propagates forward through the program representation. The backward propagation of demand and forward execution of statements are not necessarily separated into distinct phases. The demand may designate multiple statements that potentially contribute to a computation at a given program

point; none of these statements execute until it is determined which ones will actually contribute.

Definition 2 (Sequential Evaluation) *Evaluation that begins at a uniquely designated start point and executes program statements in the order they appear in the program format, executing all statements on the evaluation path regardless of the relevance to the end result. Evaluation of a statement cannot begin until the previous statement has successfully terminated [Sel89, Rey98].*

Definition 3 (Lazy Evaluation) *Evaluation that executes only those program statements known to contribute to the program result. Execution sequence is not determined by the order in which statements appear in the program format, other than that required for semantic correctness. [Kri89, Sel89].*

Program execution semantics may be segregated into *first order* or *higher order* semantics. First order semantics allow only primitive values to be assigned to variables or flow between program statements, while higher order semantics allow functions to be assigned to variables and flow as values between program statements.

Program evaluation may also be classified according to the effect of undefined program elements on termination. Strict evaluation requires that elements of the program be well defined for successful termination of the computation. Strictness criteria, defined in Definition 4, typically is applied only to those statements that execute (consider the branches of an if-then-else construct for which only one branch will execute) [CF89, Par92]. This makes the strictness criteria synonymous with sequential evaluation, because in both models every statement along the control path from the start to finish must be well defined regardless of their contribution to the result. A statement, or its computational result, that is not

well defined is given the value of $\perp$, and the program result is $\perp$. A strict memory model requires that all program variables be represented and mapped to a valid value, regardless of which variables are accessed, for the store to be valid.

Definition 4 (Strictness Property) *The strictness property requires that all elements of a program be well defined for the result to be valid. Strictness may be applied to program evaluation, to the memory model, or both.*

One model of program evaluation transforms the program in a piecewise fashion as intermediate results are achieved. This process is known as *rewriting*, and is defined in Definition 5. As specified below, rewriting is independent of representation, although the term is typically used in the literature in association with graph representations. The notion is that the program is evaluated via successive step-wise rewritings until either an undefined state is reached ($\perp$), or the computation terminates normally. Each rewriting is the result of the application of a single rule. If the language adheres to a deterministic semantics, then the result of any single-step rewriting will be deterministic. Likewise, the overall result of rewriting performed on the program from initial to final states will be deterministic. If sequential evaluation rules are in effect then the choice of which rewriting rule will be applied next can be determined at any point in the program evaluation. For non-sequential evaluations such as lazy evaluation, the actual rewriting step performed at any given program point may be non-deterministically chosen from a set of options allowed by the semantics, though the overall program result remains deterministic. Non-sequential evaluation allows for multiple correct execution sequences of a single program, which in turn implies that one sequence, or a parallel sequence, may be chosen over another to enhance program performance without affecting program results.

Definition 5 (Rewriting) *Rewriting is the process of performing program evaluation via step-wise rules that systematically modify the program representation.*

2.2 Representative OO Language, J

To provide clarity of discussion and analysis, we restrict ourselves to a subset of the full Java language that incorporates the primary OO features of class definitions, interface definitions, inheritance, and packages. We refer to this language as J, to distinguish it from a complete implementation of the Java language. We restrict J to single-threaded applications having structured control flow, and having no mechanism to handle exceptions such as *try-catch-throw*. J is comprised of a set of primitive variable types (e.g. int, double, etc.), operations on these primitives (e.g. +, -, *, etc.), and a set of base classes approximating the Java language specification. Program authors define additional classes that extend these base classes. A program in J is composed of a set of class definitions containing class variables, instance variables, and both class and instance methods defining functions or operations on the variables. Programs may contain *interfaces*, or abstract classes, that must be *implemented* prior to instantiation into a program as an object. Each program must contain at least one class having a single class method called *main* so that initiation of program execution may be specified. Statements may initiate execution of methods through *method calls*. The term *call site* refers to the specific point in a statement calling a method. Variables and methods may be public (accessible by elements outside the class), private (accessible only by other elements of the same class), or protected (accessible by subclasses and members of the same package).

Classes are comprised of a collection of variables and methods (both class and instance). To maintain structured control flow, methods may have at most one *return* statement. Methods may contain any number of side-effects. Classes are instantiated to create objects in a program using a *constructor* method. The language specification provides each class with a default no-argument constructor, though program authors may define additional constructors. Classes enter into a *superclass/subclass* relationship when one class (the subclass) declares itself as *extending* another (the superclass). Instance variables and methods not defined in the subclass are *inherited* from the superclass. Shadowing is not allowed in J, so variables are declared exactly once in any superclass/subclass hierarchy of classes. A method call to a constructor instantiating a subclass results in a method call to the constructor of the superclass as well. The call to the superclass constructor is an implicit call to the default class constructor unless an explicit call is made. An interface is a collection of abstract methods, each having method type and input parameters defined. A class *implementing* the interface must define every method in the interface using the same number and type of input and output parameters as in the abstract definition.

Statements in J allow *output*, *return*, *declaration*, *assignment*, *input*, *if-then-else* constructs, and *while* loops. We do not consider *switch* statements, as they may be reproduced through multiple *if-then-else* structures. Similarly, we don't allow *for* statements as they can be affected through *while* loops. We do not allow *continue* or *break* statements, as they introduce unstructured control flow. We further simplify J by abstracting the details of input and output operations away, allowing programs in J to accept input at run time and produce output without specifying detailed syntax. Specifically, we will use "x = <input>;" to represent receipt of a value for variable "x" at run time, "<output> = x;" to represent program output. The introduction of the <*input*> and <*output*> terminology

slightly modifies the observable program behavior as discussed in Section 2.1; we now consider program meaning as a transformation from an input *stream* to an output *stream*. An example of a program written in J is provided in Figure 2.1.

Assignment statements and typed methods explicitly list the program store mapping affected by their execution. The target of the update for an assignment statement is obvious from the form of the statement. The target of a store update accomplished within a method is discerned from the method's type and return statement. The affected store is local to the method, and the store value passed via the *return* statement to the calling context.

A *side effect* is an update to the program store not explicitly represented in the assignment statement or method type. Within J, a side effect is an assignment within a method to a variable not specified as a formal output parameter of the method. For example, the program statement "y = x * i++;" contains both an explicit update to variable "y" and an implicit update to variable "i". Increment and decrement operators within a statement act as a side effect performing an update to the store in addition to the targeted action of the statement. Other examples include constructor methods, which have no formal output useful in a computational model, yet are used to provide initial values for instance variables. Similarly, the ability for a class's variable values to be modified by external entities is commonly provided through definition of a mutator method. A typical mutator method is of type *void*, and assigns values to one or more variables, technically through a side-effect. These examples are generally considered acceptable in typical programming practices. A more pernicious example, the *sideEffectExample* method, is given below in which an instance variable, $o.x$ is modified through a side effect. It is difficult to analyze the effect of a call to this

method, particularly as the actual object being affected may not be known prior to run-time.

```
public int sideEffectExample(Object o) {
    int result = o.x + o.y;
    o.x = F();
    return result;
}
```

```
1. public class Program1 {
2.   public static void main( String args[]) {
3.     int y;
4.     Obj1 a;
5.     Obj1 b;
6.     Obj2 c;
7.
8.     a = new Obj1();
9.     b = a;
10.    c = new Obj2();
11.
12.    a.setp(20);
13.    y = c.getp() + b.getp();
14.    <output> = y;
15.  }
16. }

17. class Obj1 {
18. int p;
19.
20. public Obj1() {
21.   p = 10;
22. }
23.
24. public void setp(int x) {
25.   p = x; }
26.
27. public int getp() {
28.   return p; }
29. }

30. class Obj2 extends Obj1 {
31.
32.   public Obj2() {
33.     p = 20;
34.   }
35.
36.   public void setp(int x) {
37.     p= 2*x;  }
38.   }
```

Figure 2.1: Language J Example: Program 1

Evaluation of programs in J follow a sequential programming semantics operating on a central program store. This implies that program variables (memory locations) may have different values assigned to them during a single program execution. We define DEF and USE statements within programs written in J as the program points where variable values are defined and used, respectively. Determination of which value is in effect at any given time is done through variable *liveness* analysis that tracks which value assignments have the potential to reach specific program points [ALS07]. This liveness analysis establishes a series of DEF-USE relationships between program statements that give rise to

dependences between them based on data flow. Similarly, the results of predicate evaluations in a program specify which other program statements may or may not execute. This gives rise to dependence relations established between the predicate and affected statements. These dependences are defined more fully in the following section.

Throughout this thesis the following notation is used when referring to elements of J programs.

- Π represents an entire J program.
- π represents a segment of a J program. Typically, π_c represents class bodies, π_m represents method bodies, π_l represents loop bodies, and π_T and π_F represent the True and False branches of an if-then-else structure.
- c represents class names, while o represents instantiated classes (objects).
- s represents a single statement in a J program.
- x represents a single class or instance variable.
- t represents variable types, while val represents a generic value.
- "dec-list" represents a list of parameters found in a declaration statement. For example "dec-list x;" is the general form for declaration of variable x.
- $c.x$ and $o.x$ represent class and object variable designations, respectively. For convenience we allow c to contain the full class definition necessary to fully define the scope of variable x. For example, given variable x in class "Class.subclass1.subclass2", the complete notation of "Class.subclass1.subclass2.x" is represented as $c.x$, with $c =$ "Class.subclass1.subclass2", and $x = x$. Similarly, "o.x" may actually represent "Object1.Object2.Object3.x."

- X_f and X_a represent tuples of formal and actual parameters, respectively. X_f is of the form "$t_1\ x_1, t_2\ x_2, \ldots, t_k\ x_k$", and X_a is of the form "$x_1, x_2, \ldots, x_k$."

- m represents method names. Method definitions are presented as "dec-list $m(X_f)$ { π_m }" and method calls presented as "$o.m(X_a)$".

- $F(X_a)$ represents an expression (function) within a program statement. $X_a = (x_1, x_2, \ldots, x_k)$ is the k-tuple of variables occurring in F. We reserve $P(X_a)$ for expressions found in *predicate* statements.

- C and C' represent program control points of the form (pid, b). The pid is a predicate node identifier, and b represents a Boolean value.

Using this notation, we represent a typical program as a collection of classes, Π = dec-list c_1 { π_1 } dec-list c_2 { π_2 } dec-list c_3 { π_3 } ... dec-list c_k { π_k }. Class bodies contain variable declarations, "dec-list x;", and method definitions, "dec-list $m(X_f)$ { π_m }." *Predicate* structures take the form "if $P(X_a)$ { π_T } else { π_F }" and *while* loops take the form "while $P(X_a)$ { π_l }."

2.3 Object Aliasing

Object aliasing occurs when a variable of type "object" is assigned a value. After the assignment, a reference to either object's variable refers to the same memory space. Effectively, a reference to one object is indistinguishable from a reference to the other. The alias may occur in an assignment statement, "$o_1 = o_2$." Aliasing may also occur as a result of an object's handle being passed into a method as a parameter. For example, if method m is defined as follows: "dec-list m(c o_a) { π_m }", and a call to m is made as follows: "o_1.y = o_1.m(o_2);", then o_a and o_2

are aliased within the context of the method. Class variables present a special case in that the single store location may be referenced via the class ("c.x") or instantiated object name ("o.x"). In this case, we add the class variable to the alias list using the class name, and add each object name version upon processing an object instantiation assignment statement ("o = new c();").

Alias relationships at any point in a program's execution are typically classified as *cannot alias*, *must alias*, and *may alias* [FYD06]. The *cannot alias* relationship occurs when object handles o_1 and o_2 cannot refer to the same object. The *must alias* relationship occurs when handles o_1 and o_2 definitely refer to the same object. The more problematical *may alias* relationship occurs when the aliasing occurs on one branch of a predicate. In this case, the actual alias relationship cannot be determined until execution.

We present an example using Program 1 from Figure 2.1. During execution of Program 1 objects a and b are aliased as of line 9. The method call *b.getp()* in line 13 returns the value 20, as that was the value set by the method call *a.setp(20)* in line 12. Since a and b refer to the same object at this point, DEF statements using either object names affect all future USEs of the object.

2.4 Program Dependences

Program dependences describe formal relationships between program statements, specifying which other statements influence a given statement's execution [KKP81, BM92]. Some dependences, such as *output* (Definition 6) and *anti-* (Definition 7) dependences, are relics of the details of a specific programming language or coding style [KKP81]. These may be resolved through techniques such as variable renaming in the absence of arrays and pointers [KKP81, TGH92]. The remaining

dependences, called *true* dependences, are inherent to the computation regardless of program and memory models used. The true dependences, are *control*, *flow*, and *def-order* dependences. We use the terminology provided in [CF89, Par92] to formally define these in Definitions 8, 9, 10, respectively. Control dependences identify which statements are executed based on a specific predicate statement result. When a lazy execution semantics is in effect, satisfaction of control dependence alone is a necessary but not sufficient condition for ensuring statement execution. Flow dependence specifies which DEF-USE statement pairs are involved in a WRITE-READ relationship for a given variable. Def-order dependences exist when multiple DEF statements each may supply a value to the same USE statement. This occurs when at least one of the DEF statements is executed conditionally.

Definition 6 (Output Dependence) *B is output dependent on A iff the execution of A occurs before B in a strict execution semantics program sequence, and both A and B assign to the same variable.*

Definition 7 (Anti-Dependence) *Statement B is anti-dependent on statement A iff A precedes B in a sequential execution, and B assigns a value to a variable used as input in A.*

Definition 8 (Control Dependence) *B is control dependent on A iff*

1) A is a program control flow statement containing a predicate expression that will evaluate to Boolean True or False.

2) B executes upon either A's evaluation to True or False, but not both.

3) There are no intervening statements for which (1) and (2) apply to B.

Definition 9 (Flow Dependence) *Statement B is flow dependent on A iff A is a DEF and B a USE statement for the same program variable, and there are no intervening DEF statements for that variable between A and B on some control flow path from A to B.*

Definition 10 (Def-Order Dependence) *B is Def-Order dependent on A iff*

1) Both A and B are DEF statements for the same program variable.

2) A precedes B in a strict execution sequence.

3) There is some statement C that is flow dependent upon both A and B.

It is widely accepted that compilers and interpreters transform programs to achieve some performance improvement. There are some program transformations that are known to revise dependence relations without affecting program meaning. Examples are constant propagation and variable renaming. Other transformations affect the execution sequence of the computation. The computation (input-output mapping) performed by the new execution sequence must provide the same result as the original sequence. Dependences provide a yardstick by which program meaning can be measured. Any execution sequence that respects the dependences defined by the original sequence of statements will yield the same result as the original [KKP81, Par92]. In effect, the dependences impose a partial ordering that must be respected by any correct execution sequence.

2.5 Single Assignment

Data dependence analysis is intrinsic to program analysis, optimization, and parallelization algorithms [HU75, KKP81, PP96]. The ability to determine, for a

given USE, what DEF supplied a value for a specific variable is critical to these analyzes in terms of discussing variable liveness and reaching definitions. Single Assignment (SA) has been used as a basis by several authors in attempts to specify unambiguous sources of values for program variables during program execution [AWZ88, CFR89, SHW93]. The precise definition for SA (Definition 11) allows at most one value assignment to each user defined variable during the course of a program's execution. While the property occurs naturally in pure functional programming languages [Hug89], renaming of program variables is required to obtain SA in imperative languages [CFR89]. In addition, some mechanism must be introduced to allow only one value to flow beyond program points where distinct control paths converge. The traditional method to enforce the SA property is done in two phases. The first phase requires that user defined program variables receive unique names at each program point where they receive an assignment. The second phase resolves data flow at converging control dependence paths. We will follow the technique introduced by Cytron et al [CFR89] in the discussion of incorporating the SA property into program dependence graphs (see Section 2.7). This method introduces a pseudo-function at the merge points of control paths, referred to as the ϕ-function. A statement containing a ϕ-function takes the form $X = \phi(x_1, x_2, \ldots, x_k)$, where the inputs to the function are the renamed forms of a single original program variable, and the output is one of the values. Thus, the statement assigns to variable X only one of the k values flowing into the function. This ensures that only one DEF statement will supply a reaching definition to any USE statements beyond the ϕ-function in the static program representation. Figure 2.2 presents an example of a (partial) program written in J and its SA form.

Definition 11 (Single Assignment Property) *A program exhibits the SA property if, for each variable defined in the program, at most one assignment is made to the variable.*

1. $y =< input >;$	1. $y =< input >;$	1. $y =< input >;$
2. $x =< input >;$	2. $x_1 =< input >;$	2. $x =< input >;$
3. $if\ (P(y))\ \{$	3. $if\ (P(y))\ \{$	3. $if\ (P(y))\ \{$
4. $x = 42;$	4. $x_2 = 42;$	4. $x = 42;$
5. $\}\ else\ \{$	5. $\}\ else\ \{$	5. $\}\ else\ \{$
6.	6.	6. $x = x;$
7. $\}$	7. $\}$	7. $\}$
8.	8. $x_3 = \phi(x_1, x_2);$	8.
9. $z = F(x);$	9. $z = F(x_3);$	9. $z = F(x);$
10. $< output > = z;$	10. $< output > = z;$	10. $< output > = z;$
Program 2	*Program 2*	*Program 2*
Language J	*SA Form*	*SFU Form*

Figure 2.2: Program 2 (partial): Original Form, SA Form, and SFU Form

2.6 Single Flow to Use

As stated previously, data flow analysis requires that DEF-USE relationships be readily determined between program statements. We introduce the Single Flow to Use (SFU) property as a useful characteristic to determine which DEF statement actually supplies a value to a USE statement. The SFU property, defined

in Definition 12, requires that during program execution exactly one value flow to each USE statement for each variable in the USE statement. This dynamic requirement holds regardless of the number of DEF statements potentially able to provide a value based upon static program representation. The SFU property is achieved through the use of strategically placed identity assignments (see statement 6 in the SFU form of Program 2 in Figure 2.2) that ensure that all DEF statements potentially supplying a value to a USE statement for a given variable are control-wise mutually exclusive. Thus, regardless of what control path is taken to reach the USE statement, only one DEF statement may execute for each variable and supply a value to the USE statement. Programs satisfying the SA property do not necessarily obtain the SFU property. Figure 2.2 provides an example of a partial program written in J in its original, SA , and SFU forms.

Definition 12 (Single Flow to Use Property) *A program exhibits the SFU property if, for each USE statement execution, exactly one DEF statement provides a value for each variable required for the USE statement to execute.*

2.7 Program Dependence Graph

The PDG is an acyclic, directed graph, composed of a node set and two edge sets, that explicitly represents the control and data flow dependences within a program [FOW87, Par92]. PDG nodes roughly correspond to program statements while edges represent control and flow dependences. In addition to allowing static program analysis, PDGs provide an adequate representation to perform program execution through graph re-writing [FOW87, Par92].

The PDG node set is composed of a unique *Start* node that indicates where control flow initiates, a unique *end* node that specifies the program result (output), *idef* nodes that allow initial value definitions for program variables at run time, *set* nodes that correspond to the binding of values to variables, *predicate* nodes that correspond to the predicate portion of program control statements, and *while* nodes allowing loop behavior. The PDG edges explicitly represent control and flow dependences. Control dependence edges signify which nodes may execute based on the outcome of a specific predicate, and can be determined from a Control Flow Graph [Rei78, FOW87]. Flow edges represent the potential for a value to flow from a DEF node to a USE node. The PDG structure releases the computation from the requirement for a central store. Each node has access, via incoming flow edges, to the variable-value mapping appropriate for the node's computation. Thus each node effectively contains a local store maintaining incoming values for use in the computation, and retaining any resulting variable-value mapping update (DEF) for use by nodes that are the target of outgoing flow edges.

Output and anti-dependences existing in a program's textual representation are not explicitly represented but are respected by the PDG creation algorithm. The necessary sequencing constraints are embedded using the flow dependence edges as the PDG is built through use of variable liveness analysis [BMO90, Par92]. Def-order dependences may be explicitly resolved in a PDG using def-order edges that restrict execution sequences [HR92]. Alternatively, def-order relationships may be resolved implicitly within the PDG through special constructs such as ϕ-nodes corresponding to ϕ-functions placed in text programs [CFR91]. Cartwright et al [CF89] introduced the use of strategically placed identity assignment nodes in the PDG to resolve Def-Order dependence. As these nodes restrict the flow of values they were named *valve* nodes [CF89, Sel90a]. We

explore ϕ-nodes and valve nodes more fully in the following sections discussing forms of the PDG specifically supporting the SA or SFU properties.

Graph *rewriting* (Definition 5) rules are associated with the PDG, and provide a mechanism allowing computations to be modeled through modifications to the PDG representation [CFR89, CF89, Sel90a]. This allows analysis of the dynamic program behavior in addition to analysis of the static program representation. Rewriting rules vary to allow sequential or lazy evaluation semantics, or to accommodate modifications to the basic PDG structures defined in this section. Rewriting is typically a node-by-node process, with the act of rewriting a node corresponding to the execution of the program statement the node represents. In this thesis we use the terms *rewriting* and *execution* of a node (Definition 13) interchangeably unless the context of the discussion requires one or the other for clarity. Rewriting rules specify control and flow criteria that must be satisfied before an individual node is permitted to execute (i.e. be rewritten). Typically, satisfaction of a node's control dependence criteria requires that the node's incoming control dependence edge is identified as being on the program execution path (Definition 14) . Since this determination is dependent on the resolution of the predicate node at the head of the edge, there is a sequencing constraint embedded in the graph. In a sense, satisfaction of a node's control dependence supplies "permission" for the node to execute. Within a given program execution, a *bypassed* node is one for which control dependence can never be satisfied due to resolution of a predicate at some control ancestor. Similarly, satisfaction of a node's flow dependence criteria requires that the node receive correct variable values along its incoming flow edges sufficient to perform the rewriting operation. We present an informal definition of this notion in Definition 15. Whereas control dependence corresponds to "permission" to execute, satisfying flow dependence criteria corresponds to "capability" to execute. We describe a rewriting seman-

tics for the SOOPDG in Chapter 4 that utilizes the control and flow criteria of the SFU form of the PDG presented in Section 2.9.

Definition 13 (Node Execution) *Given a node, n, in a PDG, node execution corresponds to the act of rewriting the node and allowing the result to be available to all outgoing edges. We refer to the point in rewriting where this occurs as the* instant of the execution *of the node.*

Definition 14 (Control Dependence Criteria) *Given a node, n, with an incoming control dependence edge associated with a branch, b, of a predicate node, p, the control dependence criteria for n is satisfied when the rewriting of predicate p results in value b.*

Definition 15 (Flow Dependence Criteria) *Given a node, n, that is a USE node for variables $x_1, x_2, \ldots, x_k$, then flow dependence is satisfied when a correct value is received for each variable, $x_i, 1 \leq i \leq k$.*

Although not central to the theme of this thesis, the topic of parallel computation deserves a brief mention. The basic PDG inherently supports parallel computation. This is due to the fact that there is no restriction on rewriting sequencing beyond those explicitly represented through dependence edges. The program result is the same regardless of the specific sequence of node execution, as long as the dependences are respected [Sel89]. This allows the PDG to directly support parallel rewriting of nodes or subgraphs. The growth of explicitly parallel languages, and the availability of parallel execution environments gave rise to explicitly parallel variations on the basic PDG. Horwitz et. al. [HRB90] introduced the System Dependence Graph, extending the PDG to represent collections of procedures as opposed to monolithic programs. This was the first PDG variation providing explicit depictions of parallel or multi-threaded programs.

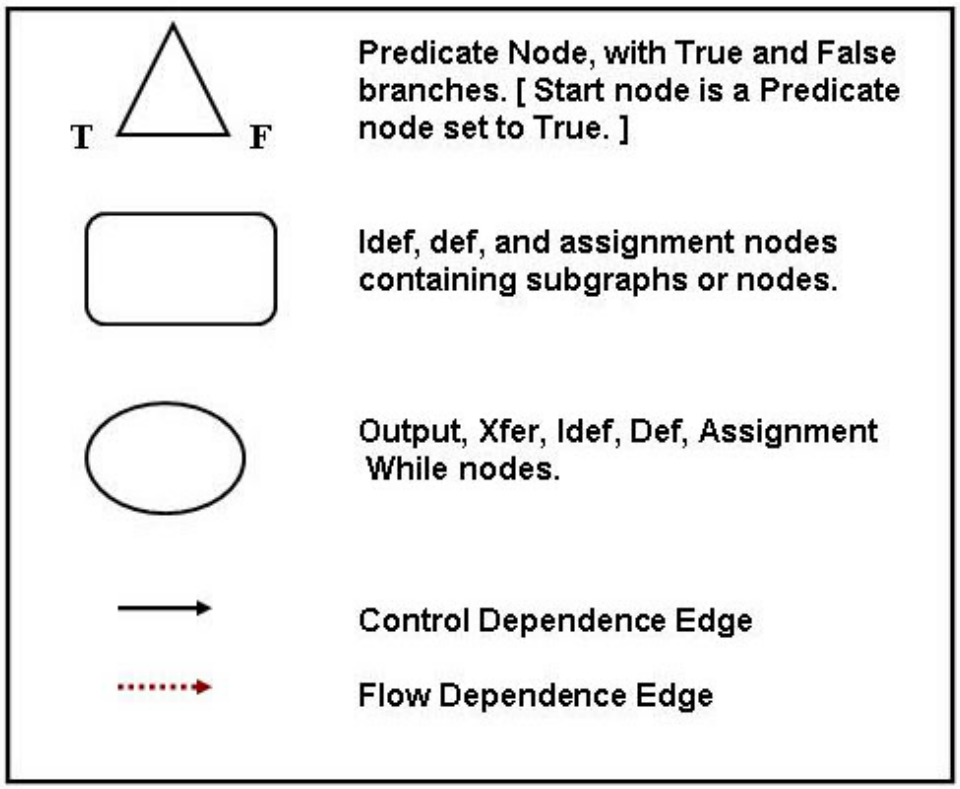


Figure 2.3: Key to Elements Used in Example Graphs

Throughout this paper we will use the formats shown in Figure 2.3 to depict program graphs. Predicate nodes are shown as triangular shaped nodes, with control dependence edges associated with a *True* predicate result emanating from the left-hand corner, and control dependence edges associated with a *False* predicate result emanating from the right-hand corner. All other node types are represented as ovals or elongated rectangles. Control dependence edges are shown in solid arrows, while flow dependence edges are shown as dashed.

2.8 SSA Form

The Static Single Assignment (SSA) form of the PDG imposes the Single Assignment (SA) property on the static program representation [CFR89, LCH04]. The SSA form requires that each variable in the graph have exactly one DEF node, and thus each variable in each USE node depends on exactly one definition (assignment) [BCH98]. To achieve this, variable renaming is performed such that each renamed variable receives exactly one assignment, and the PDG format and semantics are extended to include the ϕ-node [CFR89, RWF03]. Renaming is a straightforward exercise in the absence of array variables and pointers [KKP81]. The ϕ-node resolves multiple assignments to the same (pre-renaming) variable occurring along separate control flow paths using a pseudo-function having the form $X = \phi(x_1, x_2, x_3, \ldots, x_k)$. A ϕ-node has k incoming flow edges, one for variable x_1 through x_k, and one outgoing flow edge for variable X connected to each USE of X. The ϕ-function assigns a value from at most one of the k incoming edges to flow beyond the ϕ-node to uses of X. In summary, the SSA form has the two properties that each *programmer specified use* of a variable is reached by exactly one assignment to that variable, and the program contains ϕ-functions to distinguish the correct value to flow beyond the merge point of distinct control flow paths [CFR89]. An example of an SSA form PDG is presented in Figure 2.4.

The advantages of ϕ-nodes for PDGs are that they provide a clear framework to perform data flow analysis involving DEF-USE chains [CFR89, BCH98], and $O(n)$ algorithms (in terms of time and number of nodes required) for placing ϕ-nodes have been developed [CFR89, SG95, BCH98]. The disadvantages are that the ϕ-node introduces additional complexity to PDG semantics [RWF03], does not have composable semantics [Par92], is not referentially transparent [Hav93], does not provide a mechanism to discriminate among the various definitions

reaching them [BMO90], does not support certain forms of copy propagation [BCH98] and eventually requires replacement with assignment nodes when transitioning to machine code since real world systems do not support ϕ-node semantics [BCH98]. In addition, the SSA form supports the SA property only on the static program graph, but does not support the SA property during rewriting, as the same variable may receive repeated assignments during repetitive executions of loop structures [BCH98].

There are several variants of the SSA form that attempt to incorporate explicit control flow information in the ϕ-function. The Gated Single Assignment (GSA) form incorporates control flow by flowing predicate results directly into the ϕ-node [BMO90]. This variant places γ-nodes of the form $X = \gamma(P, x_1, x_2, x_3, \ldots, x_k)$ at merge points, replacing the ϕ-node form of $X = \phi(x_1, x_2, x_3, \ldots, x_k)$. The P term contains the predicate information, and effectively turns a control flow issue into a data flow issue by explicitly representing control flow information as input values to the γ function. A variation on this concept that extends the GSA form to unstructured control flow is the Thinned Gated Single Assignment form [Hav93]. The γ-nodes of the GSA are extended to directed acyclic graphs of γ-nodes when unstructured control flow is involved. The Static Single Information (SSI) form is an extension to the SSA form that performs variable renaming at both merging and diverging control flow points [Ana99]. The renaming is performed at divergent points using σ-nodes of the form $\sigma(x_1, x_2, x_3, \ldots, x_k) = X$ if at least one of the divergent paths uses the variable [Ana99]. The effect of this node is to distribute the single incoming value for X to the k renamed variables, x_1 through x_k, in use along the k distinct control paths. Since variable renaming is performed at both diverging and converging control flow points, some degree of control flow information is retained. The Interpretable SSA [RWF03] incorporates

operational semantics for ϕ-nodes facilitating efficient interpretation by a stack-oriented virtual machine.

As with the original PDG, explicitly parallel versions of the SSA have been published. The concurrent SSA form was introduced by Lee et. al. [LMP99] to represent parallel programs with structured control flow and using *cobegin/coend* nodes to explicitly express parallelism. The Concurrent SSA form extends ϕ-nodes to apply to control flow merges at *coend* program points, and introduces π-nodes having the same form but containing conflict edges arising from statement interleaving. Resolution of a π-node is non-deterministic due to the concurrency of the incoming edges.

2.9 SFU Form

In this section we describe a new form of the PDG that respects the SFU property. The SFU property for programs requires that, for each variable in a USE statement, exactly one DEF statement will supply a value to the USE statement during program execution. The SFU form of the PDG requires that, for each variable in a USE node, exactly one DEF node may flow a value to the USE node during graph rewriting. The SFU property is not enforced on the program representation, but is upheld during program execution (graph rewriting). The SFU property is achieved by arranging DEF and valve nodes in such a way that all definitions potentially flowing to the same USE are control-wise mutually exclusive. Thus, the control dependence criteria is met for exactly one of the DEF nodes during a single graph rewriting.

The use of *valve* nodes to resolve Def-Order dependence was introduced by Cartwright et. al. [CF89], and expanded upon by Parsons [Par92] to produce the

Semantic PDG. Valve nodes resolve Def-Order dependence without the semantic issues of the ϕ-nodes. Because valve nodes are simply assignment nodes, the basic PDG semantics is preserved. In addition, valve nodes are referentially transparent, support more instances of copy propagation than SSA nodes [BCH98], and allow for discrimination of reaching USE nodes along various control flow paths. Parsons [Par92] presents an algorithm creating PDGs from program text that inserts valve nodes in such a way as to enforce the SFU property during rewriting. In later sections we characterize valve node placement locations to support the SFU property, and then use the graph creation algorithm provided by Parsons as a basis for determining the number of valve nodes generated for typical programs. We use the Semantic PDG as a basis to discuss the SFU property and demonstrate that the Semantic PDG enforces the SFU property during rewriting. We will therefore refer to it in this thesis as the SFU form.

The SSA form does not satisfy the SFU property because the SA property is not upheld during program execution, allowing multiple values to potentially flow to a single USE node for a given variable during execution. An example of an SFU form PDG for Program 2 is presented in Figure 2.4 with a comparable SSA form PDG.

2.10 Control Dependence Structures Within the PDG

This section develops terminology for structures within PDGs that are useful to the discussion of the SFU property. The definitions and theorems of this section formalize intuitive notions that have been presented in previous discussions of dependence graphs. We assume structured control flow throughout. Theorem 5,

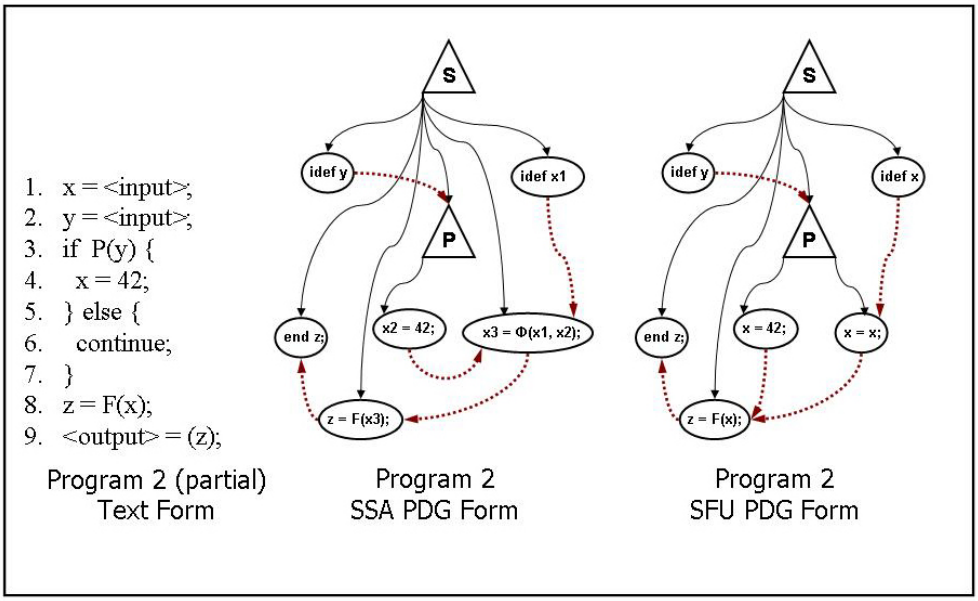
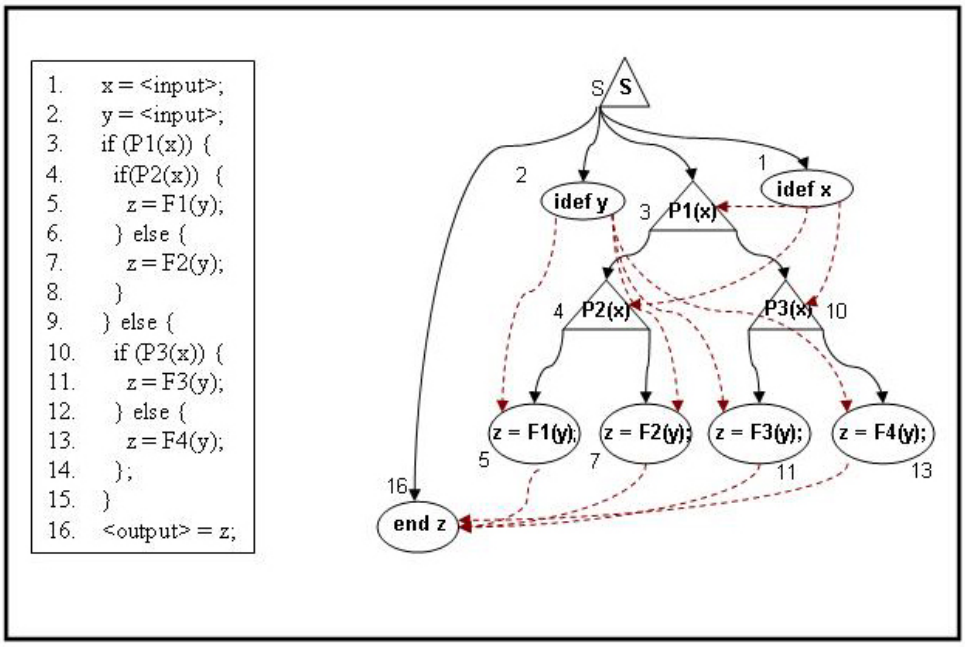


Figure 2.4: PDG forms: SSA and SFU

developed through application of Theorems 1 through 4, is used in Chapter 4 to prove expected and upper bounds on the size of the SOOPDG.

This discussion focuses on predicate nodes and the control dependence edges associated with them. In particular, it is useful to identify the *control points*, defined in Definition 16, that specify the True and False branches of predicate nodes in a PDG. For example, in the PDG program control flow initiates at the *Start* node. We represent this node as a predicate node labeled S, with outgoing control flow edges emanating only from the True branch. We will refer to this control point as S_T. Control points represent decision points in a program. Similarly, chains of these control points connected by control dependence edges specify control dependence paths within a PDG. These control dependence paths will be critical in addressing criteria for valve node placement. Figure 2.5 provides examples of the control dependence structures discussed in this section.



Def 16 Control Point:	S^T, 3^T, and 3^F are examples of Control Points.
Def 17 Control Parent:	$CP(7) = 4$ and $CP(4) = 3$
Def 18 Control Parent Point:	$CPP(7) = 4^F$ and $CPP(4) = 3^T$
Def 20 Control Dependence Ancestor:	S and 3 are ancestors of 10 and 4
Def 21 Control Dependence Path:	For node 7, CDP, $P = S^T, 3^T, 4^F$
Def 22 CDP Prefix:	$P' = S^T$ and $P'' = S^T, 3^T$ are prefixes of CDP P.
Def 23 Common CDP:	Given CDP for nodes 5 and 7, $P_5 = S^T, 3^T, 4^T$ and $P_7 = S^T, 3^T, 4^F$, then the Common CDP is $P_C = S^T, 3^T$
Def 24 Control Dependence Subgraph:	For nodes 5 and 7, CDS, $D = \{\{S, 3, 4\}, \{(S, 3, T), (3, 4, T), (4, 5, T), (4, 7, F)\}\}$
Def 26 CDS Exit Points:	For CDS D above, the exit points are S^F and 3^F. S^F is trivial, as it is not reachable from S^T.

Figure 2.5: Example Control Structures for SFU Form

Definition 16 (Control Point) *Given a PDG, $G = \{ N, E_f, E_c \}$, and a predicate node, $p \in N$, a* control point *is a pairing of the predicate node id and a Boolean value, b. We will denote this pairing as p^b.*

The most primitive control dependence structure present within a PDG is that created by the two nodes involved in the control dependence relationship. These nodes are involved in a parent-child relationship where the control dependence flows from the parent (predicate) node to the child node. For structured control flow, this is a one-to-many relationship as a single predicate node may have many control dependence children, but a node may have only one control parent (excepting the *Start* node which has no control parent). This basic parent-child relationship, defined in Definition 17, will be extended to form control dependence paths composed of control ancestors from the *Start* node to a specific node.

Definition 17 (Control Parent) *Given a PDG, $G = \{ N, E_f, E_c \}$, nodes $n, p \in N$, and control dependence edge $(p, n, b) \in E_c$, we define p to be the* Control Parent *of n, and denote the relationship as $CP(n) = p$.*

We combine the notions of control parent and control point to define the Control Parent Point (Definition 18) in a natural fashion to include the Boolean value of the Control Parent satisfying Control Dependence Criteria for a node. This is useful to specify both the predicate node and branch involved as the parent in the control dependence relationship.

Definition 18 (Control Parent Point) *Given a PDG, $G = \{N, E_f, E_c \}$, and control dependence edge, $(p, n, b) \in E_c$, we define point p^b to be the* Control Parent Point *of n, designated CPP(n) = p^b.*

We extend the notion of a control parent to a control ancestor in a natural manner by considering nested determinations of control parents and developing a shorthand notation as follows:

1. $CP^1(d) = CP(d)$
2. $CP^i(d) = CP(CP^{i-1}(d))$, $i > 1$

The notion of a control dependence ancestor, formally given in Definition 20, is actually a return to the idea of control dominance (Definition 19) in control flow graphs appearing in the traditional PDG literature [CFR89], and which in turn gave rise to the immediate control dependence relations represented in the PDG.

Definition 19 (Control Dominance) *Statement A dominates statement B if A lies on all control paths from start of program execution to B.*

Definition 20 (Control Dependence Ancestor) *Given a PDG,* $G = \{N, E_f, E_c\}$*, and nodes* $p, d \in N$*, we define p to be a control dependence ancestor of d, if there is some value of* $i > 0$ *such that* $CP^i(d) = p$*.*

Chains of control dependence points connected by control dependence edges lead to the concept of the Control Dependence Path, presented in Definition 21. Informally, this path specifies the series of predicate evaluations that must occur from the initiation of program execution to satisfy the control dependence criteria for that node's execution.

Definition 21 (Control Dependence Path) *Given a PDG,* $G = \{N, E_f, E_c\}$*, having* Start *node* $S \in N$*, and a node,* $n \in N - \{S\}$*, we define the Control Dependence Path (CDP) for n to be the tuple of control points,* $P = p_0^{b_0}, p_1^{b_1}, ..., p_r^{b_r}$*, such that:*

1. $p_0^{b_0} = S^T$.

2. $(p_r, n, b_r) \in Ec$ *(i.e. CPP(n) =* $p_r^{b_r}$ *)*

3. for all i, $0 < i < r$, p_i = *CP(* p_{i+1} *)*

We designate the CDP for node n as $CDP(n) = P$.

Referring only to complete CDPS as if they were monolithic entities will not meet the needs of this discussion. We borrow standard notation from formal languages, and use $u, v, w, \ldots$ *etc* as designations of sub-paths within CDPs. Of particular interest to us will be prefixes of control dependence paths, defined in Definition 22. For example, CDP $P = uP'$ where $u \in PREFIX(P)$. The recognition of control dependence prefixes is important as the knowledge that control dependence criteria (Definition 14) has been satisfied for some node, n, implies that control dependence criteria has been satisfied along the entire CDP and every prefix of the CDP(n) (Theorem 1).

Definition 22 (CDP Prefix) *Given the CDP,* $P = p_0^{b_0}, p_1^{b_1}, ..., p_r^{b_r}$, *then any CDP,* $u = p_0^{b_0}, p_1^{b_1}, ..., p_s^{b_s}, 0 \leq s < r$ *is a prefix of P, designated* $u \in PREFIX(P)$.

Theorem 1 (Control Dependence Criteria Met for CDP Prefixes) *Given a PDG, G =* { *N,* E_f, E_c }, *for which control dependence criteria have been met for node* $d_1 \in N$, *and* $\exists$ *node* $d_2 \in N$ *such that CDP(*d_2*)* $\in$ *PREFIX(CDP(*d_1*)), then control dependence criteria have been met for* d_2.

Proof Theorem 1 can be proved directly. Given that Control Dependence Criteria have been met for node d_1, there exists a tuple of control points, CDP(d_1) = P = $(S_T, p_1^{b1}, p_2^{b2}, \ldots, p_r^{br})$ that are tagged as traversed by the control flow. By definition of PREFIX, CPP(d_2) $\in$ P. Thus, CPP(d_2) has executed, satisfying control dependence criteria for d_2. ∎

We now begin to explore slightly more complex control dependence structures that allow discussions of control dependence relationships arising among arbitrary numbers of nodes. We will be interested, ultimately, in establishing control dependence criteria for multiple DEF nodes capable of supplying a value to a single USE node for the same variable. As a first step, we recognize that all nodes in a PDG (excepting the *Start* node) share the control point S_T as the first point in their CDP. This is true by definition of the CDP. The more interesting question, and one pertinent to the SFU discussion, is to determine for a given subset of the nodes within a PDG, $N' \subset N - \{S\}$, the longest CDP common to the set, all of the control points, and control dependence edges. We formally define the Common Control Dependence Path (CCDP) and Control Dependence Subgraph (CDS) below in Definitions 23 and 24, respectively.

Definition 23 (Common Control Dependence Path) *Given a PDG,* $G = \{ N, E_f, E_c \}$ *having* Start *node* S*, and a set of two or more nodes,* $N' \subset N - \{\mathrm{S}\}$*, and the set of CDPs arising from the nodes in N,* $P\text{-}SET = \{P_i | P_i = CDP(n_i) \forall n_i \in N'\}$*, we define the Common Control Dependence Path (CCDP),* P_C*, to be the longest CDP such that control point* $p_i^{b_i} \in P_C$ *iff* $p_i^{b_i} \in P_j \ \forall P_j \in P\text{-}SET$*.*

Theorem 2 (Uniqueness of CCDPs) *Given a PDG,* $G = \{N, E_f, E_c\}$ *having* Start *node* S*, and a set of two or more nodes* $N' \subset N - \{S\}$*, then there is exactly one CCDP defined by the CDPs of node set* N'*.*

Proof We prove Theorem 2 in two parts. In Part 1 we show that there must be at least one CCDP defined by any two or more CDPs and in Part 2 we show that there cannot be more than one CCDP defined by two or more CDPs.

Part 1: There must be at least one CCDP defined by two or more CDPs. This is proved directly by the definition of the CDP (Definition 21), which requires that the first control point comprising a CDP is S^T. Since every CDP starts with control point, S^T, then all CDPs have this point in common and will have a CCDP containing at least S^T. Therefore, there must be at least one CCDP defined by the CDPs of node set N'.

Case 2: There can be no more than one CCDP for the CDPs defined by node set N'. Assume the negation of the previous statement, T. That is "There exist two or more CCDPs for the CDPs defined by node set N'." We consider a set of unique CCDPs, π_1 through π_k. If T is true, then π_1 through π_k are of equal length, and each differs from the others at one or more positions. The length of π_1 through π_k are both greater than zero by definition of the CDP (Definition 21) requiring all CDPs to contain at least S^T. Each pair of CDDPs π_i and π_j, $1 \leq i, j \leq k$, differ at some position, and thus by definition of the CCDP neither π_i nor π_j are CCDPs. Since no two CCDPs in the set can be unique and meet the definition of a CCDP, T cannot be true, and there can be no more than one CCDP for the CDPs defined by node set N'.

Since there must be at least one CCDP (Part 1) and can be no more than one CCDP (Part 2), then there must be exactly one CCDP defined by the CDPs of node set N'. ∎

Because we may be interested in multiple DEF nodes potentially supplying values for the same variable to a single USE node, we will be required to discuss a more complex structure defined by the composition of individual CDPs. Informally we define the Control Dependence Subgraph (CDS) to be the aggregation of the individual CDPs for some set of nodes, N', in the graph. The node set N' defines the CDS, but is not contained within the CDS. While we do not restrict

the node types contained within N' in the definition of the CDS, we are most interested in exploring sets that contain only DEF nodes in this discussion.

Definition 24 (Control Dependence Subgraph) *Given a PDG,* $G = \{ N, E_f, E_c \}$ *having* Start *node* S*, and a subset of* N*,* $N' \subset N - \{S\}$*, we define the Control Dependence Subgraph,* $D = \{ N_d, E_d \}$*, such that:*

1. $N_d = \{ p \mid \exists i, n \text{ where } i \geq 0, n \in N' \text{ such that } CP^i(n) = p \}$
2. $E_d = \{ (p_i, p_j, b) \mid (p_i, p_j, b) \in E_c, \wedge\ p_i, p_j \in N_d \cup N' \}$

Later discussion will require the definition of the difference between two subgraphs (Definition 25). Since graphs and subgraphs are sets, it is natural to use standard set difference as a basis. Informally, given subgraphs $D_1, D_2 \subset G$, the differencing operation will perform standard set difference for nodes and common edges. The remaining edges must be removed if they are "dangling" due to only one of the involved nodes having been removed.

Definition 25 (Differencing of Control Dependence Subgraph) *Given two Control Dependence Subgraphs,* $D_1, D_2 \subset G$*, where* $G = \{N, E_f, E_c\}$*,* $D_1 = \{N_{D_1}, E_{D_1}\}$ *and* $D_2 = \{N_{D_2}, E_{D_2}\}$*, we define the difference operator,* $D_1 - D_2 = \{N_{D_1} - N_{D_2}, E_{D_1} - \{E_{D_2} \cup \{(p_i, p_j, b) | p_i \in N_{D_2} \wedge p_j \in N_{D_1} \wedge (p_i, p_j, b) \in E_c\}\}\}$.

Theorem 3 (CDS Closed under Differencing) *Given* $G = \{N, E_f, E_c\}$ *and two CDS of G,* $D_1 = \{N_1, E_1\}$ *arising from* N_1'*, and* $D_2 = \{N_2, E_2\}$ *arising from* N_2'*, then* $D_3 = \{N_3, E_3\} = D_1 - D_2$ *is also a CDS.*

Proof Theorem 3 is proved directly. We require $D_3 = \{N_3, E_3\}$ to have the properties that

1. $\forall p \in N_3, \exists i, n$ where $i \geq 0, n \in N_1' - N_2'$ such that $CP^i(n) = p$ }

2. $\forall (p_i, p_j, b) \in E_3, (p_i, p_j, b) \in E \wedge p_i, p_j \in N_3$ }

This follows directly from the definitions of D_1, D_2, and the differencing operations of Definition 25. ∎

Of particular interest to the placement of valve nodes will be the boundaries of a given CDS. We refer to the boundary points as *exit points*, and define them formally in Definition 26. Exit points are based on the notion that traversing a control dependence edge extending from these control points causes one to exit the CDS. We will require that for each exit point, p^b, there is exactly one DEF node, d, such that $CPP(d) = p^b$, and then show that this criterion exactly satisfies the SFU property.

Definition 26 (Control Dependence Subgraph Exit Points) *Given a Control Dependence Subgraph $D =$ { N_D, E_D }, we refer to point p^b as an* exit point *if $p \in N_D$ and $\exists$ no p' such that $(p, p', b) \in E_D$.*

Theorem 4 (Criteria for CDS Exit Points) *Given a PDG, $G = \{N, E_f, E_c\}$, a set of DEF nodes, $N' \subset N$ resulting in Control Dependence Subgraph, $D \subset G$, where $D = \{N_D, E_D\}$, and node $p \in N_D$, then point p^b is an exit point iff there does not exist a node $d \in N_D$ having CDP(d) = uCPP(d), where u is a prefix of CDP(d), such that $p^b \in u$.*

Proof Theorem 4 is proved below.

Part 1: If p^b is an exit point, then there does not exist a node $d \in N_D$, having CDP(d) = uCPP(d), where u is a prefix of CDP(d), such that $p^b \in u$.

Proof by contradiction. Assume the negation of Part 1, proposition T. That is, assume p^b is an exit point and there exists a node $d \in N_D$, having CDP(d) = uCPP(d), where u is a prefix of CDP(d), such that $p^b \in u$. By Definition 26, there can be no edge $(p, p', b) \in E_D$ for exit point p^b. CDP(d) = uCPP(d) = $(p_1^b, p_2^b, \ldots, p_k^b, CPP(d))$ by definition of CDP and CDP prefixes. Since $p^b \in u$, p^b must reside at some position $i, 1 \leq i \leq k$. By definition of the CDP, for each position $i, 1 \leq i < k$, edge $(p, p_{i+1}, b) \in E_D$ exists. Similarly, for position k, edge $(p, CPP(d), b) \in E_D$ exists. Thus, p^b cannot be in u. Since T leads to a contradiction, the original statement of Part 1 is true.

Part 2: If there does not exist a node $d \in N_D$, having CDP(d) = uCPP(d), where u is a prefix of CDP(d), such that $p^b \in u$, then p^b is an exit point.

Proof by contradiction. Assume the negation of Part 2, proposition T. That is, assume if there does not exist a node $d \in N_D$, having CDP(d) = uCPP(d), where u is a prefix of CDP(d), such that $p^b \in u$, then p^b is not an exit point. Since $p \in N_D$ and p is not an exit point, then edge $(p, p', b) \in E_D$ exits. By definition of CDS, there is some node $d \in N'$ such that $p^b \in CDP(d)$. By definition of CDP and CDP prefixes, CDP(d) = uCPP(d) = $(p_1^b, p_2^b, \ldots, p_k^b, CPP(d))$. Since there does not exist a node $d \in N_D$, such that $p^b \in u$, then p cannot be in position $i, 1 \leq i \leq k$. Therefore, $p^b = CPP(d)$ and there is no point $p' \in CDP(d)$ such that edge $(p, p', b) \in E_D$ exists. As T leads to a contradiction, the original statement of Part 2 is true. ∎

2.11 Satisfying the SFU Property

The proof of the following theorem uses the PDG structures defined previously to demonstrate that the SFU form of the PDG satisfies the SFU property.

Theorem 5 (SFU Criteria) *Given a PDG, $G = \{N, E_f, E_c\}$, USE node $u \in N$, a set of DEF nodes $N_d = \{d|(d,u) \in E_f)\}$ providing assignment to the same variable, and the Control Dependence Subgraph $C = CDS(N_d) - CDS(u)$, then the following rules must be satisfied at the instant of the execution of node u to maintain the SFU property:*

1. *$|N_d| > 0$.*
2. *If $|N_d| = 1$, then $N_d = \{d\}$, and $CDP(d) = PREFIX(CDP(u))$.*
3. *If $|N_d| > 1$, then for all nodes $d \in N_d$, $CPP(d)$ must be an exit point of C.*
4. *If $|N_d| > 1$, then for all exit points, p^b of C, there is exactly one node $d \in N_d$ such that $CPP(d) = p^b$.*

Proof Theorem 5 is proved below:

Item 1: Proof by contradiction: Assume the negation of Item 1, proposition T. That is, assume *The SFU property can be satisfied with zero DEF nodes in a PDG* is true. Proposition T allows for zero DEF nodes, which allows for zero values to be supplied to the USE node, which contradicts the SFU requirement of exactly one value being supplied to a USE node for each variable in the node. Since T leads to a contradiction, then the original statement in Item 1 is proven.

Item 2: Item 2 follows directly from Theorem 1.

Item 3: Proof by Contradiction: Assume the negation of Item 3, proposition V. That is, assume *the SFU property may be maintained with DEF nodes not at exit points in the CDS, supplying values to a single USE node.* Proposition V leads to a contradiction as follows. By definition, SFU requires that exactly one value flow to the USE node. By definition of CDS, all DEF nodes reside along points on the CDS. Presuming V to be correct, we then construct a PDG such

that a given CDS has DEF nodes not at exit points. We then construct a control flow path from the *Start* node to the USE node that does not encounter a single DEF node, and no value flows to the USE node for the variable in question. This violates the SFU property, implying V cannot be true, and Item 3 is true.

Item 4: Proof by contradiction: Assume the negation of Item 4, proposition W. That is, assume *there is an exit point* p^b *of* C *such that the number of nodes* $d \in N_d$ *satisfying* $CPP(d) = p^b$ *is not one.* The exit point may act as a CPP for either zero nodes or more than one node in N_d. Case 1: There is an exit point p^b on C such that there is no $d \in N_d$ satisfying $CPP(d) = p^b$. Then a control flow path may be formed for USE node u containing p^b such that no value flows to u for the variable in question. This contradicts the SFU property. Case 2: There is an exit point p^b on C such that there is more than one node $d \in N_d$ satisfying $CPP(d) = p^b$. Then a control flow path may be formed for USE node u containing p^b such that more than one DEF node flows values to u for the variable in question. This contradicts the SFU property. Since both cases of W violate the SFU property, W must be false, and Item 4 is true. ∎

The definitions and results of this section will be used in later chapters to discuss and prove properties of the SOOPDG.

3

THE SPARSE OBJECT ORIENTED DEPENDENCE GRAPH

The literature provides examples of extensions to the traditional PDG forms introduced in Chapter 2 that incorporate Object Oriented programming constructs [MMK94, CY96, LH98, CX01]. Zhao [Zha98] and Walkinshaw et al [WRW03] present PDG adaptations capable of static representation of Java programs, including a variation for multi-threaded programs [Zha99]. These representations incorporate OO features through additional node, edge, and subgraph types. The presence of these specialized members introduces syntactic and semantic complications and results in overly large static program representations. The details of the representations given by Zhao and Walkinshaw et al are presented in Chapter 6 and compared to the SOOPDG.

There is a need for a cleaner representation that is amenable to employment of a rewriting semantics capable of modeling program execution. The Sparse Object Oriented PDG (SOOPDG) defined in this chapter is capable of representing single-threaded Java-like programs (J programs). The SOOPDG employs node types similar to the SFU PDG, the same edge types as the SFU PDG, and enforces the SFU property. Slight modifications to the PDG static representation are introduced to incorporate class, interface, and method definitions, and to facilitate parameter passing between call sites and called methods. An associated graph rewriting semantics permits the SOOPDG to model program execution and allows reasoning about dynamic program properties. The rewriting semantics is

higher order to accommodate inheritance, method calls, and dynamic binding. Section 3.1 of this chapter provides an informal discussion of the components of the SOOPDG. Section 3.2 provides a formal definition and examples. The SOOPDG rewriting semantics is presented in Section 4.2.

3.1 SOOPDG Elements

The SOOPDG is an acyclic, directed graph capable of representing J programs using only flow dependence edges, control dependence edges, and a node set similar to the SFU PDG form presented in Chapter 2. An associated rewriting semantics models program execution and allows analysis of dynamic program behavior. The node set represents program *output*, *return*, *declaration*, *assignment*, *input*, *if-then-else* structures, and *while* loops. Nodes may be *primitive*, meaning they do not contain other nodes or subgraphs. Non-primitive nodes may contain nodes representing variable declarations or providing a path for passing parameters to and from methods at calling sites. Non-primitive nodes may also contain subgraphs representing compound statements such as loop bodies and method definitions. Nodes are decorated to designate variable types and access restrictions, to note *static* or *final* keywords, and to track package membership as appropriate. The edge set consists of control and flow dependence edges. Control dependence edges represent program control flow in the traditional way. Flow dependence edges represent traditional data flow, method calls, inheritance, and interface implementation.

Nodes have the general form of *nid:ntype:nexp:nstore:nstatus:ndecor*, where:

1. *nid* uniquely identifies the node,

2. *ntype* designates the node as one of *output*, *idef*, *assignment*, etc.,

3. *nexp* is the node expression containing the actual contents of the node. Node expressions generally correspond to program statements, but may contain additional nodes and subgraphs.

4. *nstore* contains the local store providing a landing pad for values flowing to the node for use in reduction of the node expression.

5. *nstatus* tracks the node's status to designate whether the node has flow dependence criteria satisfied, control dependence criteria satisfied, has been visited, executed, or bypassed.

6. *decor* contains keywords found in the declaration list such as variable type, *static*, *final*, *abstract*, etc.

The *nstore* and *nstatus* fields track node properties that could be inferred through other means. An alternate (and potentially more efficient) implementation of the SOOPDG would not require them. An alternative to depicting a local store explicitly would flow data values directly to the node expression, replacing terms with values. Similarly, an alternative to explicitly depicting node status is to infer the status through the status of incoming and outgoing edges. We retain explicit use of the *nstore* and *nstatus* fields in this thesis for convenience in discussion of graph creation and execution.

The SOOPDG node set is composed of *output*, *idef*, *xfer*, *def*, *assignment*, *predicate*, and *while* nodes as defined in the following subsections. The *output* and *xfer* nodes approximate the function of the *end* node in the traditional PDG forms. Traditional PDGs contain a single *end* node to indicate transfer of some portion of the store to some external environment and termination of computation. This model of program execution doesn't map to OO programs, where

methods encapsulated inside objects routinely transfer portions of local stores to non-local environments with no consideration of overall program termination. We resolve this by replacing the *end* node type with *xfer* and *output* nodes. These new node types facilitate transfer of values to non-local environments and have no bearing on computation termination. *Xfer* nodes transfer values from a local store to a non-local environment that resides within the overall program environment, while *output* nodes represent transfer of store elements to some external environment.

Traditional PDGs utilize the *idef* node as a placeholder for values for program variables not defined in the static program. This notion of a placeholder allowed for an abstract representation of program input supplying values during program execution. The SOOPDG requires a mechanism accepting values not defined in the static program representation from any non-local environment. This non-local environment may be external to the program, such as program input, or may be returned values from called methods. Program input from the external environment is represented in the SOOPDG through *assignment* nodes having the form "y = <input>;" We introduce the *def* node to receive values from a non-local environment. The *xfer* and *def* nodes act in tandem to pass parameters to and from methods and calling sites. Parameter values flow to a non-local environment through *xfer* nodes and are received from non-local environments through *def* nodes. Thus, every method definition will contain a collection of *def* nodes receiving values for formal and informal input parameters as well as a collection of *xfer* nodes transferring returned values (including side effects) to call sites. Similarly, every call site for a given method will contain a matching collection of *xfer* nodes to transfer values to the method *def* nodes, and *def* nodes receiving returned values from the method *xfer* nodes.

SOOPDG *idef* nodes specify initial declaration of variables, and also specify definition of classes and methods. *Idef* nodes representing variable declarations contain type and access information. *Idef* nodes that define classes contain additional *idef* nodes specifying variable and method definitions. *Idef* nodes defining methods contain a subgraph representing the method's computation.

3.1.1 Output Nodes

Output nodes in the SOOPDG correspond to output statements in a J program. Output statements in J take the form "<output> = x;", and designate output of the value of variable "x" to some environment (device) external to the program and program store. The node expression in the corresponding *output* node is of the form "*output* x". Execution of an SOOPDG *output* node does not result in assignments to program variables, and *output* nodes have no outgoing control or flow edges. In this sense, the execution of an SOOPDG *output* node has no effect on the course of the current computation. *Output* edges do play a role in determining program meaning, which is generally presented in terms of observable input/output behavior as discussed in Section 2.1.

The use of *output* nodes within the SOOPDG performs part of the role played by *end* nodes in traditional PDGs. Traditional PDG *end* nodes specify both termination of program computation and a return of the program result to some environment. Within the SOOPDG, *output* nodes no longer result in termination of computation, just as program output statements do not signify the end of program execution. Termination criteria is discussed in the context of graph rewriting in Sections 4.2.

3.1.2 Idef Nodes

Idef nodes in the SOOPDG correspond to initial definitions of variables, methods, classes, and interfaces within a program. When representing initial variable definitions they perform a similar function as in traditional PDGs, each serving as a record of the variable type and properties, and allowing initial values (potentially $\perp$) to be assigned to the variable. Variable declarations in J appear in the form "dec-list x;" where the "dec-list" provides variable type, access restrictions, and may contain *final* and *static* keywords. The *idef* node expression has the form "idef x" and the node's *ndecor* field contains the elements found in the "dec-list".

Idef nodes representing method definitions provide a container for the SOOPDG subgraph representing the method. Method definition occurs in J in the form "dec-list m(args-list) { π_m }" where "dec-list" contains access restrictions, and may contain *abstract* and *final* keywords, "m" is the name of the method, "args-list" is the list of input arguments for the method, and the statement block "π_m" contains the statements in the method. The node expression resulting from this definition takes the form "idef m, $S_m, i_1, i_2, \ldots, i_k, s_1, s_2, \ldots, s_l, x_m$" where "m" is the method identifier, "S_m" is the local *Start* node, "$i_1, i_2, \ldots, i_k$" are *idef* nodes representing the input arguments in "args-list", "$s_1, s_2, \ldots, s_l$" represent the method statements, and "x_m" is the method's *xfer* statement returning a result to the calling context. Additional *xfer* nodes may be required to return side-effected values to a calling context. The node's *ndecor* field contains the elements found in "dec-list". Figure 3.1 supplies an example of a method definition in the SOOPDG from program text. Additional discussion on method definition is given in Subsection 3.1.9.

Idef nodes representing classes and interfaces provide a container node for the representation of variables and methods defined within the class. Class and interface definition occurs in J in the form "dec-list c { π_c }", where "dec-list" contains access restrictions, and may contain *static* and *final* keywords. The variables and methods defined within the class or interface are contained in π_c. The node expression resulting from this definition takes the form "idef c, $i_1, i_2, \ldots, i_k, m_1, m_2, \ldots, m_j$". In this expression "c" is the class identifier, i_1 through i_k represent *idef* nodes for variables contained within π_c, and m_1 through m_j represent *idef* nodes containing the subgraphs representing the methods defined within π_c. The *decor* field contains the elements found in "dec-list". If the class being defined extends another, a flow edge is constructed from the superclass to this class to allow inheritance of features not specifically defined within π_c. Figure 3.2 supplies an example of class definition in the SOOPDG from a snippet of program text.

3.1.3 Xfer Nodes

SOOPDG *xfer* nodes are used to transfer values from a local context to some other context. This occurs when input values are passed from a call site to a method, and when resultant values are transferred back to a call site. The number of *xfer* nodes required at a call site is determined by the number of formal and informal parameters passed to the method. This number can be determined through inspection of the subgraph representing the method being called, as the subgraph will contain a single *def* node for each incoming parameter. During graph rewriting, the *xfer* nodes at the call site and the *def* nodes contained within the method subgraph are connected via flow edges at the time the call is made.

A parameterized renaming of the formal parameter variable names is performed during graph construction to facilitate correct edge construction during rewriting.

Methods contain *xfer* nodes to transfer returned and side-effected values to calling sites. Return statements are of the form "return $F(X_a)$;" and the resulting *xfer* node expression has the form "$F(X_a)$". The J language allows each method to contain at most one "return" statement so that structured control flow is maintained. This makes the correspondence between *xfer* nodes and return statements in typed methods straightforward. Side effects within methods are treated as a special case. A side effect is essentially the return of a value, not to the calling site, but to the program store. The use of a *xfer* node in this case makes the side-effect assignment visible to contexts outside of the method, and allows for the SFU property to be maintained in the case of multiple potential side-effect assignments within a single method. Thus, methods of type *void* will only contain *xfer* nodes corresponding to side-effects within the method. The use of *xfer* nodes is discussed more fully when discussing construction of methods in the SOOPDG in Section 4.1.

3.1.4 Def Nodes

SOOPDG *def* nodes act as placeholders for values to flow into (that is, be defined within) a given context from a non-local context. Methods receive formal and informal input parameter values via *def* nodes. Similarly, *def* nodes are used at call sites to receive returned values from called methods. Formal input parameters for a method are clearly identified in the method's signature, and a single *def* node is required within the method subgraph for each one. For example, method call "o.foo(x, y);" results in a *def* node for "x" and a *def* node for "y" (with matching

xfer nodes at each call site). Additional *def* nodes are required when a method references visible class and instance variables. The presence of the *def* nodes creates a need for a matching *xfer* node at the call site that acts as a conduit for live DEFs to pass to the method. Enforcement of the SFU property ensures exactly one value will pass to the method during program execution.

The SOOPDG also utilizes a *def* node within *while* nodes. An empty *def* node serves as a placeholder for the *while* node contents to be copied into during node expansion (See Subsection 3.1.7 and Section 4.2).

3.1.5 Assignment Nodes

Assignment nodes correspond to assignment of values to variables, instantiation of classes, and side effects due to method calls of type *void*. We also use identity assignments as *valve* nodes to enforce the SFU property within the SOOPDG.

Assignment of values to variables occurs in J in the form "$y = F(x_1, x_2, x_3, \ldots, x_k)$", where function F may be (or contain) a method call. The expression contained in the *assignment* node corresponding to this program statement will have the form "$y = F(x_1, x_2, x_3, \ldots, x_k)$, $d_1, d_2, \ldots, d_j$", where d_1 through d_j are *def* nodes associated with method calls. Each method *def* node contains a set of *xfer* and *def* nodes acting as a template of method input output parameters. The *assignment* node is a DEF node for variable y. The J statement "y = <input>;" is a form of this statement that represents an abstract form of program input. This form corresponds to traditional PDG *idef* nodes and provides a placeholder for variable values assigned at run time as input. Since this form has no method calls, there are no internal *def* nodes required.

Instantiation of classes in J take the form "$o = new\ C(x_1, x_2, x_3, \ldots, x_k)$". The act of class instantiation invokes a constructor method associating a *handle* (memory location) with the object name, setting initial values for instance variables, and potentially updating class variable values. The corresponding *assignment* node contains a *def* node representing the call site for the constructor method. The constructor method *def* node in turn contains appropriate *xfer* and *def* nodes to supply input parameters to the constructor function and receive values for class and instance variables. This format provides a unique DEF node for instance variables belonging to this specific copy of the class. The node expression corresponding to class instantiation has the form "$o = new\ C(x_1, x_2, x_3, \ldots, x_k)$, d". Term d is the *def* node associated with the constructor method.

As discussed in Section 2.2, method calls of type *void* having side effects are permitted in J. These method calls take the form "$o.F(x_1, x_2, x_3, \ldots, x_k)$". The node expression contains only the *def* node associated with the called method, plus additional *def* nodes for side effect as required.

3.1.6 Predicate Nodes

Predicate nodes represent control decision points in the program in the standard way presented in Section 2.6. Predicate statements in J appear in the form "if $P(x_1, x_2, \ldots, x_k)$". Function $P(x_1, x_2, \ldots, x_k)$ returns a Boolean value and may be (or contain) a method call. This results in a *predicate* node expression of the form "$P(x_1, x_2, \ldots, x_k), d_1, d_2, \ldots, d_j$", where d_1 through d_j are *def* nodes corresponding to method calls. The statements contained in the True and False branches of the *if-then-else* structure in J are represented in the SOOPDG as

nodes control dependent on the True and False control points of the *predicate* node.

A unique predicate node called the *Start* node designates the initiation of program control, and is associated with the *main* method. Each method subgraph also contains a local *Start* predicate node to initiate control flow for that method's execution. These *Start* nodes have no incoming control flow edges, and have outgoing edges only on their True control point. The node expression for *Start* nodes consists only of the Boolean value "True". The implication is that control criteria is satisfied for nodes directly control dependent upon the program *Start* node upon initiation of program execution, and for nodes directly control dependent upon a method *Start* node upon execution of a call site of the method.

3.1.7 While Nodes

While nodes represent while loop structures in the SOOPDG. The *while* node must contain the loop predicate, a subgraph representing the loop body, accommodations for loop carried dependences, and the ability to iterate the loop during program execution. J *while* statements occur in the form " while $P(x_1, x_2, \ldots, x_k)$ $\{ \pi_l \}$". The function $P(x_1, x_2, \ldots, x_k)$ returns a Boolean value and may be (or contain) a method call. The loop body is represented by π_l and may contain loop carried dependences. Expressions for *while* nodes in the SOOPDG take the form "$P_l, d_l, s_1, s_2, \ldots, s_j$", where P_l is the loop predicate, d_l is an empty *def* node serving as a placeholder for the "i + 1" loop iteration, and $s_1, s_2, \ldots, s_j$ are the nodes representing the program statements in π_l. The rewriting rules for *while* nodes, discussed further in Section 4.2, execute loop iterations by unrolling the loop once for each iteration executed. The unrolling technique was selected for

SOOPDG rewriting rules to maintain the acyclic nature of the graph and to allow for each node in the loop body to execute at most one time. The *def* node, d_l, is added to the subgraph to provide a placeholder for the "i+1" loop iteration. This node has two uses. The first is to provide a target for forward loop carried flow dependences. The second is to provide a container for the *while* subgraph during unrolling of the *while* node during rewriting.

3.1.8 Control and Flow Edges

The SOOPDG edge set is composed of flow dependence and control dependence edges. Control dependence edges perform the same role as in traditional PDGs and specify which nodes are eligible to execute. Control dependence is determined directly from statements in J using Definition 8, and control dependence edges are placed in the SOOPDG directly from this relationship. Consider a J program having some statement s_d that is control dependent on a predicate statement, s_p. Statement s_d resides in a control block of s_p corresponding to Boolean value b. If SOOPDG node d corresponds to statement s_d, and node p corresponds to s_p, then the control dependence edge in the SOOPDG representing this dependence relationship takes the form "(p, d, b)". Control dependence edges always emanate from predicate nodes, and are associated with a specific control point. The targets of control dependence edges may be any node type.

Flow edges correspond to data flow between DEF and USE statements in a J program, flow of a method between its definition in a class and its use at a call site, and flow of inherited features from a superclass to a subclass. For example, if a flow dependence relationship exists between DEF statement, s_d, and USE statement, s_u in a J program, then edge "(s_d, s_u)" will exist in the SOOPDG.

Flow edge usage is extended beyond traditional data flow to also represent inheritance and the flow of methods from definition to calling sites. In addition to connecting DEF and USE nodes of program variables, flow edges may connect superclasses to subclasses, interface definitions to implementing classes, class definitions to instantiation, and instantiations to USE of variables or methods defined in the class. These each represent a form of DEF-USE relationship directly determinable from J program statements. A single flow edge is placed in the SOOPDG in each case, and is allowed to flow any class member to a USE of the member. Due to this use of flow edges, a single class definition in the static graph is sufficient for any number of instantiations of the class in the graph and any number of calls to a method. Class methods called during graph rewriting (program execution) flow from the class definition to the instantiating node, and then to the call site. Similarly, inherited methods may flow from a superclass to the subclass, then to the node instantiating the class, and finally to the calling site. Flow edges connecting interface definitions to classes implementing them have no use during graph rewriting, but are useful to verify the implementing class has correctly implemented every member of the interface during static program analysis.

3.1.9 Method Subgraphs

Methods in J are represented in the SOOPDG as *idef* nodes containing an SOOPDG subgraph representing the method arguments and statements. An example method subgraph is given in Figure 3.1. Each method subgraph has a local *Start* node designating initiation of control dependence within the method. Local *idef* nodes provide correct data flow of formal input arguments into the method

subgraph. With the exception of the *main* method which has no *end* node, each method contains a local *end* node to represent the effect of the method on the program store. Recall that the original definition of the SFU PDG *end* node is to return the value(s) in the output variable list to the environment. In the case of method subgraphs, the environment is defined as the calling site requiring execution of the method.

There is no incoming control dependence edge to the method's *Start* node. When a method is called during graph rewriting, a copy of the method subgraph flows to the calling site and is incorporated within the SOOPDG at that site. The *Start* node is assigned a control dependence edge such that the control point associated with the calling node becomes the control point associated with the method's local *Start* node. Since the calling node is executing, it clearly has control dependence criteria satisfied. Thus the method's local *Start* node will also have control dependence criteria satisfied upon instantiation. Similarly, flow edges connect the method's *idef* nodes to incoming flow parameters, and the method's *end* node allows results to flow back to the calling site.

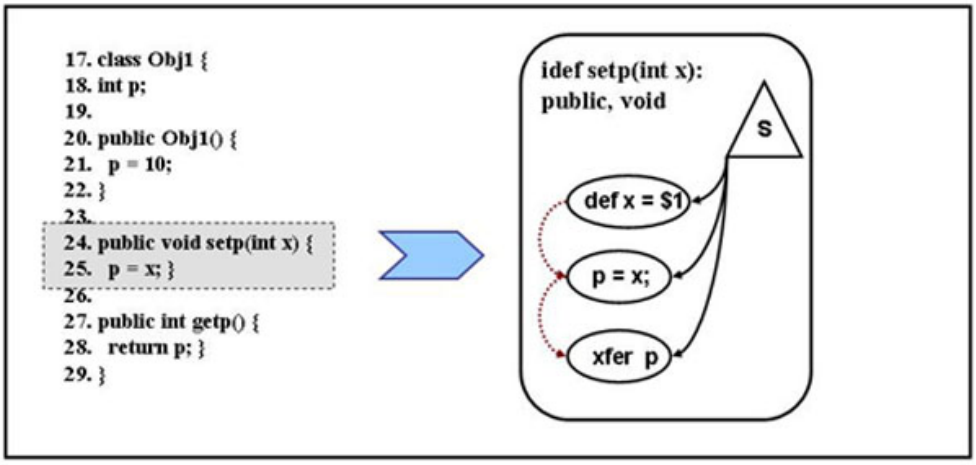


Figure 3.1: Example: Method Definition Using an SOOPDG Subgraph

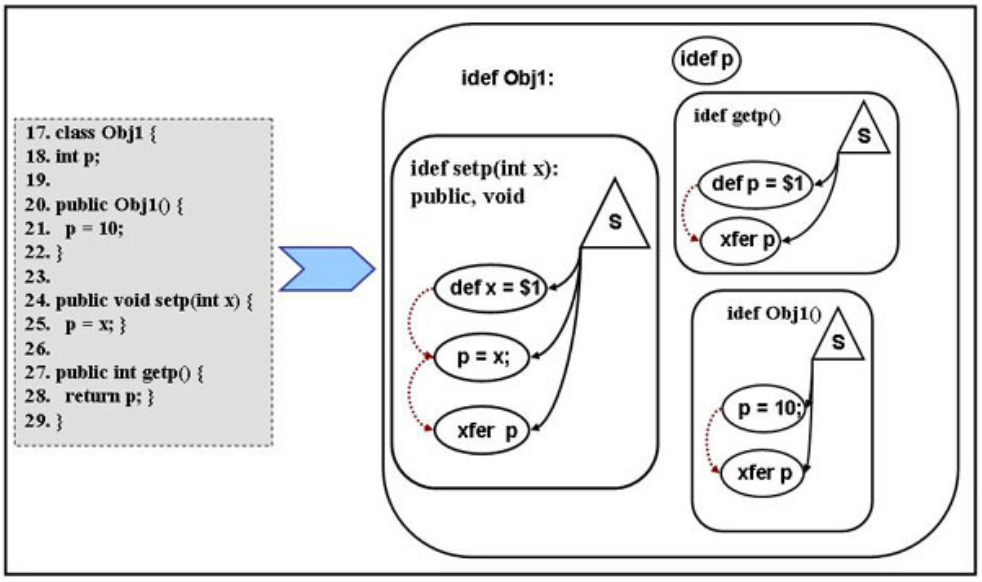


Figure 3.2: Example: Class Definition

3.1.10 Object Representation in the SOOPDG

Objects in J are represented in the same manner as objects in Java. In terms of an execution environment, objects are represented as a store containing the values of all instance variables and information sufficient to retain access to class variables and methods. The *state* of an object at any point in program execution is simply the value of all variables associated with the object. We refer to a value pointing to a unique object or class as a "handle", which acts as a pointer to the actual object in the execution environment. Language J allows variables to have an object type in exactly the same manner as Java, and assignments to these object variables represents the assignment of an object handle. We designate the release of an object handle as assignment to $\perp$. The ability to assign handles to variables representing objects has two major effects in the context of this thesis. The first is that determination of DEF and USE cannot be determined

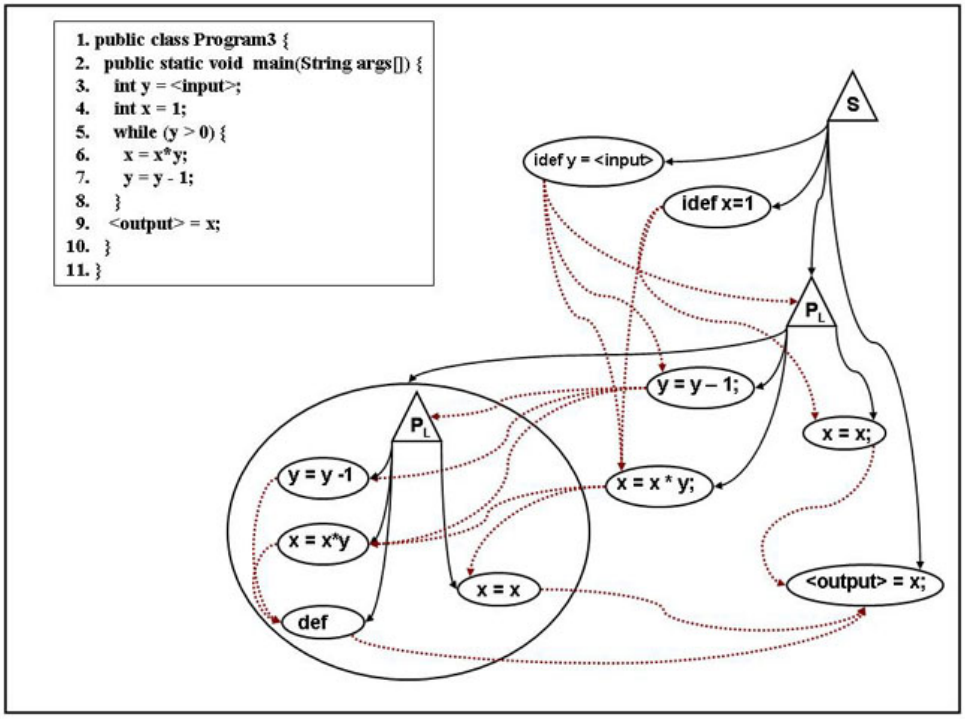


Figure 3.3: Example: While Node

solely by variable name. For example, assuming normal liveness criteria has been met, DEF statement "o.x = 5;" supplies a value to USE statement "y = o.x;" only if the handle (value) of "o" is the same during the execution of both statements. The second major effect is the presence object aliasing as described in Section 2.3. Aliasing may result in multiple object variables referring to the same actual object (memory space), which may result in changes to object state that are not determinable purely through syntactic program analysis. Aliasing that occurs in a predicate is referred to as the *may alias* condition [FYD06] and acts as a mechanism to introduce Def-Order dependences on class andinstance variables. The *may alias* condition also complicates dynamic program behaviors such as garbage collection. For example, it may be unclear from a static program

Table 3.1: SOOPDG Node and Edge Set Summary

Feature	Type	Usage
Nodes	Start	Predicate node specifying the start of control for methods and program (*main* method).
	output	Node specifying program output (<output> = x;) or return of method result to calling environment (return x;).
	xfer	Transfers values from current context to a non-local context.
	def	Recceives values from a non-local context to the current context.
	idef	Contains declaration of variables.
		Contains definition of Classes.
		Contains definition of Interfaces.
	assignment	Corresponds to assignment statements, (y = F(x)), object instantiation (c = Obj1();), and methods with side effects (c.setx();).
	predicate	Corresponds to program decision points (if P(x)).
	while	Corresponds to a while loop (while P(x) π_L). The loop body is represented as a subgraph containing a def node for future iterations, and Start node representing the loop predicate.
Edges	flow	Explicitly represents flow dependence as in traditional PDG.
		Explicitly represents flow of methods between definition and call sites.
		Represents inheritance between classes and interfaces.
	control	Explicitly represents control dependence as in traditional PDG.

representation when an object involved in a *may alias* condition has had all pointers to it removed and is thus eligible for garbage collection.

Objects in the SOOPDG are represented by the variables associated with the object and a handle designating what variables belong to the object. The value of the handle is not important in the SOOPDG representation, but the ability to disambiguate object references is. The SOOPDG uses integer value handles assigned to the object name as a proxy for actual handle values used in a real world execution environment. Thus, object names (e.g. $o_1, o_2, \ldots$) become variables taking on integer values. These values flow through flow dependence edges and the SFU property ensures that a single handle value will flow to each object reference during program execution. Aliasing is supported through assignment ($o_2 = o_1$), with assignments residing in predicates requiring placement of valve

nodes to enforce the SFU property. During program execution (graph rewriting), DEF nodes assigning to variable "$o_i.y$" may flow values to USE nodes of "$o_j.y$" (where i may equal j) only if $o_i = o_j$ at the time of execution.

The SOOPDG does not rely on a central store, as each node contains a local store sufficient to allow the node to execute. Variables of instantiated objects are distributed in keeping with this usage of local stores, and are bound to the object through the value of the object name (variable handle). Object state at any point during computation is defined by the values provided by the distributed DEF nodes of the variables comprising the object. In typical Java-like programs, objects are passed to methods as input. In actuality, the object handle is passed as input and the object's instance variables are colocated and accessed through the use of the handle value. In the SOOPDG the passing of an object to a method potentially requires incoming flow edges from multiple DEF nodes providing values for the object variables for use in the method.

3.2 Formal Definition of the SOOPDG

The SOOPDG is formally presented in Definition 27. We use *nid* for node identifications, *pid* for predicate node identifications, *nexp* for node expressions, x, y, and *var* for variables, o for variables representing instantiated classes and interfaces, C for class names, and m for methods. A summary of the SOOPDG node and edge sets and their functions is presented in Table 3.1. The complete SOOPDG for Program 1, first presented in Figure 2.1, is presented in Figure 3.4.

Definition 27 (SOOPDG) *The SOOPDG is an acyclic, directed graph,* $G = \{N, E_f, E_c\}$*, where* N *is a set of nodes corresponding to program statements in*

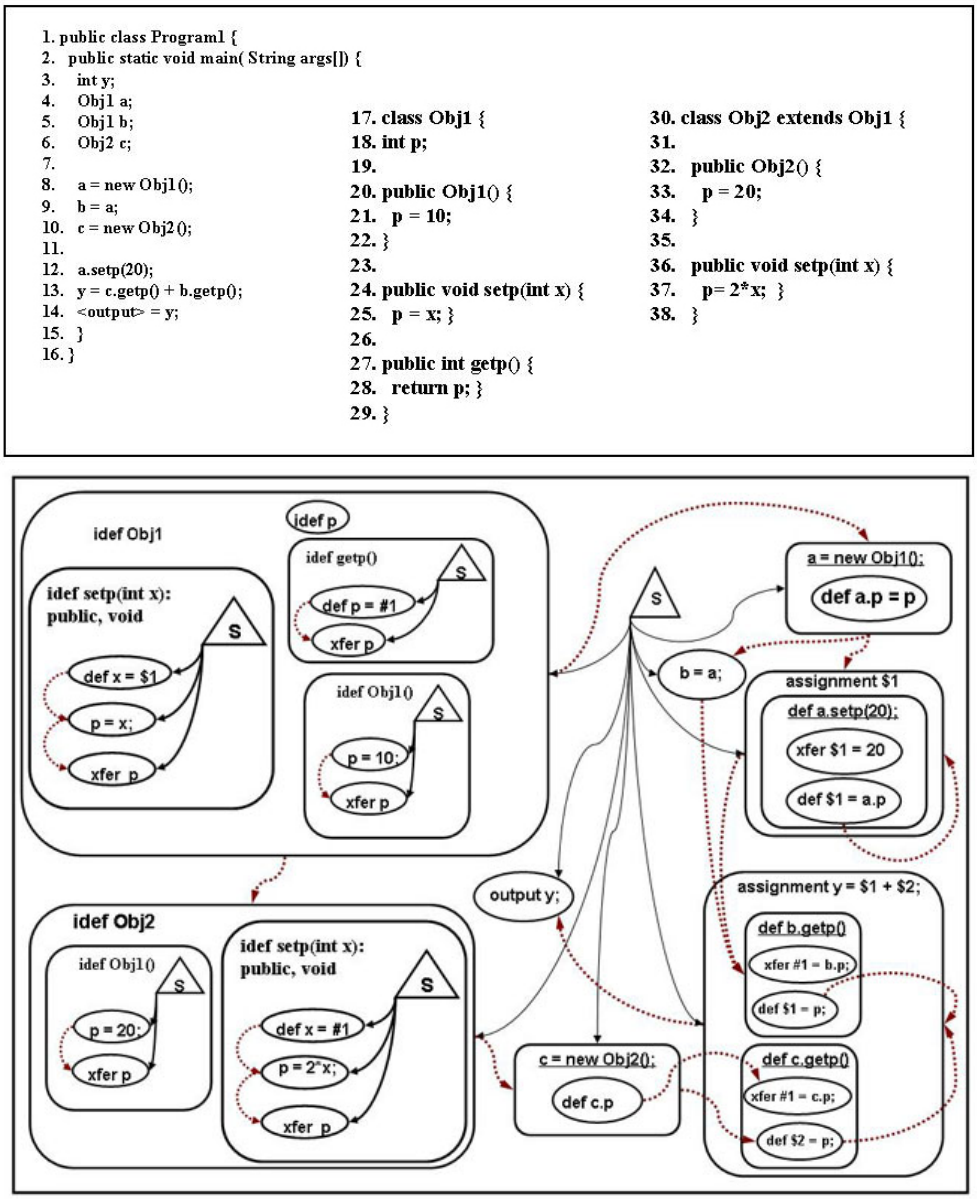


Figure 3.4: SOOPDG for Sample Program 1

J, E_f is a set of flow dependence edges, and E_c is a set of control dependence edges. A node, $n \in N$, has the form nid:ntype:nexp:nstore:nstatus:ndecor *where* nid *is a unique node identifier,* ntype *and* nexpr *are discussed below,* nstore *is the local store,* nstatus *specifies the node status as being one of "unvisited", "flow dependence met", " control dependence met", "ready", "executed", or "bypassed", and* ndecor *contains modifiers found in declaration statements. Flow edges have the form (n_d, n_u), where n_d is a DEF node and n_u is a USE node. Control dependence edges have the form (p, n, b), where n is the nid of a node control dependent on control point p^b.*

1. *Given a flow dependence between a DEF statement i and USE statement j in a J program, and corresponding nodes nid_i and nid_j in G, then E_F contains edge (nid_i, nid_j).*

2. *Given statement i control dependent on control point p^b in a J program, and corresponding nodes nid_i and nid_p in G, the E_C contains edge (nid_p, nid_i, b).*

3. *Given J statement "<output> = x;", then the corresponding node will have* ntype = output, *and* nexpr = "x''.

4. *Given variable declaration "dec-list $x = F(X_a)$;" or "dec-list x;" in a J program, then the corresponding node will have* ntype = idef, *and* nexpr = "$x = F(X_a), d_1, d_2, \ldots d_k''$ *(for "dec-list $x = F(X_a)$;") or "$x = \perp''$ (for "dec-list x;;"). Terms $d_1, d_2, \ldots d_k$ represent* def *nodes associated with methods called in $F(X_a)$.*

5. *Given class definition "dec-list C { π_c }" in a J program, then* ntype = idef, *and* nexpr = "$i_1, i_2, \ldots, i_j, m_1, m_2, \ldots, m_k''$, *where $i_1, i_2, \ldots, i_j$ are* def *nodes for the j variables defined in the class, and $m_1, m_2, \ldots, m_k$, are* def *nodes for the k methods defined in the class.*

6. *Given method definition "*$dec - listm(x_1, x_2, \ldots, x_k)\{p_m\}$*" in a J program, then* ntype = def*, and* nexpr = "$S_m, i_1, i_2, \ldots, i_k, p_m, x_1, x_2, \ldots, x_l''$*, where* S_m *is the method* Start *node,* $i_1, i_2, \ldots, i_k$ *are the* def *nodes for the k method arguments, and* p_m *is the method subgraph defined in program statements* $\{p_m\}$*, and* $x_1, x_2, \ldots, x_l$ *represent* xfer *nodes returning the method result and any side effects to a calling site.*

7. *Given assignment statement "*$y = F(x_1, x_2, \ldots, x_k);$*" in a J program, then* ntype = assignment*, and* nexp = "$y = F(x_1, x_2, \ldots, x_k), d_1, d_2, \ldots, d_j)''$*, where* $d_1, d_2, \ldots, d_j$ *are the* def *nodes required for any methods called in the statement.*

8. *Given assignment statement, "*$o = newC(x_1, x_2, \ldots, x_k);$*" in a J program, then* ntype = assignment*, and* nexp = "$o = C(x_1, x_2, \ldots, x_k), d_1, d_2, \ldots, d_j'',$ *where* $d_1, d_2, \ldots, d_j$ *are the* def *nodes required for any methods called in the statement.*

9. *Given a statement calling a method of type void, "*$o.F(x_1, x_2, \ldots, x_k);$*" in a J program, then* ntype = assignment*, and* nexp = "$o.F(x_1, x_2, \ldots, x_k), d_o, d_1, d_2, \ldots, d_j'',$ *where* d_0 *is the* def *node for the called method, and* $d_1, d_2, \ldots, d_j$ *are the* def *nodes required for other methods called within the statement.*

10. *Given J predicate statement "if* $P(x_1, x_2, \ldots, x_k)$*", then* ntype = predicate *and* nexpr = "$P(x_1, x_2, \ldots, x_k), d_1, d_2, \ldots d_j'',$ *where* $d_1, d_2, \ldots, d_j$ *are the* def *nodes required for any methods called within the statement.*

11. *Given J while statement "while* $P(x_1, x_2, \ldots, x_k)\{p_l\}$*", then* ntype = while*, and* nexpr = "$S_l, p_l, d_{loop}, d_1, d_2, \ldots, d_j'',$ *where* S_l *is the loop* Start *node representing the loop predicate,* π_l *is the subgraph representing the loop body,* d_{loop} *is the empty* def *node used to replicate the loop in the "i+1" iteration.*

3.3 SOOPDG Examples

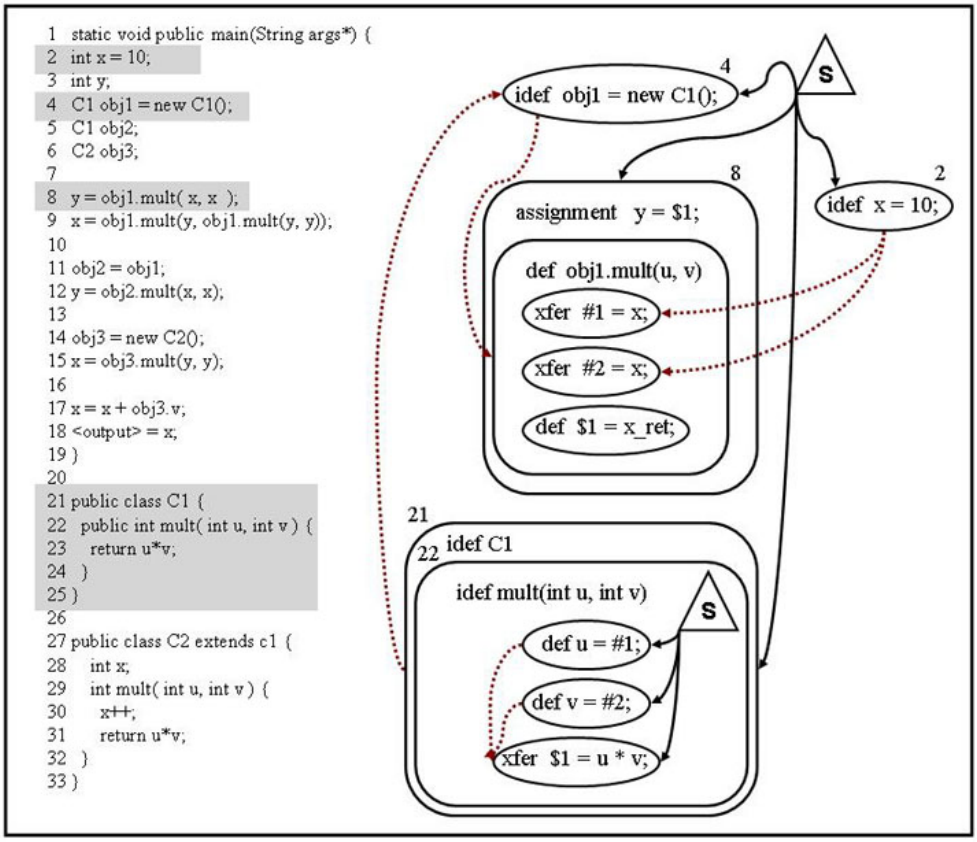


Figure 3.5: Example: Depiction of Call Sites

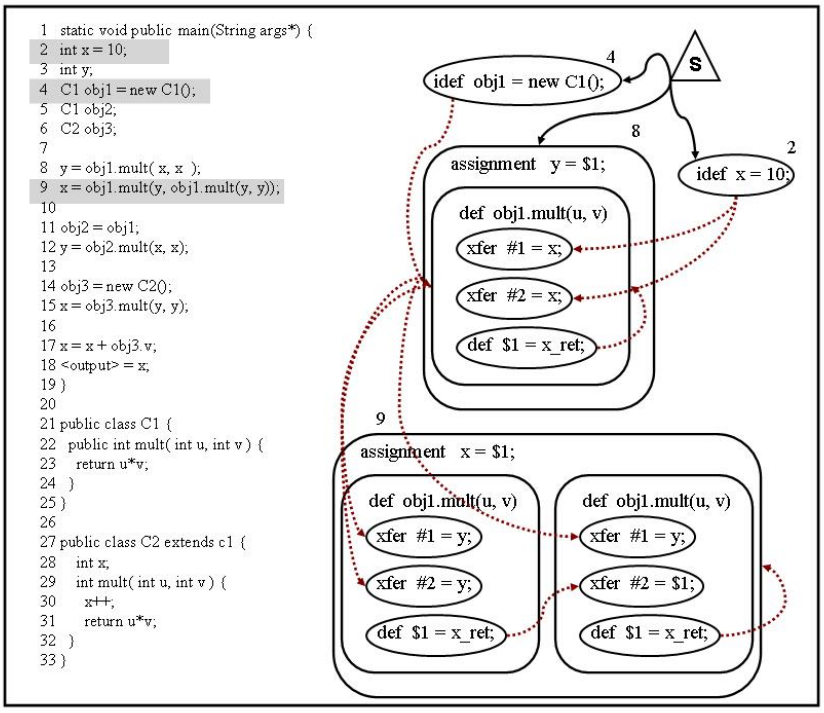


Figure 3.6: Example: Depiction of Multiple Call Sites Respecting Call Sequence

4

SOOPDG CREATION AND REWRITING SEMANTICS

This chapter presents the MakeG algorithm, which creates an SOOPDG from an arbitrary program written in the J language, and the ExecuteG algorithm, which performs program rewriting on the SOOPDG. During the discussion of MakeG we presume the program being operated on is syntactically correct. This is not a restriction as the program may be passed through a syntax-checker prior to development of the SOOPDG form. We do not define the syntax-checker as this is not the focus of the thesis and the process has been well understood for some time. A discussion on the topic is presented in Aho et al [ALS07]. This chapter does not address program analysis algorithms on the static graph structure as these are presented in Chapter 5.

The ExecuteG algorithm provides a semantics describing execution of a program through graph rewriting operations. The purpose of developing rewriting semantics is to allow program analysis on program behavior and components in a dynamic environment. We do not propose to use the SOOPDG as the basis for a Java Virtual Machine or runtime environment, though techniques have been proposed to utilize traditional PDGs to facilitate runtime optimizations within Java-like programs [GCH00, RSE04, VE01]. Identifying similar opportunities for dynamic program analysis using the SOOPDG is a goal for future research as discussed in Chapter 8.

4.1 Graph Creation Algorithm, MakeG

The graph creation algorithm, MakeG, translates code written in text format in the J language into the SOOPDG format. While Java programs (and therefore J programs) typically consist of multiple files containing source code, for convenience we assume all classes are contained in a single file or list. This is acceptable as the individual components of programs consisting of multiple files must all be visible to the compiler during the compilation process. For example, in Java, all participating classes must be made visible to the Java bytecode compiler by one of three methods. They are either defined within standard Java base classes available to the compiler (e.g. Math or System classes), made available through the "import" keyword, or provided in the source files through the "-classpath" compiler parameter. In any case, the bytecode compiler does not distinguish the source of the class definition and acts as though a single source file is being processed. Thus, the original program is composed as a collection of classes represented as a single list of program statements, Π = "dec-list c_1 $\{\pi_1\}$" "dec-list c_2 $\{\pi_2\}$" ... "dec-list c_k $\{\pi_k\}$".

The MakeG algorithm processes individual program statements to incrementally build the SOOPDG. A single program statement is generally represented by a single graph node, though compound program statements result in multiple nodes or nodes containing subgraphs. A single program statement may also require representation by multiple nodes when accommodating parameter passing and side effects resulting from method calls. Flow and control dependence edges are created in conjunction with new nodes to connect these to the existing graph. The entire graph is created in one pass through the text program. It is feasible to employ post processing techniques such as dead code elimination on the completed graph to make the graph representation more efficient. We do

not present detailed algorithms for these techniques as they are not central to the thesis. Development of such routines is reserved for future work. Selected algorithms that discover program features (e.g. program slicing and call chain discovery) or support compiler optimizations are presented in Chapter 5.

Some SOOPDG elements have no direct correspondence to program statements. As discussed in Chapter 2, *valve* nodes have no direct correlation with program statements but are added to the graph to resolve def-order dependences. Similarly, *xfer* and *def* nodes resulting from method calls do not directly represent program statements, but provide a pathway between contexts. The addition of these nodes results in a worse case complexity is $O(n^2)$ for size and construction cost (See Sections 4.1.4, 4.1.5, and 4.1.6).

MakeG is a recursive algorithm. Statements that are control dependent on the same control parent point are processed iteratively. When compound statements are encountered, the MakeG algorithm recursively calls itself on the compound statement. For example, when a *predicate* statement is encountered, each control branch is explored recursively in turn (parallel processing is possible, but not considered in this thesis). This results in a depth-first creation of the SOOPDG with respect to E_c. This is not to say that the deepest control dependence paths are created first; only that the paths are built depth-first as they are encountered. Compound statements requiring recursive MakeG calls are class bodies (π_c), method definitions (π_m), predicate control branches (π_T and π_F), and loop bodies (π_l).

MakeG requires several supporting data structures and helper functions. A worklist, W, contains the statements from program Π that require processing. The MakeG algorithm processes the statements in W, and terminates when W is empty, returning the graph G. The worklist is populated by the *getClass(c,* Π*)*

and *getMethod(c, m,* Π*)* functions. We do not provide complete algorithms for these functions. Notionally, they operate completely at the syntactical level such that getClass(c, Π) returns "dec-list c { π_c }", and getMethod(c, m, Π) returns "dec-list m(X_f) { π_m }". The MakeG algorithm also makes use of *templates* when a specific subgraph pattern is repeatedly added to the graph being constructed. For example, methods require a set of nodes providing pathways for parameter passing and another for side effects. The collection of nodes and edges added to the graph may be constructed for the first call site encountered, and retained as a template to allow future call sites to be processed more efficiently.

Initially, W contains the *main* method designated as the point of entry for program execution by the caller of the MakeG algorithm. A reference to a new class results in the class being added to G through a recursive call to MakeG. The new class may be encountered during object instantiation or through reference to a static member of the class. The worklist of statements used during the recursive call is populated by a *getClass(c,* Π*)* call. The statements in c are processed within the recursive call before processing the statement containing the instantiation. The worklist used in recursive MakeG calls that process method, predicate, and loop bodies is populated in a similar manner.

The MakeG algorithm tracks the active control point, $C = (pid, b)$, to correctly create control dependence edges. A liveness set, L, identifies DEF nodes that potentially supply values to USE nodes, and an alias set, A, tracks object aliasing. The L and A sets are required to create flow edges appropriately. Their uses are discussed more fully below.

4.1.1 Variable Liveness Analysis

Throughout the creation of the SOOPDG, variable liveness is updated as each program statement is processed or as statements dependent on specific control points are processed. This liveness information ties to control dependence to account for dominance frontiers changing as control structures are entered and exited. We designate the data structure tracking variable liveness as L. L provides the active DEF nodes for each variable and associated program control points as they are processed by the MakeG algorithm. L is a set of *liveness lists* tracking DEF nodes for each variable and program control point as defined in Definition 28. Final variables are noted in the liveness set with the keyword "final" appended to the identifier for the single DEF node providing a value for the variable. The entire set, L, is defined in Definition 29. Updates to L, described in detail below, are performed using standard set difference, union, and intersection operations.

Definition 28 (Liveness List) *A liveness list, $l_C^{x_i} = \{d_1, d_2, \ldots, d_k\}$ is a set of nodes, $d_i, 1 \leq i \leq k$, where d_i is a DEF node capable of supplying a value for variable x_i for the control point C. The notation { d:final } represents a definition that may not be superceded.*

Definition 29 (Liveness Set) *The liveness set for program P, L, is a set of all active liveness lists at any point in construction of the SOOPDG.*

The MakeG algorithm requires a liveness list, L, as input. When initiating the processing of a *main* method, L is initialized to the empty set. When initiating the processing of the compound statements associated with a specific control point, C_b', L_{C_b} is initialized to the current L. Similarly, when processing the

compound statement representing a loop body, L_{loop} is initialized to the current L.

Rules for updating L are straightforward. While processing statements that are control dependent upon control point, C, the processing of any DEF node for a program variable kills all preceding definitions for that variable for the remaining statements dependent on that control point. Therefore, when encountering DEF node d for program variable v along control point C, the liveness set is updated such that $l_C^x = \{d\}$. Definitions occurring along the true or false branch of a predicate statement may reach beyond the compound statements comprising the predicate. This is reflected in L by defining the liveness set continuing forward from a predicate to be the union of the liveness lists for each branch. The MakeG algorithm as presented contains no parallel operations. This implies that the algorithm is exploring program statements that are control dependent on one control point at any given time. This implies that, in the worse case, L is comprised of at most $O(n)$ lists (with n being the texts ize of the program) at any given time during the MakeG algorithm. Searching and maintaining L can be acheived in $O(logn)$ time for each entry processed, and thus the searching and maintaining of L does not dominate the $O(n^2)$ worse case cost of the MakeG algorithm shown in Section 4.1.6.

Object aliasing is not explicitly represented in L, though aliasing effects are taken into account. For example, if objects o_1 and o_2 are aliased and a node is being constructed that is a USE of an instance variable in either object, then incoming flow edges must connect all live DEF nodes for both objects to this USE node. Object aliasing is maintained in a separate alias set described below.

Figure 4.1 provides an example demonstrating how the update rules affect the liveness set while entering and exiting a control structure. This example

demonstrates, for a single variable x, how control dependence is incorporated in L, and how control points become associated with specific DEF statements in L. The control point in effect at the beginning of the snippet is the *Start* node's True branch, S_T. When a predicate statement is found (3), the True control branch is explored first, followed by the False branch. Upon entering the True branch, the active control point is 3_T. Any DEF statement encountered along the branch (4) supplies the reaching definition (i.e. is "live") for following USE statements along the same control point (5). The liveness set is updated to reflect this. Upon entering the False branch, the active control point is 3_F, and DEF statements on this path are also recorded in the liveness set. Upon leaving the False branch, control reverts to S_T, but the DEF nodes discovered along each predicate branch are still live. Since DEF nodes occurred along both branches, the previous DEF statements associated with S_T can no longer supply reaching definitions. This is reflected in L by removing existing DEF statements associated with S_T and adding in the union of those associated with 3_T and 3_F. The liveness set for variable y beyond statement 2 is the same for control points S_T, 3_T, and 3_F throughout.

4.1.2 Object Alias Analysis

The MakeG algorithm tracks alias relationships using the alias set, A. The alias set lists all handles referring to each object instantiated in the program. The alias set is a tree of depth one, containing an abstract reference to the actual object (memory space) at the root, and all program references that may point to the actual object as leaves. The reference to the single, actual object at the root of the tree provides a simple mechanism to carry data flow through the

LINE	STATEMENT	CP	L (for x only)
0	*Start*	S_T	$L = \phi$
1	x = <input>;	S_T	$L = \{l^x_{S_T}$: { 1 }$\}$
2	y = <input>;	S_T	$L = \{l^x_{S_T}$: { 1 }$\}$
3	if (P()) {	S_T	$L = \{l^x_{S_T}$: { 1 }$\}$
4	x = 42;	3_T	$L = \{l^x_{S_T}$: { 1 }, $l^x_{3_T}$: { 4 }$\}$
5	z = x / 2;	3_T	$L = \{l^x_{S_T}$: { 1 }, $l^x_{3_T}$: { 4 }$\}$
6	} else {	3_F	$L = \{l^x_{S_T}$: { 1 }, $l^x_{3_T}$: { 4 }, $l^x_{3_F}$: { 1 }$\}$
7	x = 17*y;	3_F	$L = \{l^x_{S_T}$: { 1 }, $l^x_{3_T}$: { 4 }, $l^x_{3_F}$: { 7 }$\}$
8	}	S_T	$L = \{l^x_{S_T}$: { $l^x_{3_T} \cup l^x_{3_F}$ } = { 4, 7 }$\}$
9	return x;	S_T	$L = \{l^x_{S_T}$: { 4, 7 } }

Figure 4.1: Liveness Example: Effect of Predicate Nodes

object itself, rather than through potentially numerous handles. When an object is instantiated in the program, a tree is created, "O_i: o", where "O_i" represents the actual object, while "o" represents the handle referring to the object. When aliasing occurs, the additional object handles are added as additional leaves. When the aliasing is no longer valid, the handle is removed. Thus, the alias set presents, for each object instantiated in the program, all possible ways to refer to that object.

Aliases have two main effects on the building of the graph. First, they complicate the construction of flow edges by increasing the number of potential DEF nodes supplying a value to a given USE. Secondly, Def-order relationships may arise from a may-alias condition. Def-order relationships that arise in this way require that valve nodes be instantiated within the predicate acting as a control parent to the program statement causing the may-alias condition.

Definition 30 (Alias Set) *The alias set for program* Π*,* A*, is a set of trees, each of depth 1. Each tree contains the reference object at the root acting as a parent to each alias. We denote this structure as* $(O_R : o_1, o_2, \ldots, o_k)$ *and require that any reference to* o_i*,* $1 \leq i \leq k$*, is equivalent to any other reference* o_j*,* $1 \leq j \leq k$*.*

4.1.3 Description of the MakeG Algorithm

This section provides an informal description of the MakeG algorithm, followed by a formal definition. The MakeG algorithm requires as input a program, Π = "dec-list c_1 $\{\pi_1\}$" "dec-list c_2 $\{\pi_2\}$" ... "dec-list c_k $\{\pi_k\}$", a sequence of statements from Π comprising the worklist, W, an existing (potentially empty) SOOPDG, G; an initial control point, C = (pid, b); and initial alias and liveness sets, A and L. We assume the existence of utility functions getClass(c, Π) and getMethod(c, m, Π) to populate the worklist from the list of program statements. Initially, W contains only the program *main* method. Other program classes are extracted from Π and processed as they are referenced by statements already residing in W. The liveness lists provide a mechanism to track which classes have been instantiated in G to prevent duplications.

The SOOPDG requires a distinct *Start* node specifying the entry point for initiation of program execution. J programs are collections of classes, and it is possible for such a collection to contain more than one *main* method. As an SOOPDG contains one *Start* node for each method defined within the program, the MakeG algorithm must generate a distinct *Start* node representing the start of program execution. This is accomplished by initializing W with a *main* method specified by the user, "W = getMethod(c, *main*, Π);". Graph creation continues

until all statements within this *main* method, plus all statements within classes potentially executed or referenced as a result of the method, have been processed. References to object instantiation or static elements of classes not previously encountered results in a recursive call to MakeG with W seeded with the new class. In the resulting graph, G, all method *Start* nodes are encapsulated within *idef* nodes with the exception of the single *Start* node corresponding to the originally specified *main* method. The location of this *Start* node external to an *idef* node designates it as the initiation of program execution.

Statements containing expressions with method calls result in multiple nodes being added to G. These additional nodes provide a mechanism for orderly transfer of parameters in and out of the called method. Recall that *xfer* nodes supply values to another context, and *def* nodes receive values from another context. Thus, a statement containing a method call results in a node representing the statement plus the *xfer* nodes required to provide input parameters to the called methods and *def* nodes required to receive results (including side effects). We eliminate ambiguity as to which nodes are associated with specific calling sites through a simple renaming scheme. Each call site receives a specific variable name ($1, $2, $3, ...) and the method *def* node representing the returned value is associated with a specific site through an assignment to the renamed variable (e.g. "$1 = x_{ret}"). A similar renaming scheme (#1, #2, #3, ...) ensures that parameters are passed to the method in an orderly and unambiguous fashion. The use of this parameterized naming scheme results in a set of *xfer* and *def* nodes that serve as an interface between the calling site and method that can be connected to a method subgraph at runtime. The scheme also respects expression precedence semantics in that methods affected by methods executing prior to them in the expression will have appropriate values flowed to them from the

def nodes associated with the previous method's call site. A simple example is given in Figure 4.2.

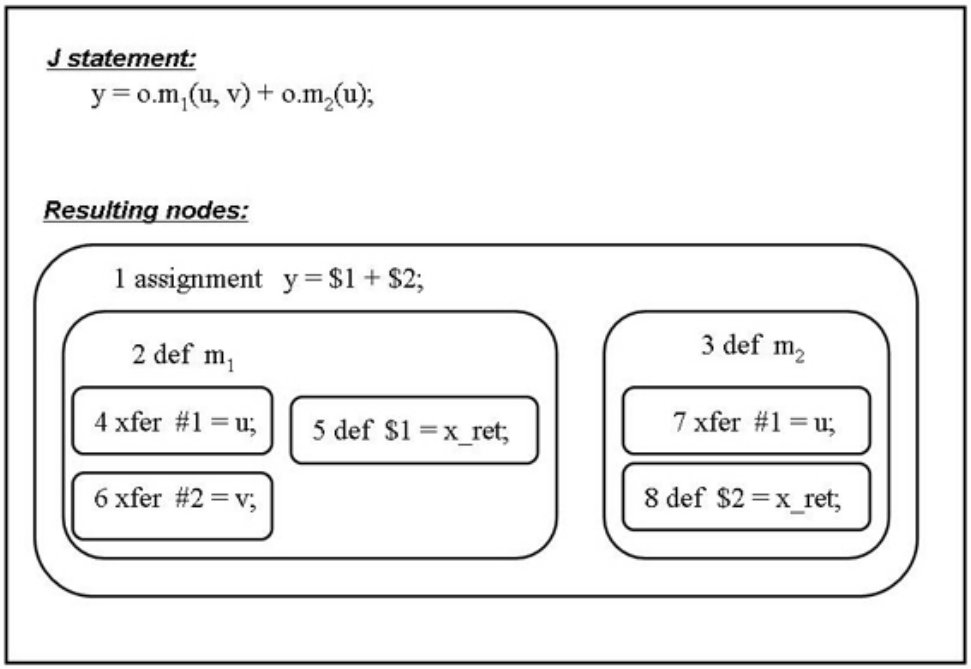


Figure 4.2: Example of Assignment with Multiple Method Call Sites

To relieve clutter from the MakeG algorithm, we employ a utility function performing the actions discussed above. The *processExpression(X_a, nid, L, G)* receives the expression X_a, identification of the current node being created, *nid*, liveness lists, L, and existing SOOPDG G and produces two results. The first, $g.N$ is the set of *xfer* and *def* nodes required by the method calls in the expression X_a. These are easily obtained as follows: for each call site a single *def* node is created. This *def* node contains a single *xfer* node for each *def* node residing within the method called, and a single *def* node for each *xfer* node residing in the method called. The *xfer* nodes in the method include the returned parameter as well as side effects. The existence of the nodes is found by scanning the *idef* node

in G defining the method. The second result, $g.E_f$, is a set of flow edges required to provide input to variables within X_a, as well as any flow edges that may arise between *def* nodes created due to method calls. These edges are created using the *nid* and L provided. The previous steps need only be performed the first time a method is encountered, as the set of nodes and edges created are maintained in a template and re-used.

Finally, MakeG updates L so that *def* nodes representing side effects are recognized as the current live DEF for the side affected variable. We will call this utility function using the form "g = processExpression(X_a, nid, L, G)", and refer to the set of returned nodes as $g.N$, and the set of returned flow edges as $g.E_f$.

MakeG adds nodes and edges to G through iterative processing of the statements in W, resulting in a complete representation of the program. The processing of class definitions, method statements, true and false predicate branches, and while loops require that MakeG be called recursively. In these cases the program statements input to the MakeG algorithm are the compound statements composing the class, method, branch, or loop bodies. The control point, alias set, and liveness set are initialized appropriately before the recursive call is made. When the MakeG algorithm is initiated at the program level the following initializations take place: $G = \{N, E_c, E_f\} = \{\{Start\}, \phi, \phi\}$, $C = (Start, True)$, and $L = \phi$. We present a discussion of steps taken to process statement types, followed by the MakeG algorithm in Tables 4.3 - 4.16 at the end of this section.

Output statements result in an *output* node being formed as follows. Output statements are of the form "<output> = x;", and result in an *output* node of the form "output x". An incoming control edge connects the new node to the

current control point, and incoming flow edges are created from each live DEF node for x. This process is presented in Table 4.3.

Return statements result in a *xfer* node being formed. Return program statements are of the form "return F(X_a)", and the resulting *xfer* node expression is of the form "x_{ret} = F(X_a);". The pseudo-variable, x_{ret}, has the same type as the method and allows the ordered return of values from a method to a specific calling site within an expression. The algorithm for creating the *xfer* node from a return statement is presented in Table 4.4.

Variable declaration statements result in *idef* nodes. Declaration statements are of the form "<dec-list> var;" and are processed as follows. The MakeG algorithm creates a new *idef* node, and constructs an incoming control dependence edge from the current control point to the new node. The node is decorated as public, private, final, static, etc per the <dec-list> list. If the declared variable is of an object type an entry is made in the alias set, A. L is updated to reflect the new node as the live DEF node for the current control point. If the variable is declared as final, L is annotated to show no further updates are allowed. If the variable is declared as static, then the alias list is updated to show this variable name is aliased for all instantiated objects of the same class. If the statement is of the form "<dec-list> var = value;", the assignment is contained within the *idef* node, otherwise an assignment to $\perp$ is placed in the *idef* node. These steps are detailed in Table 4.5.

Class and interface definitions result in *idef* nodes containing the definition. Class and interface definitions are of the form "dec-list c { π_c }" and the resulting *idef* nodes contain subgraphs representing the features being defined. The *idef* node is decorated to establish its accessibility and associate it with a package if applicable. A control edge is added to E_c in G from the current control point to

the new node. If the class extends another, a flow edge is added to E_f connecting the superclass to the new node. Similarly, if this class implements an interface, a flow edge connects the interface definition to the class. The contents of the node are created through a recursive MakeG call on π_c. Details of this process are provided in Table 4.6.

Method definitions also result in *idef* nodes containing the definition. Method definitions are of the form "dec-list m(X_f) { π_m }" and the resulting *idef* node contains a subgraph representing the method. The node is decorated per the "dec-list". To process a method, the MakeG algorithm makes a recursive call on π_m. Additional *def* and *xfer* nodes are created to accommodate parameter passing in and out of the method (including potential side effects). These steps are detailed in Table 4.7.

Assignment statements result in *assignment* nodes and are processed as follows. If the statement expression is of the form "o.x = <input>", the resulting node expression is also of that form. If the statement is of the form "o.x = F(X_a)", with input variable list $X_a = (x_1, x_2, x_3, \ldots, x_k)$, then the statement is a USE node for each $x_i \in X_a$. An *assignment* node having the node expression "o.x = F(X_a);" is created. Incoming flow edges are created for the object handle "o" and for each $x_i \in X_a$, with the number of flow edges and associated DEF nodes as determined by the liveness structure L. If the target of the assignment is an object, the alias set, A, is updated. L is then updated to associate this node as the only live DEF node for variable $o.x$ for the current control point. $F(X_a)$ may be, or contain, a method call. In this case, additional *xfer* and *def* nodes are created within the *assignment* node to allow parameter passing to and from the method. These steps are detailed in Tables 4.8 and 4.9.

If the assignment statement is of the form "o = new c()", a single flow edge is created to connect this node to the class definition node. If there is no live DEF node for the class definition, the *idef* node representing the class must be created via the getClass(c, Π) function and a recursive call to the MakeG algorithm. Additional *xfer* and *def* nodes are added to the *assignment* node as needed to support the constructor method. L is updated to reflect this node as the live DEF node for o plus each class and instance variable, $o.var$, assigned to in the constructor method. These steps are detailed in Table 4.10.

Finally, if the statement expression is of the form $o.m()$, then incoming flow edges connects the statement to the instantiation of each potential object o, and *xfer* and *def* nodes are added for parameter passing. L is updated to reflect side effects so that the appropriate *def* node remains the single live DEF node for the side effected variable in this control point. This case is detailed in Table 4.11.

Predicate statements result in *predicate* nodes plus additional graph components representing the True and False branches. Predicate statements take the form "if (P(X_a) { π_T } else { π_F }", where P(X_a) is a Boolean expression. To process a predicate statement, MakeG creates a new *predicate* node, a new control dependence edge from the current control point to the new node, and an incoming flow edge from each live DEF node for each $x_i \in X_a$. As always, new nodes may be instantiated within the *predicate* node to accommodate method calls within X_a. The True and False branches of the *predicate* node are incorporated into the graph through recursive calls to MakeG. These details are presented in Table 4.12 and 4.13.

Loop structures appear in J in the form "while (P(X_a) { π_l }" and result in multiple graph components representing the loop predicate and body. To support loops in the general case, the SOOPDG must define a predicate node controlling

loop iteration or exit, provide for the initialization of all values entering the loop, and create the loop body. The resulting graph must have an ability to iterate the loop zero or more times, and the ability to extract results upon loop exit. The process of incorporating a loop in the SOOPDG begins by creating a *predicate* node and associated incoming control and flow edges. A recursive call to MakeG on π_l creates the loop body, including appropriate control and flow edges to connect the loop body with the existing nodes in G. A new *while* node is created containing a copy of the loop body and predicate. During graph rewriting, the *while* node is expanded for execution, effectively unrolling the loop one time for each iteration. The *while* node contains an empty *def* node that acts as a placeholder for a copy of the loop body for the "i+1" iteration loop definition during graph rewriting. The algorithm creating the loop graph features from a "while" statement is presented in Table 4.14, 4.15, and 4.16.

4.1.4 Size of the SOOPDG Created by MakeG

The following proof demonstrates that the worse case size of the SOOPDG created by the MakeG algorithm is $O(n^2)$, where n represents the text size of the program after valve nodes have been accounted for. The literature for traditional PDGs typically presents graph size in terms of the number of statements in text programs [CFR91, Par92]. We use program text size as a basis to more easily discuss the impact of introducing *def* and *xfer* nodes associated with method calls. Each *def* and *xfer* node at a call site is a result of an additional term in the method call, and thus correlates more readily to program text size rather than number of statements. The addition of the valve nodes to the program text is

expected to increase the size of the program representation in a linear fashion, which we address in section 4.1.5.

The worse case SOOPDG size of $O(n^2)$ is not a surprising result, and is compatible with the $O(n^2)$ (in terms of program statement) worse case sizes of the SSA and SFU forms of PDGs. An expected size of $O(n)$ is presented for SSA [CFR91] and SFU (See Section 4.1.5) forms based on empirical evidence of normal programming practices. The following discussion shows that the source of the $O(n^2)$ size is due to *def* nodes at call sites that are introduced due to side effects within the called method. A program having $O(n)$ side effects at $O(n)$ call sites requires $O(n^2)$ *def* nodes. While good programming practices discourage the widespread use of side effects, we do not have the empirical evidence at this time to claim an expected SOOPDG size of $O(n)$.

Theorem 6 (Size of the SOOPDG) *The SOOPDG, $G = MakeG()$, has a worse case size of $O(n^2)$ nodes, where n represents the text size of a J program.*

Proof Theorem 6 is proved directly. Let n be the text size (number of characters) of a J program. We prove by cases:

1. Case 1: Statements are of type <output>. As each of these in the MakeG algorithm explicitly require the addition of one *output* node per statement processed, there is a linear correspondence of program length to nodes for these statement types.

2. Case 2: Statements are of type "return". As each of these in the MakeG algorithm explicitly require the addition of one *assignment* node per statement processed, there is a linear correspondence between program length and nodes for these statement types.

3. Case 3: Statements are of type variable *declaration*. As each of these in the MakeG algorithm explicitly require the addition of one *idef* node per statement processed, there is a linear correspondence between program length and nodes for these statement types.

4. Case 4: Statements of the type Class and Interface definition. As these statements result in a single *idef* node populated with subgraphs created from processing the program statements contained within the class, π_c, and these program statements can only contain statement types from the other cases, it follows that these statements also contribute nodes to G in a linear fashion with respect to their text size.

5. Case 5: Statements declaring *methods*. As method dependence graphs correspond directly to traditional PDGs, the graph created by each method is expected to be linear with respect to the size of the method [Par92]. Since a method cannot be longer than the program containing it, methods contribute nodes in a linear fashion to G.

6. Case 6: Statements are of type *assignment* through input. In the case of assignment via input exactly one node is added to G, and there is a linear correspondence between program length and nodes for these statement types.

7. Case 7 : This case represent assignment to a variable, with a potentialcall to a method resulting in side effects. In the case of assignment to a variable, exactly one *assignment* node is added to G, with the possible addition of *def* and *xfer* nodes due to method calls. The addition of the *xfer* nodes represents parameter passing to the called method, and are in a 1-to-1 relationship with parameters required by the method. In that respect they

are linear with respect to program text size at the calling site. The *def* nodes represent values returned from the method and are linear with respect to the number of DEF statements residing within the method whose effects are to visible variables that are external to the method. In the worse case, a single call may result in $O(n)$ *def* nodes, and $O(n)$ call sites would result in $O(n^2)$ nodes.

8. Case 8: In the case of instantiation of a new object, an assignment node, plus one node for each class variable initialized within the class constructor is added to G. The number of additional *def* nodes is linear with respect to the number of assignment nodes in the class constructor, and therefore linear with respect to text size of the class representation. In the worse case a program may contain $O(n)$ instantiations, resulting in $o(n^2)$ SOOPDG nodes.

9. Case 9: This case represents assignment via a call to a mutator method having tpye *void*. This case is a duplicate of Case 7, with the exception that no node is required to record the explicit variable assignment.

10. Case 10: Statements are of type *predicate*. The number of nodes contributed by predicate statements is clearly $1 + n_T + n_F$, where the first node is the predicate itself, and n_T and n_F are contributions by the True and False branches, respectively. We must consider nested structures and will do so by considering the number of nodes in n_T. Each non-predicate statement in n_T must belong to the remaining cases and thus contribute nodes either in a 1-to-1 fashion, or linearly with respect to the statements within the structure (See Cases 5, 6, and 7). Each predicate node in n_T contributes exactly one node, so n_T as a whole contributes nodes in a linear fashion with respect to the size of the True branch. The same reasoning applies

to n_F, and so the entire predicate structure is linear with respect to the number of program statements within it. Predicate statements are the only structure that may require valve nodes to be added to G. We will show in Section 4.1.5 that the number of valve nodes added to a graph G representing program P is expected to be linear with respect to the size of the original program P.

11. Case 11: Statement is of the type *while*. We consider the case of nested loops, and recognize that all program statements may be classified by the depth of the loop that most closely contains them. We further argue that, based on structured control dependence, each statement may reside at exactly one depth. Let k be the maximum depth in a nested loop structure. Consider the n_k statements at the k^{th} level. They have the property that none are *while* statements, or a contradiction would occur where a $k+1$ level would exist and the k^{th} level is not the deepest. The logic of MakeG for the loop case demonstrates that loop carried dependences require at most three representations within SOOPDG: one for the initial pass through the loop, one for the i^{th} iteration, and potentially a valve node retaining the SFU property at the i^{th} iteration. Since each statement in nested loops resides at exactly one nesting depth, each statement may be represented at most three times in G. Loop bodies are comprised of the statement types perviously discussed, thus the size of loop representation is of the same order as shown for previous statement types.

As all cases contribute nodes to G in either a linear or $O(n^2)$ fashion, with respect to the original program size augmented by statements corresponding to valve nodes, the sum of the contributions from all program statements is $O(n^2)$ in the worse case. ∎

The number of flow dependence edges contained within the SOOPDG is O(n^2) in the worse case, while the number of control depedence edges is $O(n)$ in both the worse and expected cases. Nodes may have multiple incoming and outgoing data flow dependence edges. Each DEF node may be associated with outgoing flow edges for exactly one program variable, which results in the worse case of $O(n)$ DEF nodes each flowing values to $O(n)$ USE nodes, resulting in $O(n^2)$ flow edges in the SOOPDG. Empirical evidence of program metrics is required demonstrating the number of live definitions for any program variable at any program point is not a function of program size. This evidence would allow a claim of $O(n)$ for the expected number of flow edges. We have not found such evidence to date. In the absence of unstructured control flow, each node has at most one incoming control dependence edge, and thus the number of control dependence edges is $O(n)$ in the expected and worse cases.

4.1.5 Upper and Lower Bounds of Valve Node Placement

This section presents a bound on the expected number of valve nodes required to attain SFU in typical programs. The definitions of graph structural forms are found in Chapter 2. Earlier literature introducing the valve node and developing Semantic PDGs establish the upper and lower bounds of valve nodes as $O(n^2)$ and zero, respectively [Par92]. The upper bound is determined by reasoning that for each DEF node, there is potentially one valve node required at each predicate node along the CDP for the DEF node. Since a given CDP may be of $O(n)$ length, each DEF node may generate $O(n)$ valve nodes. As there may be $O(n)$ DEF nodes, the number of valve nodes may be $O(n^2)$. The lower bound of zero

is determined by recognizing that programs may contain DEF nodes at all exit points of the CDP Subgraphs, thus requiring no insertion of valve nodes.

These bounds provide unrealistic extremes for the number of valve nodes required for a typical program. The lower bound is unreasonable as it requires that programmers create DEF nodes that are control-wise mutually exclusive. The upper bound of $O(n^2)$ requires that both the number of program variables and the depth of the deepest nested predicate structure grow linearly with program size. Empirical evidence [JPP94] indicates that increased program length does not tend to increase predicate nesting depths.

The following discussion provides a more realistic expectation for the number of required valve nodes. There is one sense in which we cannot make a determination regarding the necessity of inserting a valve node in a PDG. Valve nodes are generated as a result of existing DEF nodes within the PDG; however, if the DEF node itself always results in an identity assignment, then a valve node is not required. In general it is undecidable to determine whether a computation specified within a DEF node results in an identity assignment [AWZ88]. Thus these results hold within the limits of Turing computability.

Within this discussion we use V to represent the number of valve nodes required to achieve the SFU property within a PDG. Other nomenclature definitions useful to the discussion are grouped for convenience in Definition 31. We show that, though the number of Def-Order relations may be $O(n^2)$ with respect to the number of nodes within a program, V is O(n). This is a result of the application of Theorem 5 to the Control Dependence Subgraphs arising from the set of assignment nodes.

We conservatively assume that all statements (nodes) assigning to the same program variable are involved in at least one Def-Order dependence relationship,

thus requiring valve node insertion to obtain the SFU property. We do not explicitly discuss the existence or location of the USE node required for the Def-Order Dependence. The existence is presumed by the assumption that all assignment nodes are involved in a Def-Order relationship. Its location is presumed to allow the maximum number of valve nodes. More specifically, we presume that, given a set of DEF nodes, D, and single USE node u, $CDP(D) - CDP(u) = CDP(D)$.

Definition 31 (Program Parameters) *Let* Π *be a program written in language* J*, with variables renamed to eliminate output dependences. Let* G *be an SFU form SOOPDG representing program* Π*.*
n *= the number of nodes in* G*.*
k *= number of variables in* G*.*
α *= the number of assignment statements (nodes) in* P *(*G*).*
α_i *= the number of assignment statements (nodes) in* P *(*G*) assigning to the* i^{th} *variable,* $1 < i \leq k$*.*
V *= number of valve nodes to be added to* G *to achieve the SFU form.*
V^i *= the number of valve nodes to be added to* G *due to assignments to the* i^th *program variable,* $1 < i \leq k$*.*
In addition, let V^i_{mn} *represent the number of valve nodes contributed by assignment nodes (statements)* a^i_m *and* a^i_n*.*

The total number of assignment statements, α, can be written as the sum of statements assigning to each specific program variable.
$\alpha = \alpha_1 + \alpha_2 + \ldots + \alpha_k = \sum_{i=1}^{k} \alpha_i$
Also $\alpha < n$ due to the presence of at least one *Start* node in G. Similarly, since assignment statements must assign to the same program variable in order to be involved in a Def-Order dependence,
$V = V^1 + V^2 + V^3 + \ldots + V^k = \sum_{i=1}^{k} V^i$.

We now consider the set of assignment statements (nodes) in program P. It is convenient to consider the set of assignment nodes and partition them based on the variable being assigned to. This will allow us to consider the number of valve nodes contributed to the overall program on a per variable basis.

Definition 32 (Program Variable Set, A) *Let G be a PDG in the SFU form representing program P written in language J. We define A to be the set of all assignment nodes in G.*
$A = \{a_j \mid a_j$ is an assignment statement in $P\}$.
We can partition A according to the variable assigned to, such that
$A^i = \{a^i_j \mid a^i_j$ is an assignment statement to the i^{th} variable in $P\}$

Definition 33 (Def-Order Structural Forms) *Given a PDG, G, and assignment nodes for variable i involved in Def-Order relationships, we classify the structural form giving rise to the dependence as follows:*

1. *Figure 4.3 Case 1:* Single Breadth Def-Order
   (Shown as part of Multi Breadth Def-Order) This form involves two assignment nodes such that a^i_n is Def-Order dependent on a^i_m but $CP(a^i_m)$ is not a control ancestor of a^i_n.

2. *Figure 4.3 Case 1:* Multi Breadth Def-Order
   This form involves multiple assignment nodes, each having a pairwise Single Breadth Def-Order relation with at least one other node. Note that the number of Def-Order dependences arising from this structure is $O(n^2)$ for n assignment nodes.

3. *Figure 4.3 Case 2:* Single Depth Def-Order
   This form involves two assignment nodes such that a^i_n is Def-Order dependent on a^i_m and $CP(a^i_m)$ is a control ancestor of a^i_n.

4. *Figure 4.3 Case 3:* Multi Depth Def-Order

   This form involves multiple assignment nodes, each having a pairwise Single Depth Def-Order relation with at least one other node. Note that the number of Def-Order dependences arising from this structure is $O(n^2)$ for n assignment nodes.

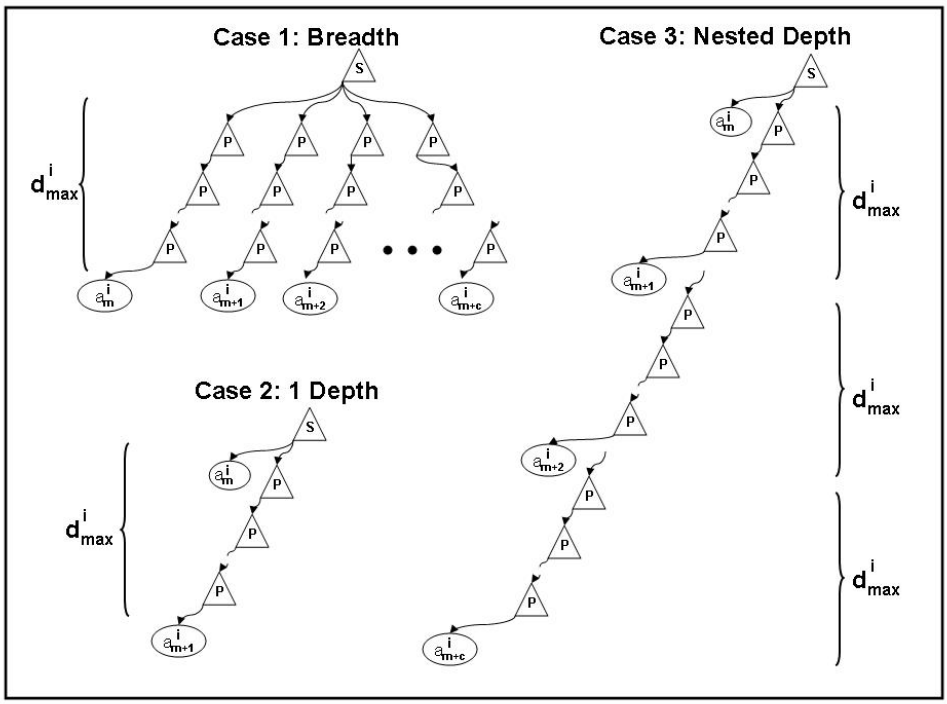


Figure 4.3: Graph Structure Resulting in Def-Order Dependences

We can determine the number of valve nodes required to resolve Def-Order dependences in each of the structural forms. For the i^{th} program variable, application of Theorem 5 requires that one valve node be placed at the exit point of the CDS defined by assignment node set A^i. The number of valve nodes placed is a function of the depth of the nested predicate structures creating the CDP for each of the assignment nodes in A^i.

Definition 34 (Def-Order Node Depth) *Given a PDG, G, and assignment nodes for variable i involved in Def-Order relationships, we define depth d^i_{mn} to be the nested predicate depth of node a^i_m with respect to a^i_n as follows:*

1. *Figure 4.3 Case 1:* Single Breadth Def-Order
   We define d^i_{mn} to be the length of $CDP(a^i_n)$ - $CCDP(a^i_m, a^i_n)$.

2. *Figure 4.3 Case 1:* Multi Breadth Def-Order
   For a set of assignment nodes, P-SET, assigning to variable i and involved in at least one Def-Order dependence relationship, we define d^i_{mn} for each $a^i_n \in P-SET$ to be the length of $CDP(a^i_n)$ - $CCDP(a^i_m, a^i_n)$, where $a^i_m \in P-SET$ is selected to minimize $CCDP(a^i_m, a^i_n)$.

3. *Figure 4.3 Case 2:* Single Depth Def-Order
   We define d^i_{mn} to be the length of $CDP(a^i_n)$ - $CDP(a^i_m)$.

4. *Figure 4.3 Case 3:* Multi Depth Def-Order
   For a set of assignment nodes, P-SET, assigning to variable i and involved in Multi Depth Def-Order dependence relationships, we define d^i_{mn} for each $a^i_n \in P-SET$ to be the length of $CDP(a^i_n)$ - $CDP(a^i_m)$, where $a^i_m \in P-SET$ is selected to maximize $CDP(a^i_m)$.

Application of Theorem 5 requires that a single valve node resides at each exit point of a CDS. The number of valve nodes contributed to the PDG by each assignment node is equal to some depth, d^i_{mn}. Since PDGs are finite, there is some finite upper bound on the value of all depths encountered in a program, which we shall call d_{max}. Similarly, we can define the average depth for a program to be d_{ave}. Thus a reasonable upper bound for number of valve nodes required in a typical program is:

$V = V^1 + V^2 + V^3 + \ldots + V^k = d_{max} * \alpha_1 + d_{max} * \alpha_2 + \ldots + d_{max} * \alpha_k = d_{max} * \sum_{i=1}^{k} \alpha_i < d_{max} * n$

The number of valve nodes is clearly $O(n)$. Johnson, et. al [JPP94] estimate $d_{max} = 13$ and $d_{ave} = 2.68$, so these provide good upper and expected estimates for the number of valve nodes required in a SFU form PDG.

4.1.6 Cost of the MakeG Algorithm

The cost of the MakeG algorithm is based on the size of the resulting SOOPDG and is $O(n^2)$ in the worse case. The addition of each primitive node to an SOOPDG requires constant cost, and Theorem 6 proves a worse case graph size of $O(n^2)$. As discussed in Section 4.1.4, analysis of program metrics may provide an expected SOOPDG size of $O(n)$, which would provide a basis to claim an expected cost of the MakeG algorithm of $O(n)$.

Theorem 7 (Cost of MakeG) *MakeG is worse case $O(n^2)$, where n represents the text size of the program acting as input to MakeG.*

Proof Theorem 7 is proved directly. Let $G = \{N, E_f, E_c\}$ be the graph created from program P by MakeG. The worse case size of the node set, N, is $O(n^2)$, per Theorem 6. The addition of each node to N is of constant cost for each node type in MakeG, and so the cost of the addition of all nodes is $O(n^2)$. The worse case size of E_f is $O(n^2)$, per Section 4.1.4. The addition of each flow edge is of constant cost, so the worse case cost of creating E_f is $O(n^2)$. The size of the control dependence edge set, E_c, is $O(n)$, per Section 4.1.4. Each edge is added at constant cost, and the cost of creating E_c is $O(n)$. Since the cost of creating

each component of G is at most $O(n^2)$, the cost of creating G using MakeG is $O(n^2)$. ∎

4.2 Program Evaluation

We present an informal rewriting semantics in this section, followed by a description of the ExecuteG algorithm that performs rewriting on an SOOPDG.

4.2.1 Graph Rewriting - Informal Semantics

Program execution may be enacted on the SOOPDG through graph rewriting rules. The SOOPDG rewriting semantics is a modified form of the semantics presented for the Semantic PDG [Par92]. In general, rewriting is a straightforward node-by-node process involving reduction of expressions within the individual nodes, and flowing the results along the directed edges. In the case of flow edges, values flow from DEF nodes to USE nodes to be stored in the USE node's local store. Note that, due to the SFU property, exactly one value will flow to a USE node for each variable regardless of the number of incoming flow edges. This allows nodes to mark incoming flow edges as bypassed or otherwise unnecessary based solely on local information. Control dependence edges notify dependent nodes either that control dependence criteria have been met for them, or that they have been bypassed. Rewriting potentially increases the size of the graph, as in the cases of loop iteration and instantiating methods at call sites prior to execution. In these cases, the rewriting semantics must specify how nodes containing the loop and method bodies are expanded to incorporate

the new nodes into the existing graph. A rewriting semantics must also specify initiation and termination criteria, and present a description of the effect on the graph of processing each node type.

Traditional PDGs contain a single *end* node signifying both program output and termination of the computation. Computation halts when the *end* node is executed. Output implies supplying some portion of the program store to an external entity. The presence of such a distinguished node allowed a lazy rewriting semantics to be developed [CF89, Par92], as demand initiates at a single, distinguished point in the graph and flows backwards. Computation and results flow forward to satisfy the demand. The SOOPDG replaces the *end* node with the *xfer* node supplying values from a local store from one context to another, and the *output* node supplying values of the program store to an external entity. Neither of these node types are associated with termination criteria within the SOOPDG. Since we allow an arbitrary number of these nodes, they do not form a natural feature to initiate execution via a lazy semantics (though this is a topic reserved for future work, see Chapter 8). Thus, initiation and termination criteria for the SOOPDG differ from the traditional PDG.

SOOPDG rewriting initiates at the unique program *Start* node, and terminates when all nodes for which this node is a control ancestor are marked as executed or bypassed. The rewriting semantics maintains a worklist, W, of nodes ready for execution, and randomly selects from among them for execution. The worklist is initiated with the *Start* node, which has no incoming control or flow edges (the predicate expression is a constant, "True") and is thus ready for execution. Other nodes are added to W as they have satisfied control dependence criteria and received values for all variables in their expressions.

At the node level, rewriting consists of one of two actions. Primitive nodes are rewritten by resolving the arithmetic/logical (A/L) expression they contain and having the results flow to dependent nodes in a manner similar to the rewriting semantics of Parsons [Par92]. Rather than remove executed and bypassed nodes and edges from the graph, we retain and tag them as the computation progresses, leaving the tagged graph as an intact record of the process. This proves valuable when discussing how the SFU property is maintained during program execution. The only modification from traditional rewriting semantics for graph edges is that we allow flow edges to be used multiple times in the case of flowing method subgraphs to call sites. If the node is not primitive, it is expanded and the internal nodes are incorporated into G. This occurs when a method is incorporated at a call site during graph rewriting. The node containing the call site is expanded so that the *xfer* and *def* nodes acting as the interface between the call site and the method can be connected to the method body and the entire set connected to the existing graph. A node containing multiple method calls may be expanded multiple times as individual methods are instantiated at the call sites. Examples of instantiating a method body at a call site are given in Figure 4.4. Expansion also occurs during loop unrolling. An example of this is given in Figure 4.5.

The concept of object aliasing requires a slight modification in the mechanics of flowing values in the SOOPDG compared to traditional PDGs. A reference to an instance variable "o.x" actually requires two values to flow to it to fully identify the variable in question. The first value is the object handle (value of "o"), and this handle dictates which incoming flow edges are eligible to flow a value for the specific instance variable "x." During rewriting, each object name, "o", is assigned a value at the time of object instantiation, but the object name may receive a new value via object aliasing "$o = o_1$". The value for the object name is passed via flow edges to future uses of the object name. The result of

this is that a DEF node assigning a value to "o.x" will not flow its value to a USE of "o.x" unless the values for "o" match in both nodes at the time of the attempted data flow.

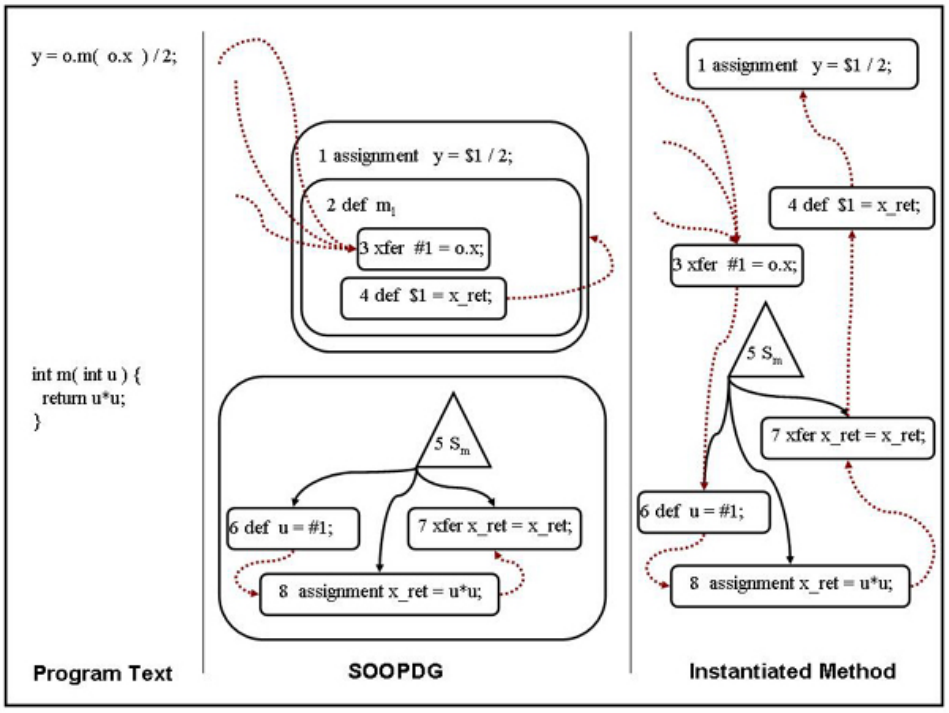


Figure 4.4: Example of Instantiation of Method at Call Sites

Output nodes are considered executed once control dependence criteria has been satisfied and a value for the output variable, "x", has been received. *Output* nodes have no outgoing edges and so do not further affect the execution of the program. These are the only nodes with no outgoing edges that execute.

Primitive *xfer* and *def* nodes are similar to *output* nodes in that they are considered executed as soon as their internal A/L expressions are resolved. These nodes then flow results along outgoing flow edges. If the nodes have no outgoing

edges, or all outgoing edges are marked as bypassed, the node will not contribute to the program and thus will not execute.

Primitive *idef* nodes specifying class and method definition are not executed per se, but are used to supply methods to calling sites. These *idef* nodes may repeatedly supply the same method to multiple calling sites within the program. *Idef* nodes declaring variables may require reduction of some expression as an initial value is assigned to the variable. In this case, the reduction occurs only upon satisfaction of node flow and control dependence criteria. Upon execution, the *idef* node flows the result along outgoing flow edges.

Primitive *assignment* nodes are rewritten in a manner similar to *idef* nodes in that the node expression is reduced upon satisfaction of control dependence criteria and receipt of variable values. The resulting value flows along outgoing flow edges to target nodes. If a target node has already received a value for this variable, the outgoing edge should be marked as such prior to the execution of the *assignment* node. In this case, no value is transferred along the edge. An example of *assignment* node rewriting and data flow is given in Figure 4.5.

Primitive *Predicate* nodes are resolved in a manner similar to *assignment* nodes in that the A/L expression is reduced upon satisfaction of control and flow dependence criteria. Upon reduction to True or False, the results flow along the outgoing control dependence edges, notifying target nodes as to whether they are bypassed or eligible for execution. This process is recursive in that, if a bypassed target is also a predicate, it will propagate the bypassed result to nodes control dependent on it along both the True and False branches. An example of predicate node rewriting is given in Figure 4.6.

While nodes are never primitive by definition, and the subgraph representing the loop body must be expanded and incorporated in G before rewriting of the

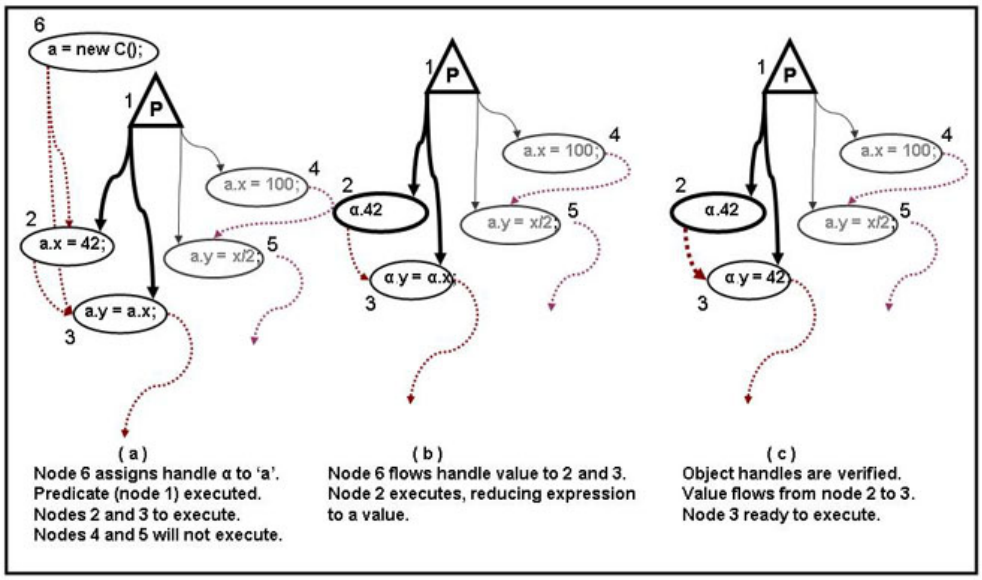


Figure 4.5: Assignment and Data Flow Example

primitive nodes within loop body can be executed. The expansion corresponds to the unrolling of a single iteration of the loop body and it occurs prior to rewriting of any nodes within the loop body. A copy of the loop body is placed in the placeholder *def* node, including a copy of this placeholder node. The *while* node is then expanded to allow the loop body to become incorporated in G. A new control edge is added from the *while* node's control parent to the loop predicate. An example of a *while* node loop expansion is provided in Figure 4.7.

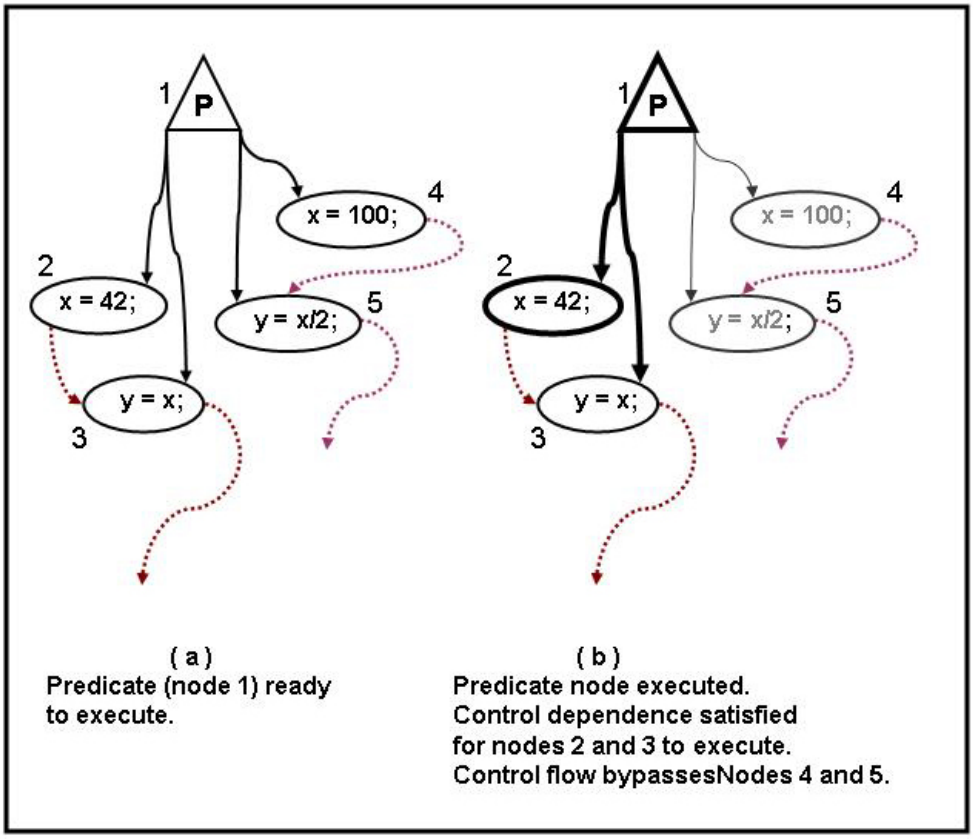


Figure 4.6: SOOPDG Predicate Node Rewriting Example

4.2.2 Graph Rewriting - The ExecuteG Algorithm

The ExecuteG algorithm maintains three sets of nodes. The first is the *executable* set, N_E, consisting of nodes that have met control and data dependence criteria and are therefore available for execution. The remaining sets identify nodes that have met control dependence criteria only, N_C, and those that have met data dependence only, N_D. The rewriting rules presented in this section are top down, but not sequential. The ExecuteG algorithm arbitrarily selects a node from N_E for execution. Results are propagated along dependence edges, and affected nodes are potentially promoted into N_E, N_C, or N_D. Executed nodes are removed from N_E, and nodes promoted into N_E are removed from N_C and N_D. Program execution is terminated when no more nodes are available in the executable set, or can be added to the executable set through the propagation of results from executed nodes.

The ExecuteG algorithm relies upon a number of utility functions to manipulate the node sets. The functions are defined below, but details are not presented. These utility functions are as follows:

1. *isPrimitive(nid)* – The isPrimitive function returns True if node nid is primitive and False otherwise.

2. *dataQuery(nid)* – The dataQuery function requires a node id as input and returns True if data dependence criteria has been met for node n, and False otherwise.

3. *controlQuery(nid)* – The controlQuery function requires a node id as input and returns True if control dependence criteria has been met for a node and False otherwise.

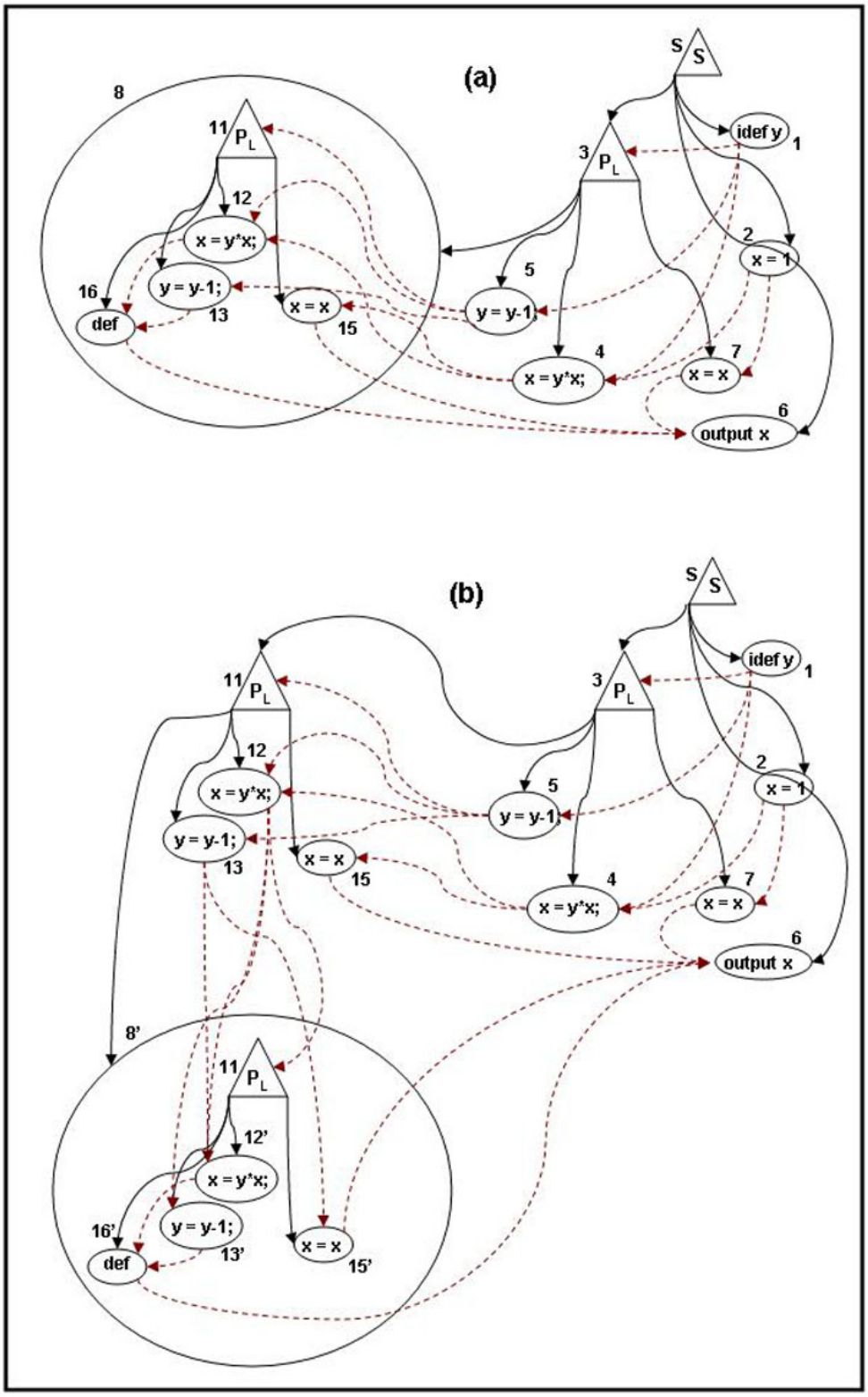


Figure 4.7: Example of While Loop Node Expansion

4. *updateD(N_D, nid)* – The updateD function adds node nid to N_D if dataQuery(n) returns True.

5. *updateC(N_C, n)* – The updateC function adds node nid to N_C if controlQuery(n) returns True.

6. *updateE(N_E, n)* – The updateE function add executable nodes to N_E, while removing them from N_C and N_D. This is performed in four steps:

   (a) $N' = N_D \cap N_C$

   (b) $N_E = N_E \cup N'$

   (c) $N_D = N_D - N'$

   (d) $N_C = N_C - N'$.

7. *select(N)* – The select function arbitrarily selects a node, n $\in$ N, where N is any node set.

8. *removeNode(n, N)* – The remove function removes node n from any set N, N = N - n.

9. *resolveExpression(nexpr, nstore)* – The resolveExpression function receives a primitive node's A/L expression and local store as input and resolves the expression to a single value.

10. *flowData(nid)* – The flowData function transmits variable values along outgoing flow edges from node nid to all target nodes. Object handle values always flow, while instance variable values only flow if object handles match in nid and the target node.

11. *flowControl(nid)* – The flowControl function flows control information from predicate node nid to the targets of all outgoing control dependence edges.

12. *getValue(nid, o.x)* – The getValue function requests values for variable "o.x" backwards through incoming flow edges.

13. *getMethod(nid, o.m)* – The getMethod function initiates a request backwards through incoming flow edges for method "o.m" to flow to node nid for instantiation as part of nid's expansion.

The ExecuteG algorithm is given in Table 4.1.

Table 4.1: ExecuteG Algorithm

```
Let worklist N_E = { (Start) };
While ( W not empty ) {
   n = select(N_E);
   if (isPrimitive(n)) {
      if (ntype is Output) {
         resolveExpression(nexpr, nstore); }
      if (ntype is xfer, idef, def or assignment) {
         resolveExpression(nexpr, nstore);
         flowData(nid);
         for (each target node, n) {
            updateD(N_D, n); }
      if (ntype is predicate) {
         resolveExpression(nexpr, nstore);
         flowControl(nid);
         for (each target node, n) {
            updateC(N_C, n); }

      updateE(N_E, N_C, N_D);
      removeNode(nid, N_E);
     } else {
      expandNode(nid);
   }
}
```

Table 4.2: MakeG Algorithm

Let $\Pi = \pi_{main} c_1 c_2 c_3 \ldots c_k$,

Let $W = getMethod(c.main, \Pi)$,

Let $G = \{N, E_c, E_f\} = \{\{Start\}, \phi, \phi\}$,

Let Control point, $C = (Start, True)$,

Let Alias set, $A = \phi$; Let Liveness set, $L = \phi$;

Let Definitions list, $\Delta = \phi$; Let packid = "*";

G = MakeG(Π, W, G, C, A, L, Δ, packid);

MakeG(Program Π, Worklist W, SOOPDG G, Control Point C, Liveness Set L, Definitions list Δ, package packid)

while (W !empty) {

s = getNextStatement(W);

Switch Typeof(s)

Case 1. <output> = x; – See Table 4.3

Case 2. Return F(X_a); – See Table 4.4

Case 3. Dec-list x = val; – See Table 4.5

Case 4. Dec-list class c { π_c }; – See Table 4.6

Case 5. Dec-list m(X_f) { π_m } – See Table 4.7

Case 6. x = <input>; – See Table 4.8

Case 7. x = F(X_a): – See Table 4.9

Case 8. o = new C(X_a); – See Table 4.10

Case 9. o.m(X_a); – See Table 4.11

Case 10. if P(X_a) { π_T } else { π_F } – See Table 4.12 and 4.13

Case 11. while P(X_a) { π_l} – See Table 4.14, 4.15, and 4.16

W = W - s; }

Return G; }

Table 4.3: Case 1: Output Statements

s is of the form "<output> = x;"

nid = makeNewNodeID();

ntype = "out";

nexp = "out x";

nstore = ϕ;

nstatus = "NULL";

ndecor = ϕ;

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

N = N $\cup$ { n };

$E_c = E_c \cup$ {(C.pid, nid, C.b)};

for (each DEF node, $d \in l_C^x$) {

    $E_f = E_f \cup$ {(d, nid)};

}

Table 4.4: Case 2: Return Statements

s is of the form "Return F(X_a);"
nid = makeNewNodeID();
ntype = "assignment";
nstore = ϕ;
nstatus = "NULL";
ndecor = ϕ;
nexp = "x_{ret} = F(X_a)";
g = processExpression(X_a, nid, L, G);
nexpr = nexpr:g.N;
n = {nid:ntype:nexp:nstore:nstatus:ndecor};
N = N $\cup$ { n };
E_c = E_c $\cup$ {(C.pid, nid, C.b)};
E_f = E_f $\cup$ g.E_f;
update L such that $l_C^{x_{ret}}$ = { nid };

Table 4.5: Case 3: Declaration Statements - Variable

s is of the form "<dec-list> x = val;"

nid = makeNewNodeID();

ntype = "idef";

nexp = "x = val;";

nstore = ϕ;

nstatus = "NULL";

ndecor = {packid:dec-list};

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

$N = N \cup$ { n };

$E_c = E_c \cup \{(C.pid, nid, C.b)\}$;

if (dec-list contains the keyword "final") {

    $l_C^x = \{nid : final\}$;

} else {

    $l_C^x = \{nid\}$ in L

}

if (dec-list contains the keyword "static") {

    $A = A \cup$ { c.x: *.x }

}

Table 4.6: Case 4: Class Definition

s is of the form "<dec-list> class c { π_c }",

nid = makeNewNodeID()

ntype = "idef";

nstore = ϕ;

nstatus = "NULL";

ndecor = {packid:dec-list};

$W' = \pi_c$;

$G' = N = E_c = E_f$ =L$' = \phi$;

$C' = (NULL, NULL)$;

$L' = \phi$;

nexp = MakeG(Π, W', G', C', L', Δ, packid);

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

$N = N \cup \{n\}$;

$E_c = E_c \cup \{(C.pid, nid, C.b)\}$;

if (c extends class c') {

For each $d \in l_C^{c'}$ {

$E_f = E_f \cup \{(d, nid)\}$;

}

}

$L = L \cup l_C^c$:{nid};

Table 4.7: Case 5: Method Definition

s is of the form "<dec-list> $m(X_f)\{\pi_m\}$ "

nid = makeNewNodeID()

ntype = "idef";

nstore = ϕ;

nstatus = "NULL";

ndecor = {packid:dec-list};

$W' = \pi_m$;

$C' = (NULL, NULL)$;

$L' = \phi$;

$N' = \{(Start_m, True)\}$;

$E'_f = E'_c = \phi$;

for (each $x \in X_a$ AND each var, x in π_m, having a USE with no DEF) {

    defNode = makeNewNodeID();

    $N' = N' \cup \{(defNode, "def", "x", \phi, NULL, packid)\}$;

    $E_c = E_c \cup \{(Start, defNode, True)\}$;

    $Addl_C^x = \{(idefNode)\} toL'$;

}

G' = { N', E'_f, E'_c }

G' = MakeG(Π, W', G', C', A, L', Δ, packid);

xferNode = makeNewNodeID():

$N' = N' \cup \{(xferNode, "xfer", "x''_{ret}, \phi, NULL, packid)\}$;

for each $d \in l_C^{x_{ret}}$ {

    $E_f = E_f \cup \{(d, xferNode)\}$; }

nexp = G';

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

$N = N \cup \{n\}$;

Add l_C^m = { (*nid*) } to L;

Table 4.8: Case 6: Assignment - Program Input

s is of the form "x = <input>;"

nid = makeNewNodeID()

ntype = "assignment";

nstore = ϕ;

nstatus = "NULL";

ndecor = {packid:dec-list};

nexp = "x = <input>;";

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

Update L such that $l_C^x = \{nid\}$.

$E_c = E_c \cup \{(C.pid, nid, C.b)\}$

Table 4.9: Case 7: Assignment

s is of the form "y = F(X_a);"

nid = makeNewNodeID()

ntype = "assignment";

nstore = ϕ;

nstatus = "NULL";

ndecor = ϕ;

nexp = "y = F(X_a)";

g = processExpression(X_a, nid, L, G);

nexpr = nexpr:g.N;

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

$E_c = E_c \cup \{$(C.pid, nid, C.b)$\}$

$E_f = E_f \cup g.E_f$;

Update A.

Update L such that $l_C^y = \{nid\}$.

Table 4.10: Case 8: Assignment Through Object Instantiation

s is of the form "o = new c(X_a);"

nid = makeNewNodeID()

ntype = "assignment";

nstore = ϕ;

nstatus = "NULL";

ndecor = ϕ;

nexp = "o = new C(X_a);";

for each var $o.x$ assigned to in constructor {

    defNode = makeNewNodeID();

    nexp = nexp:"(idefNode, "def", "x", ϕ, NULL, packid)";

    Add l_C^x = { (idefNode) } to L';

}

g = processExpression(X_a, nid, L, G);

nexpr = nexpr:g.N;

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

$E_c = E_c \cup \{$(C.pid, nid, C.b)$\}$

$E_f = E_f \cup g.E_f$;

Update $A = A \cup O_r : o$;

Update L such that $l_C^y = \{nid\}$.

Table 4.11: Case 9: Assignment Through Side Effect

s is of the form $o.m(X_a)$;
nid = makeNewNodeID();
nstore = ϕ;
nstatus = "NULL";
ndecor = {packid};
nexpr = "$c.setVar(X_a)$;"
for each " end x" node in m {
g = processExpression(X_a, nid, L, G);
nexpr = nexpr:g.N;
n = {nid:ntype:nexp:nstore:nstatus:ndecor};
$N = N \cup \{n\}$;
$E_f = E_f \cup g.E_f$;
$E_c = E_c \cup \{(C.pid, nid, C.b)\}$

Table 4.12: Case 10: Predicate (1 of 2)

s is of the form "if P(X_a) { π_T } else { π_F }"

nid = makeNewNodeID();

ntype = "predicate";

nstore = ϕ;

nstatus = "NULL";

ndecor = ϕ;

nexp = "P(X_a";

g = processExpression(X_a, nid, L, G);

nexpr = nexpr:g.N;

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

$N = N \cup \{n\}$; // Add predicate node to N.

$E_f = E_f \cup g.E_f$;

$E_c = E_c \cup \{(C.pid, nid, C.b)\}$;

$G_T = \{N_T, E_{fT}, E_{cT}\} = \{\phi, \phi, \phi\}$;

C_T = (nid, True);

$A_T = A$;

$L_T = L$;

G_T = MakeG($\Pi, \pi_T, G_T, C_T, A_T, L_T, \Delta, packid$);

$G_F = \{N_F, E_{fF}, E_{cF}\} = \{\phi, \phi, \phi\}$;

C_F = (nid, True);

$A_F = A$;

$L_F = L$

G_F = MakeG($\Pi, \pi_F, G_F, C_F, A_F, L_F, \Delta, packid$);

$G = G \cup \{ G_T \cup G_F \}$;

(Continued in following figure)

Table 4.13: Case 10: Predicate (2 of 2)

Let V_T be the set of variables receiving an assignment within control point C_T and all child control points.

Let V_F be the set of variables receiving an assignment within control point C_F and all child control points.

for (each variable, $x \in \{V_T - V_F\}$ {

nid = makeNewNodeID();

ntype = “assignment”;

nexp = “x = x;”

nstore = ϕ;

nstatus = “NULL”;

ndecor = {};

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

N = N ∪ {n};

$E_c = E_c \cup$ (C.pid, nid, C.True);

for (each DEF node $d \in l_C^{v_T}$) {

$E_f = E_f \cup$ {d, nid)}; }

}

for (each variable, $v_F \in \{V_F - V_T\}$ {

ntype = “assignment”;

nexp = “x = x;”

nstore = ϕ;

nstatus = “NULL”;

ndecor = {};

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

N = N ∪ {n};

$E_c = E_c \cup$ (C.id, nid, C.False);

for each DEF node $d \in l_C^{v_F}$ {

$E_f = E_f \cup$ {d, nid)}; }

}

$L = L_T \cup L_F$

Table 4.14: Case 11: While Loop (1 of 3)

s is of the form "while $P(X_a)$ { π_l }"

// Make predicate node...

nid = makeNewNodeID();

ntype = "predicate";

nstore = ϕ;

nstatus = "NULL";

nexpr = "$P(X_a)$";

g = processExpression(X_a, nid, L, G);

nexpr = nexpr:g.N;

n = {nid:ntype:nexp:nstore:nstatus:ndecor};

$E_f = E_f \cup g.E_f$;

$E_c = E_c \cup$ { (C.pid, nid, C.b) };

// Create subgraph representing loop body...

W' = π_l;

N' = { nid };

G' = $E_c = E_f = \phi$;

C' = (nid, True); $A' = A$; L' = L;

G' = MakeG(Π, W', G', C', A', L', Δ, packid);

// Add valve nodes to G' for the (pid, False) control point (see Predicate, Table 4.12).

// Incorporate G' into graph G...

N = N $\cup$ N';

$E_f = E_f \cup E'_f$;

$E_c = E_c \cup E'_c$;

Table 4.15: Case 11: While Loop (2 of 3)

```
// Build while node with copy of loop body... forall (nodes n ∈ N′) {
    newnid = makeNewNodeID(); }
    replaceAllOccurrences(n, newnid, G');
}

for (all variables, x, assigned to in G') {
    newxfer = makeNewNodeID();
    N' = N' ∪ {(newxfer, "xfer", "xfer x", φ, NULL, φ)};
    for (all DEF nodes d ∈ l_C^x) {
        E′_f = E′_f ∪ { (d, newxfer) };
    }
}
```

Table 4.16: Case 11: While Loop (3 of 3)

// Add empty def node for i+1 iteration...

defid = makeNewNodeID();

ntype = “def”;

nstore = ϕ;

nstatus = “NULL”;

nexpr = “NULL”;

ndecor = ϕ

N’ = N’ $\cup$ { (defid:ntype:nexpr:nstore:status:ndecor) };

}

whileid = makeNewNodeID();

ntype = “while”;

nexpr = G’;

nstore = ϕ;

nstatus = “NULL”;

ndecor = ϕ

N = N $\cup$ { (whileid:ntype:nexpr:nstore:nstatus:ndecor) };

5

PROGRAM ANALYSIS

In this chapter we discuss several common program analysis techniques carried out using dependence graphs. The specific analyses we discuss are program slicing, constant propagation, call chain construction, class inheritance, and archive optimization. We first discuss program slicing, which is defined with respect to a specific program point, n_0. A backwards program slice identifies what previous program statements may affect the computation at n_0, while a forward slice identifies those statements that may be affected by the result at n_0. Constant propagation is a processing technique allowing the effects of constants (or the values of variables that are determinate prior to program execution) to be incorporated into a program prior to program execution. Call chain analysis attempts to identify sequences of method calls such that a site in method m_1 calls m_2, and a site in m_2 calls m_3, etc. Results tend to be conservatively correct and may contain chains that are *infeasible* at run time. Inheritance analysis defines inheritance in the classic sense whereby a class, c_2 that extends c_1 implicitly contains the class attributes and methods of c_1 unless explicitly stated otherwise. Since we are focusing on a Java-like language, we consider inheritance from a single parent only. Archive optimization attempts to identify, through static analysis, which specific attributes and methods in a given class are required for program execution. The goal is to reduce the size of archives or production bytecode by including only those features potentially contributing to the program result.

5.1 Program Slicing

A program slice on a dependence graph consists of the nodes that potentially affect the computation at a program point of interest [Tip95, Zha99]. A slicing procedure operating on a graph extracts the nodes related, directly or indirectly, to a specific computation in the original program. Slices may be *backward*, containing program statements affecting the program point of interest, or *forward*, containing program points affected by the point of interest. The problem of creating a program slice is essentially a graph reachability problem [Tip95]. The backwards slice is created by traversing dependence edges backwards through the dependence graph. The forward slice is created by traversing dependence edges forward from the point of interest. Slices were originally introduced for debugging purposes, but have since been utilized in a wide range of applications such as parallelization, program differencing, program testing, complexity measurement, and reverse engineering [Tip95, Zha99].

Horwitz et al. [HRB90] presents an interprocedural slicing algorithm that serves as the basis for many OO slicing algorithms [Zha99, WRW03, LH98, CX01, AH03]. The technique resolves the *context calling* problem through the use of summary edges. Roughly stated, the context calling problem states that a naive backwards traversal of graph edges encountering a single method call would incorrectly visit all call sites (contexts) for that method, and include them in the slice. The technique resolves this problem through the presence of summary edges at method call sites that connect parameter input/output nodes. Assuming a backwards slice, the summary edges specify which input nodes are to be included in the slice for each specific output node. Depending on the specific representation and algorithm being discussed, these summary edges are either created during creation of the graph, or during a pre-processing analysis performed prior to ob-

taining the slice. With the summary edges in place, the general technique is to perform the slice in two phases. The first phase traces edges backwards from the point of interest, marking all ancestors of the node in question. During this phase, methods are not entered, but are bypassed using summary edges. During the second phase of the algorithm, no new methods are visited, but those previously tagged and bypassed are entered and explored, so that nodes internal to the methods may be included in the slice.

The structure of the SOOPDG captures sufficient information to perform slicing in a single pass with no preprocessing of the graph. At each call site, all *xfer* and *def* nodes associated with parameter input/output are contained within a single *def* node identifying the method. In addition, the subgraph representing the method has no incoming/outgoing edges in the static graph, as these are constructed during rewriting. The absence of these edges removes the danger of confounding the source of incoming edges to the method subgraph from multiple call sites. Assuming a backwards slice, if a *def* node representing output from a method is encountered, the method subgraph may be entered immediately through its corresponding *xfer* output node. The method subgraph is traversed, appropriate nodes added to the slice, and the affected input parameter *def* nodes are identified. The slice continues at the call site with the local *xfer* nodes corresponding to the *def* nodes identified in the method.

To eliminate duplication of effort, the slicing algorithm maintains a record of the results of entering each method by an individual *xfer* node. We refer to this record as a *template* as it is intended to provide a pattern to be used repeatedly to quickly identify affected nodes at a call site. A template is identified by a method and entry node, and identifies two sets of nodes: the exit nodes included in the slice due to inclusion of the entry node, and all other nodes in the method

subgraph included in the slice due to inclusion of the entry node. We refer to the template for method m and entry node n as T_m^n, as shown in Definition 35. Reference to T with no subscripts represents the set of all templates. Templates serve a portion of the purpose of the summary edges of Horwitz et al. in that they rapidly identify parameter input nodes at a call site once a parameter output node is identified. Construction of the templates is more efficient than summary edge construction as summary edges are produced for all methods and all method parameters *a priori*, while templates are produced only for method parameters encountered during a specific slice.

Definition 35 (Template) *For entry node n at method m, template $T_m^n = \{d_1, d_2, \ldots, d_j\}\{n_1, n_2, \ldots, n_k\}$, identifies the nodes in the method subgraph to be added to a slice due to inclusion of node n; $\{d_1, d_2, \ldots, d_j\}$ represents the set of* def *nodes associated with input parameters for the method, and $\{n_1, n_2, \ldots, n_k\}$ represents any other nodes within the method subgraph.*

We present a backward slicing algorithm, BackwardSlice, in Table 5.1. For a backward slice, given a graph, $G = (N, E_c, E_f)$ and point of interest n_0, we trace dependence edges backwards from n_0 and mark visited nodes. If a visited node is not a *def* node, it cannot be an entry node to a method, and is treated normally. If a visited node is a *def* node with an empty *nexpr*, then the node is a placeholder for a *while* subgraph, and is treated normally. If the visited node is a *def* node associated with a value returned from a method, the algorithm explores the method and uses templates as discussed above. We do not present an algorithm for a forward slice, as it is almost identical to the BackwardSlice algorithm, with edges traversed in a forward direction.

The BackwardSlice algorithm visits each edge at most one time. As discussed in Section 4.1.4, the expected number of edges is of the same order of magnitude

as the number of nodes in the graph, thus the slicing algorithm is O(n) in the expected case, and O(n^2) in the worse case.

5.2 Constant Folding and Constant Propagation

Constant Folding and Constant Propagation are compiler optimization techniques. Constant Folding refers to the reduction of subexpressions containing only constant values [ALS07]. Constant Propagation refers to the replacement of variables having determinate constant values with the value [ALS07]. Clearly, these processes may be iterated at one point during compilation or repeated at several points in a single compilation in order to take full advantage of determinate values prior to program execution. In terms of the SOOPDG, this process occurs through an application of the rewriting semantics.

As discussed in Chapter 2, the SOOPDG incorporates the SFU form of the PDG at its core, while other dependence based forms incorporate the SSA form. When performing Constant Folding and Constant Propagation, the SFU form exhibits an advantage in analytical capability over the SSA form, as shown in Figures 5.1 and 5.2 using the code fragment listed. The sample program is composed only of constant assignments, and thus may be fully resolved *a priori*. Constant propagation for the SSA form is dependent upon the outcome of the predicate, while propagation for the SFU form may continue to completion independently of the outcome of the predicate node.

Table 5.1: Backwards Slicing Algorithm

BackwardSlice(graph G, node n);

Given SOOPDG $G = \{N, E_c, E_f\}$ and node n

S = program slice for n_0 =$\{n_0\}$

W = worklist = $\{n_0\}$

while (W is not empty) {

W' = temporary worklist = ϕ;

for (each node $n_i \in W$) {

if (*ntype* is not *def* $\vee$ *nexpr* is empty) {

$W_i = \{ n_j \mid ((n_j, n_i) \in E_f \vee (n_j, n_i, \text{b}) \in E_c) \wedge n_j \notin S; \}$

} else (n_i is a def node for method m at a call site) {

n_x = the xfer node in method m associated with n_m;

if ($T_m^{n_x} \notin T$) {

G_m = subgraph for method m;

$T_m^{n_x}$ = BackwardsSlice(G_m, n_x);

$T = T \cup T_m^{n_x}$;

}

$T_m^{n_i} = \{d_1, d_2, \ldots, d_j\}, \{n_1, n_2, \ldots, n_k\}$;

$W_i = W_i \cup \{d_1, d_2, \ldots, d_j\}$;

$S = S \cup \{n_1, n_2, \ldots, n_k\}$;

}

$W' = W' \cup W_i$;

}

$S = S \cup W$;

$W = W'$;

}

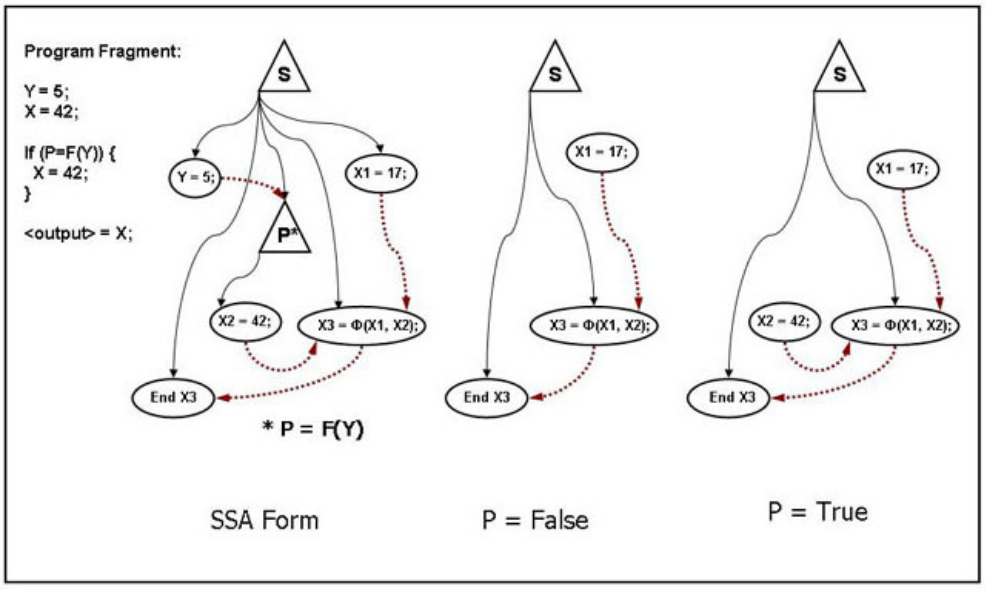


Figure 5.1: SSA Form and Constant Propagation

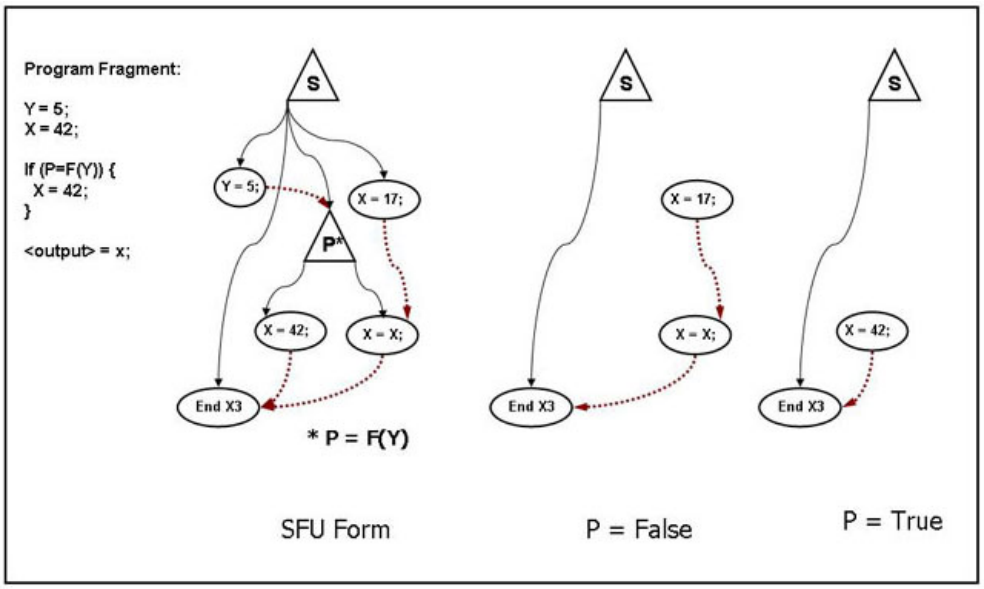


Figure 5.2: SFU Form and Constant Propagation

5.3 Call Chain Analysis

Call chains are sequences of method calls within a program. A *call chain tree* is typically used to represent calling relationships, where nodes represent methods (or call sites within methods) and the directed edges represent method calls. For example, (m_1, m_2) indicates that method m_1 calls method m_2. Call chains are *feasible* if they can be traversed by an instance of program execution, and *infeasible* if there is no possible data set that will allow a program to traverse the chain. Call chains are used for software understanding by allowing graphical representation of calling relationships among methods [RKG04]. Call chains are also used in software testing by requiring that data sets be developed exercising all feasible call chains [RKG04].

The context problem arises again in that call chains may not show how the called method relates to the calling method(s). Call chains may be *context sensitive* or *context insensitive* [GDD97]. Context insensitive call chains contain a single node for each method, regardless of the number of actual calls or calling locations. Context sensitive call chains provide more information by representing individual methods with multiple nodes, with each node providing more specific information regarding the calling context.

We define call trees with respect to specific methods such that the method resides at the root of the call tree, denoted as CT(m). This allows us to describe call trees emanating from any method within a program, and denote the call tree for an entire program as the call tree emanating from the program's *main* method, denoted CT(main). A call tree contains two components: the root method id, and nested call trees emanating from the methods called by the root method.

Call trees are formally defined in Definition 36. A *call chain* is a single path within a Call Tree, and a *call forest* is a set of call trees.

Definition 36 (Call Tree, CT(m)) *A Call Tree for method m, denoted CT(m), is the call tree formed with method m as its root, having the following form:*
CT(m) = (m, NULL) if method m calls no other methods, and
CT(m) = (m, (CT(m_1), CT(m_2), ..., CT(m_k))) if m calls $k > 0$ methods.

As an example, assume method m_1 calls m_2, which in turn calls m_3. The call trees developed by this call chain are given as follows:
CT(m_3) = (m_3, NULL) ,
CT(m_2) = (m_2, CT(m_3)) = (m_2, ((m_3, NULL))), and
CT(m_1) = (m_1, CT(m_2)) = (m_1, (m_2, ((m_3, NULL)))).
The nested format of the call trees allows for extraction of any specific call chain, and provides the context of any specific call in a manner similar to variable scoping rules. We distinguish between the actual call tree, CT(m), and a reference to the call tree, CT_m, and use this distinction to allow termination of the MakeCT algorithm in the event of cycles in call chains.

We present a recursive algorithm, MakeCT, in Table 5.2, that creates a context sensitive Call Chain Tree from static analysis of an SOOPDG. We presume type analysis has resolved variable names to classes prior to the initiation of the algorithm; in cases involving polymorphic methods the algorithm creates the set of all possible call chains. The MakeCT algorithm makes use of a call tree forest, F, to identify call trees that have already been calculated. The first encounter with a specific method results in a recursive call to MakeCT on the method. The resulting call tree is maintained in F so that subsequent visits to the same method along another path in the tree does not require duplicate calculation. MakeCT also makes use of a call chain, CTC, to identify cycles in the specific

calling sequence being explored. If a cycle is detected in the chain by reference to a method m_c, the path is "capped" with a reference to the call tree rooted at m_c, CT_{m_c}. This ensures termination of all paths in the tree, either by encountering a method calling no others, or by encountering a method calling an ancestor in its call chain.

The MakeCT algorithm requires an SOOPDG, a method, a call chain, and a call forest as input. Each node in the method is examined iteratively for call sites, and the methods referenced at each call site explored through recursive calls to MakeCT. The recursive call utilizes the same SOOPDG; to allow method extraction and examination, the call chain is appended for each recursive call to allow detection of cycles, and the completed call tree is added to the forest for use in subsequent calls. The algorithm is presented in Table 5.2.

The MakeCT algorithm results in a context-sensitive call chain tree. The resulting tree coupled with rewriting semantics may be used to verify feasible call chains identified in the tree. If a mapping of program statement to SOOPDG node is maintained during SOOPDG creation, specific nodes in the tree may be identified with specific call sites within the original program. The worse case cost of the MakeCT algorithm is $O(m!)$, with m representing the number of methods in a program. This case requires that each method references every other method in the program, which results in $n!$ possible unique paths from the root to a leaf in the tree. In the expected case, we recognize that OO programmers attempt to reduce unnecessary coupling between methods, and each method directly interacts with a relatively small number of other methods. The MakeCT algorithm becomes dominated by the number of nodes that must be visited to identify call sites. Since G contains n nodes, and the MakeCT algorithm utilizes the Call Tree

Forest, F, to reduce duplication of effort, MakeCT is expected O(n), where n is the number of nodes in the SOOPDG.

5.4 Inheritance Analysis

Inheritance is tracked in a straightforward fashion through following the flow dependence edges leading from *idef* nodes that define classes. If an *idef* node is the target of a flow dependence edge, then the target is a class definition that is a subclass of the parent node by definition of the SOOPDG, and methods and attributes not explicitly defined in the child node are inherited. This reduces inheritance analysis to simple reachability analysis along flow edges between the *idef* nodes defining classes. The approach is similar to others presented in the literature that use edges to denote superclass-subclass relationships [Zha99, WRW03]. The cost is linear with respect to the number of flow edges connecting superclasses to subclasses.

5.5 Archive and Bytecode Size Reduction

The size of bytecode can vary greatly due to the aggressiveness of the bytecode compiler. The SOOPDG can be used to reduce the size of bytecode through standard dead code analysis. Dead code appears in two forms in the SOOPDG. The first represents *unused* code and is recognized by the lack of outgoing dependence edges. Clearly, nodes having no outgoing flow edges do not contribute to program computations, and nodes having no outgoing control dependence edges cannot provide satisfaction of control dependence criteria to any nodes. The second form

of dead code is recognized as being *unreachable* along a control dependence path from the SOOPDG Start node. SOOPDG nodes not reached through forward control dependence paths may be safely removed from the program, and bytecode is not included for them.

Inclusion or exclusion of methods into an archive does not depend solely on control dependence. Classes are represented by *idef* nodes containing all of the class methods. Control dependence may reach the *idef* node and thus every method in the class. However, this does not mean every method is required in the bytecode. The only methods that require inclusion into the bytecode are candidates to flow along flow edges to a USE node. This is an advantage of the higher-order semantics of the SOOPDG; it reduces the problem to the equivalent problem of variable use in first-order representations.

We do not present a formal algorithm for archive reduction, but present a discussion of the process. Nodes are retained if they have both incoming control dependence and outgoing flow edges. Identification of unnecessary program elements can be done through a depth-first traversal of control dependence edges. For each non-*output* node visited that has no outgoing edges, remove the node and associated incoming edges. For each incoming edge deleted, verify that the head node has other remaining outgoing edges. If it does not, remove the head node and repeat the process. Mark each surviving node as visited. Visited nodes need no further inspection. The process terminates when all control dependence edges have been traversed. The process removes unused code based on the lack of outgoing edges. Unmarked nodes are unreachable through any control dependence path, and may be removed from the graph. The resulting graph contains no dead code and represents the reduced code to be contained in an archive or bytecode representation.

The complexity of this process is based on the number of edges in the graph. Each control dependence edge is visited at most twice, once for the depth-first traversal and potentially once if a target node is removed. Each flow dependence edge is visited only if it is removed from the graph, implying each flow dependence edge is visited at most once. As discussed in Chapter 4, the worse case number of edges is O(n^2), though the expected number is O(n), where n is the number of nodes in the SOOPDG. Thus, the worse case and expected complexity of archive reduction is O(n^2) and O(n), respectively.

Table 5.2: Call Tree Creation Algorithm, MakeCT

```
G = {N, E_c, E_f}
m = myclass.main;
CTC = "empty";
CTF = φ;
Algorithm MakeCT( SOOPDG G, method m, CTChain CTC) {
Global CallTreeForest, F;

CallList = φ;
for (each node, n in m)
    for (each method, m', referenced in n) {
        if ( m' ∉ CTC ) {
            if ( CT(m') ∉ F ) {
                F = F ∪ MakeCT( G, m', CTC:m, F);
            }
            CallList = CallList ∪ getCT( m', F);
        } else {
            CallList = CallList ∪ CT_m';
        }
    }

CT(m) = ( m; CallList );
return CT(m);
}
```

6

COMPARISON WITH OTHER REPRESENTATIONS

Dependence based representations providing varying degrees of support for OO features have been presented in the literature. A number of authors discuss dependence based OO representations only in the context of program slicing, and thus focus primarily on method interactions. These authors discuss method interactions by extending the traditional notion of procedures within the SDG, and do not discuss other OO features in depth [**?**, LH98, CX01, HS04, AH03]. Others support a wider range of OO features in static representation through introduction of numerous node and edge types [MMK94, Zha99, WRW03]. We are interested in representations specifically targeting Java, and supporting a wide range of OO features. We have selected two dependence based representations based on their prominence and support for a wide range of OO features, and compare these representations with the SOOPDG. These representations, the Java Software Dependence Graph and the related Java System Dependence Graph, share characteristics typical of Program Dependence Graphs supporting OO languages. Specifically, these forms extend the PDG and the related System Dependence Graph by adding new node and edge types to capture OO properties. These edge types fill special purpose roles such as package definition, class definition, inheritance, and definition and use of methods. Zhao and Walkinshaw et al use "vertex" and "arc" as opposed to "node" and "edge". For consistency within this thesis, we will use "node" and "edge" to discuss their representations.

6.1 The Java Software Dependence Graph

Zhao [Zha99] appears to be the first to publish a PDG-like representation of a full range of Java features containing class definitions, inheritance, and abstract classes within a single representation. The Java Software Dependence Graph (we denote it JSDG-Z to distinguish from the Walkinshaw representation below) is composed of a collection of interlaced graph structures capable of representing Java programs in a static form. No rewriting semantics are associated with the JSDG-Z. The JSDG-Z incorporates *while* loops through cycles in the control dependence edges of the MDG. The JSDG-Z supports polymorphism through introduction of a polymorphic choice node type.

The basic structure is the *Method Dependence Graph* (MDG) that is essentially an SSA form PDG representing an individual method. The MDG contains *assignment* nodes, and *predicate* nodes, and a unique *Start* node filling the same function as in traditional PDG forms. *Formal in* and *formal out* nodes correspond to SOOPDG *def* and *xfer* nodes, respectively. Control and flow dependences are represented within the MDG through dependence edges in the traditional fashion.

Nodes corresponding to program statements calling a method are connected to a *method call node*, which in turn is connected to *actual in* and *actual out* nodes. These nodes approximately correspond to SOOPDG *def* and *xfer* nodes identifying call sites and supporting method input and output. The actual in and actual out nodes at a single call site are connected by *summary dependence edges* to facilitate static program analysis requiring backwards traversal of the graph. The summary edge is required to allow a backwards traversal to continue across the calling site without entering the method and losing the context of the specific call. The method call site is connected to the method start node through a *call*

dependence edge, which serves the same purpose as a control dependence edge in traditional PDGs.

Classes are represented within a *class dependence graph* (CDP) as a collection of nodes depicting the class variables, instance variables, and methods. The CDP contains a unique *class start node* connected to the component variables and methods through *class-membership dependence edges*. Inheritance is represented through duplication of the inherited class and instance methods and variables within the dependence graph for the subclass. *Interfaces* are represented in a similar manner to class representation. An *interface dependence graph* (IDG) consists of an *interface start node* connected to the member method declarations through *interface-membership dependence edges*. The method declarations are in turn connected to each implementation of the method. Polymorphism is represented through a *polymorphic choice node*, providing possible destinations of a method call.

Figures 5.1 and 5.2 summarize the JSDG-Z node and edge sets. The structure of the JSDG-Z graph is more complex than the SOOPDG due to the introduction of these specialized nodes and edges. The JDSG-Z also departs from traditional dependence graph concepts in several ways. Cycles and loops are permitted in the graph, which deviates from the acyclic structure of the traditional dependence graph. This complicates control flow analyses by allowing a node to have multiple control parents, and complicates data flow analyses by violating the SA property. Control dependence edges are permitted to initiate at non-predicate nodes, which confounds the notion of control dependence and program decision points. Specifically, call sites act as control parents to method *Start* nodes, which implies that analysis requiring identification of control dependence paths or specific program control points is complicated by the potential to inspect all node

types. Edges representing membership in classes or packages do not correlate to any of the true dependences identified by Kuck et al [KKP81] that motivated definition of the PDG. These edges, while useful for investigation into program structure, do not provide information as to program meaning.

6.2 The Java System Dependence Graph

Walkinshaw et el [WRW03] proposed modifications to the JSDG-Z representation. They present the Java System Dependence Graph (JSDG), which we denote as JSDG-W to differentiate from Zhao's Java Software Dependence Graph. The JSDG-W provides a different mechanism for representing polymorphism and also represents abstract classes in addition to the interfaces defined by Zhao [Zha99]. Like Zhao, Walkinshaw et al [WRW03] provide for Java's OO language characteristics employing special purpose nodes and edges to the basic PDG definition. The JSDSG-W utilizes most of the features of the JSDG-Z. We discuss differences below.

The JSDG-W handles method calls slightly differently than the JSDG-Z to support polymorphism without the use of the polymorphic choice node [WRW03]. Method call sites are expanded into calling nodes and actual-in, actual-out nodes, but the JSDG-W provides a structure for each potential polymorphic method that may reach the call site. The JSDG-W requires that a copy of a class graph structure be incorporated in the overall graph for each instantiation of the object in the program. The techniques introduced by Zhao [Zha99] to support interfaces are used by Walkinshaw et al [WRW03] to support abstract classes.

As with the JSDG-Z, there is no discussion presented regarding rewriting nor aliasing for the JSDG-W. The usage of the *method dependence* edge permits

control dependence to initiate from non-predicate nodes in the case of method calls; this confounds the notion of control dependence. Loops are represented using cycles in the graph.

Node	Usage	SOOPDG Equivalent
Start	Predicate node specifying the start of control flow for the program and for methods.	Start
	Indicates entry into class, interfaces, and packages.	Idef
Formal In	Placeholder for formal input parameters for a method.	Def
Formal Out	Placeholder for value of result returned to calling environment.	Xfer
Actual In	Placeholder for actual method input values at calling sites.	Xfer
Actual Out	Placeholder for actual method output values at calling sites.	Def
Assignment	Corresponds to assignment statements as in traditional the PDG.	Assignment
Predicate	Corresponds to program decision points as in the traditional PDG.	Predicate
Call	Specifies a call site for a method, either in a class or an I/F.	Def
Polymorphic Choice	Selects correct method instance among polymorphic methods.	n/a (1)

(1) SOOPDG uses SFU to disambiguate among potential methods being called.

Figure 6.1: Node Set in JSDG-Z and JSDG-W

6.3 Improvements Provided by the SOOPDG

The SOOPDG has several advantages over the JSDG-Z and JSDG-W including fewer element types, cleaner semantics, improved clarity, and smaller graph size. Primarily, the SOOPDG is able to represent OO features such as inheritance and object aliasing through traditional PDG edge types. This results in far fewer edge types than the JSDG-Z and JSDG-W forms, which in turn results in cleaner analysis techniques, and allows for direct leveraging of traditional program analysis techniques. The SOOPDG has an associated rewriting semantics, which results in a more compact static program representation, and more effi-

Edge	Usage	SOOPDG Equivalent
Control	Predicate node specifying the start of control flow for the program and for methods.	Control
	Indicates control dependence between call site and actual in/out nodes.	n/a (1)
Method Call	Supplies control dependence between call site and method start.	n/a (2)
	Supplies control dependence between I/F call sites and method start.	
Flow	Explicitly represents flow dependence as in traditional PDG.	Flow
Parameter In	Provides flow of values from actual in to formal in nodes.	Flow
Parameter Out	Provides flow of values from formal out to actual out.	Flow
Transitive Flow	Represents data flow summary between actual in and actual out nodes.	n/a (2)
Class Inheritance	Designates inheritance from superclass to subclass.	Flow
I/F Inheritance	Designates inheritance from I/F to	Flow
Class Membership	Designates membership in a class.	Idef Node
I/F Membership	Designates membership in an I/F, from the Start vertex to call sites within the I/F.	Idef Node
Package Membership	Designates membership in a package.	Node Decoration
Implement Abstract Method	Connects virtual methods to implementations. (Walkinshaw et al only)	Flow

Notes: (1) SOOPDG call site does not require actual in/out nodes.
(2) SOOPDG creates a copy of the call site and instantiates method between the call and return version of the site, using same control criteria.

Figure 6.2: Edge Set for JSDG-Z and JSDG-W

cient analysis techniques, as discussed below. The ability to define a rewriting semantics ultimately allows formal reasoning regarding program transformations and the maintenance of program semantics that is not possible with the existing static representations (see Section 8.2).

The SOOPDG exhibits an advantage in terms of clarity of control dependence. The SOOPDG requires that control dependence edges emanate only from *predicate* nodes, as opposed to the JSDG-Z and JSDG-W which allow call sites to be the source for control dependence. Example 1, in Figure 5.3, illustrates the ambiguity arising in the JSDG-W representation. This figure represents a method

call in the SOOPDG and JSDG-W systems. For clarity, only flow and control dependence edges are shown. The calling sites are shown in grey boxes. Because the JSDG-W uses control dependence edges to associate actual out and actual in parameters, Node 5 in the JSDG-W is control dependent on nodes 1, 3, and 4. No determination can be made simply by considering control dependence edges as to what control dependence criteria will permit node 5 to execute. A similar confusion arises with the method's predicate node, which is control dependent on node 2 when the actual issue at stake is a requirement that a value flow out of the method. The use of control dependence in this sense convolutes control and flow dependences, and could require traversal of flow edges to analyze properties of control flow in the program. In contrast, the SOOPDG only allows predicate nodes to initiate control dependence. This property of the SOOPDG maintains clarity and continuity in control dependence paths and allows a separation of flow and control dependence issues while reasoning about program properties. For example, the development of the Semantic PDGs [Sel90a], and the SSA and SFU PDGs [CFR89] all depended on well behaved control flow (no breaks or goto's) represented by control dependence edges connecting *predicate* nodes in a continuous path.

The number of nodes required for the SOOPDG to represent a given program is smaller than the JSDG-Z and JSDG-W forms. The primary reason for this is that the JSDG-W duplicates the entire class dependence graph for each object instantiated of the class, while the SOOPDG adds nodes only for instance variables and modified class variables at the point of object construction. A secondary reason is the inefficient representation of polymorphic methods at call sites within the JSDG-W. The JSDG-W requires duplicate nodes transporting method input/output parameters for each polymorphic form, while the SOOPDG utilizes one node set regardless of which form of the method is invoked at run

time. This node penalty is not significant for a single call site, but could contribute to an O(n) growth for programs containing polymorphic utility methods used widely throughout the program. Still another inefficiency is in the representation of extended methods in that the entire superclass is duplicated within the subclass of the JSDG-Z and JSDG-W representations. The SOOPDG requires only the additional or modified class features to be explicitly represented within the subclass. Each of these inefficiencies is summarized in Figure 5.4.

The examples provide illustration of the JSDG-W inefficiencies addressed by the SOOPDG. Example 3, in Figures 5.5 through 5.8, provides another representative situation. We present a program shown by Walkinshaw et al [WRW03] encompassing a wide range of Java features. The program is reproduced in Figure 5.5, and the resulting SOOPDG is given in Figures 5.6, 5.7, 5.8. The JSDG-W given by Walkinshaw et al [WRW03] is larger than the equivalent SOOPDG (160 nodes compared to 108), and would be more so with repeated instantiations of a single class. The smaller program representations improves the efficiency of analysis on the static program representation (see Chapter 5).

The smaller graph sizes and reliance on a rewriting semantics results in more efficient program analysis. As an example we compare program slicing, which forms the basis of many program analyses as noted in Chapter 5. The program slicing algorithms provided for the JSDG-Z and JSDG-W follow the two-phase procedure of Hortwitz et al. [Zha99, WRW03], as described in Section 5.1. Summary edges are created at all program call sites prior to creating the slice, through exploration of the called methods. The slicing algorithm visits all nodes without entering called methods by traversing the summary edges during the first phase of the algorithm, and the called methods are then entered during the second phase. The slicing algorithm presented for the SOOPDG is more efficient in that it does

not require summary edges, and therefore explores only the methods encountered during an individual slicing operation. The resulting slice is also smaller due to the more efficient SOOPDG representation of objects and polymorphic calling sites, which is a direct result of the existence of the rewriting semantics.

The SOOPDG also has an advantage in that its dependence on traditional PDG forms allows it to support parallel execution of a single threaded program. The inherent ability of the PDG rewriting semantics to support parallel operations was discussed in Chapter 2. Rewriting semantics for the JSDG-Z and JSDG-W have not been presented, but there are features of these graphs that appear to be barriers to parallel operations. The JSDG-Z contains loops in the control flow edges. This violates the Single Assignment property and allows confusion over the value of variables available to differing loop iterations, which contradicts the author's intention to base the method dependence graph on the SSA form of the PDG. The authors do not provide a rewriting semantics to resolve how loops may be expanded and loop dependent values maintained. Since loop structures are often targets of parallelization techniques, this appears to be a challenge to parallel rewriting of the graph. The JSDG-W allows multiple call sites for methods to flow data in and out of the method. The choice of outgoing edge used for a returned value from the method appears to be related to the control edges connecting the actual in and actual out parameters at the call site. That is, the value is returned to the actual out node having its control dependence criteria satisfied. Without a rewriting semantics published for this representation, it is difficult to see how two or more call sites acting in parallel would disambiguate the location of the returned value from the method.

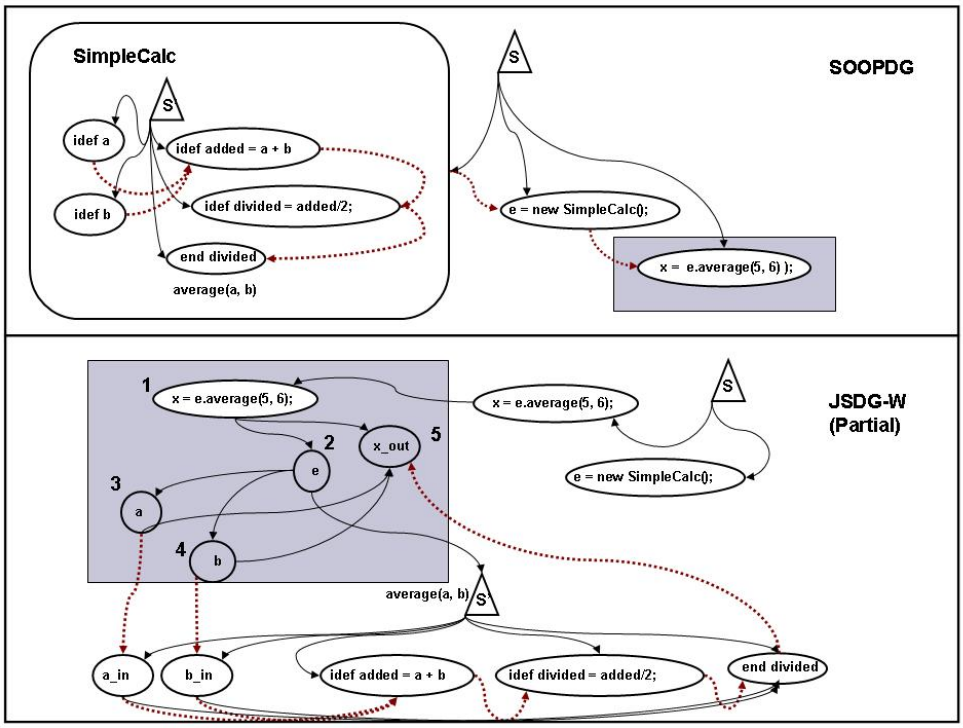


Figure 6.3: Example 1: Method Call in SOOPDG and JSDG-W

Let:

$|C_i|$ = Number of nodes in Class i.

$|V_i|$ = Number of class and instance variables in Class i.

$NOBJECTS_i$ = Number of instantiated objects of Class i.

$NPARAMS_m$ = Number of input/output parameters for method m.

$NPOlY_m$ = Number of polymorphic forms for method m.

Nodes Required to Represent k Classes:

JSDG-W: $\sum_{i=1}^{k} NOBJECTS_i * |C_i|$

SOOPDG: $\sum_{i=1}^{k} |C_i| + NOBJECTS_i * |V_i|$

Nodes Required to represent a method call site:

JSDG-W: $(1 + NPARAMS_m) * NPOLY_m$

SOOPDG: $1 + NPARAMS_m$

Nodes Required to Represent Inheritance of Depth d:

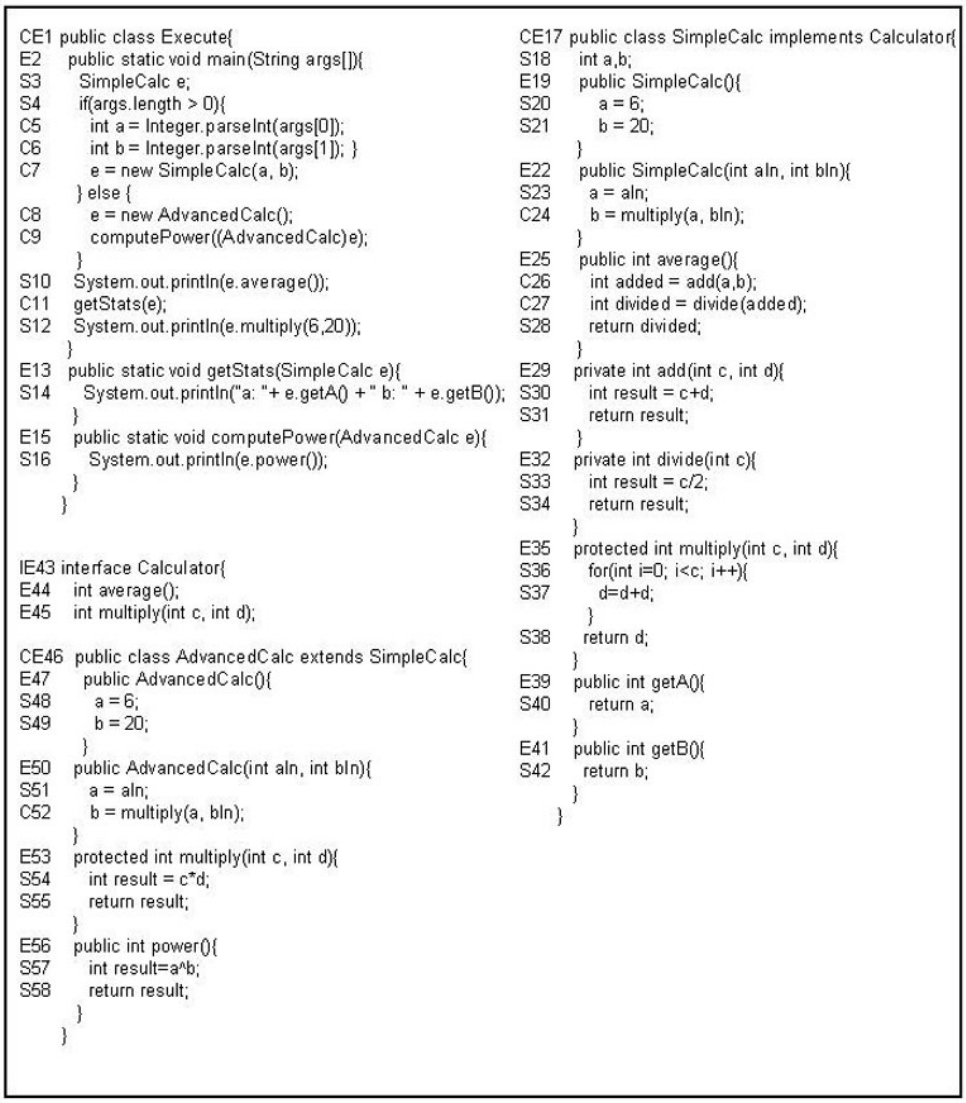
Let $|C_0|$ represent the size of the original class to be extended.

Let $|\delta C_i|$ represent the size of features unique to the ith class compared to the C_{i-1} class being extended.

JSDG-W: $(d+1) * |C_0| + d * |\delta C_1| + \ldots + |\delta C_d|$

SOOPDG: $|C_0| + |\delta C_1| + \ldots + |\delta C_d|$

Figure 6.4: SOOPDG and JSDG-W Size Comparisons



```
CE1 public class Execute{
E2    public static void main(String args[]){
S3      SimpleCalc e;
S4      if(args.length > 0){
C5        int a = Integer.parseInt(args[0]);
C6        int b = Integer.parseInt(args[1]); }
C7        e = new SimpleCalc(a, b);
        } else {
C8        e = new AdvancedCalc();
C9        computePower((AdvancedCalc)e);
        }
S10   System.out.println(e.average());
C11   getStats(e);
S12   System.out.println(e.multiply(6,20));
      }
E13   public static void getStats(SimpleCalc e){
S14     System.out.println("a: "+ e.getA() + " b: " + e.getB());
      }
E15   public static void computePower(AdvancedCalc e){
S16     System.out.println(e.power());
      }
    }


IE43 interface Calculator{
E44   int average();
E45   int multiply(int c, int d);

CE46  public class AdvancedCalc extends SimpleCalc{
E47     public AdvancedCalc(){
S48       a = 6;
S49       b = 20;
        }
E50   public AdvancedCalc(int aIn, int bIn){
S51     a = aIn;
C52     b = multiply(a, bIn);
      }
E53   protected int multiply(int c, int d){
S54     int result = c*d;
S55     return result;
      }
E56   public int power(){
S57     int result=a^b;
S58     return result;
      }
    }
```

```
CE17 public class SimpleCalc implements Calculator{
S18    int a,b;
E19    public SimpleCalc(){
S20      a = 6;
S21      b = 20;
       }
E22    public SimpleCalc(int aIn, int bIn){
S23      a = aIn;
C24      b = multiply(a, bIn);
       }
E25    public int average(){
C26      int added = add(a,b);
C27      int divided = divide(added);
S28      return divided;
       }
E29    private int add(int c, int d){
S30      int result = c+d;
S31      return result;
       }
E32    private int divide(int c){
S33      int result = c/2;
S34      return result;
       }
E35    protected int multiply(int c, int d){
S36      for(int i=0; i<c; i++){
S37        d=d+d;
         }
S38      return d;
       }
E39    public int getA(){
S40      return a;
       }
E41    public int getB(){
S42      return b;
       }
     }
```

Figure 6.5: Example 3: Program Presented by Walkinshaw et al [WRW03]

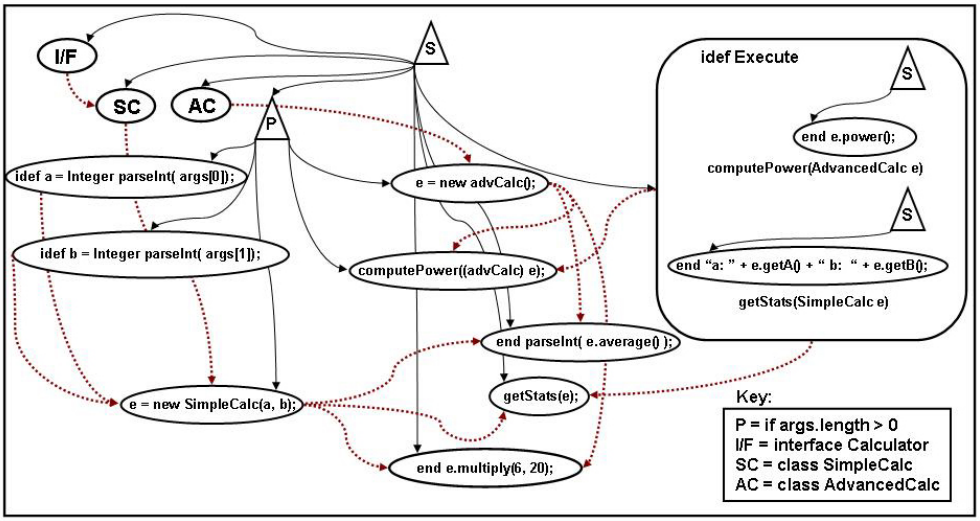


Figure 6.6: Example 3: SOOPDG for Figure 5.5, *main* method and *Execute* class (1 of 3)

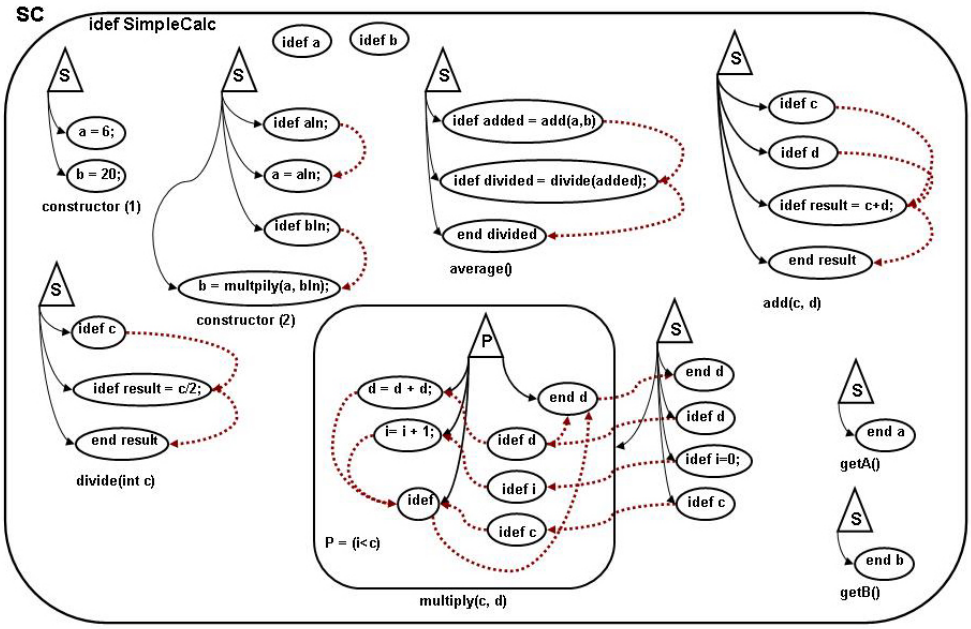


Figure 6.7: Example 3: SOOPDG for Figure 5.5, SimpleCalc class (2 of 3)

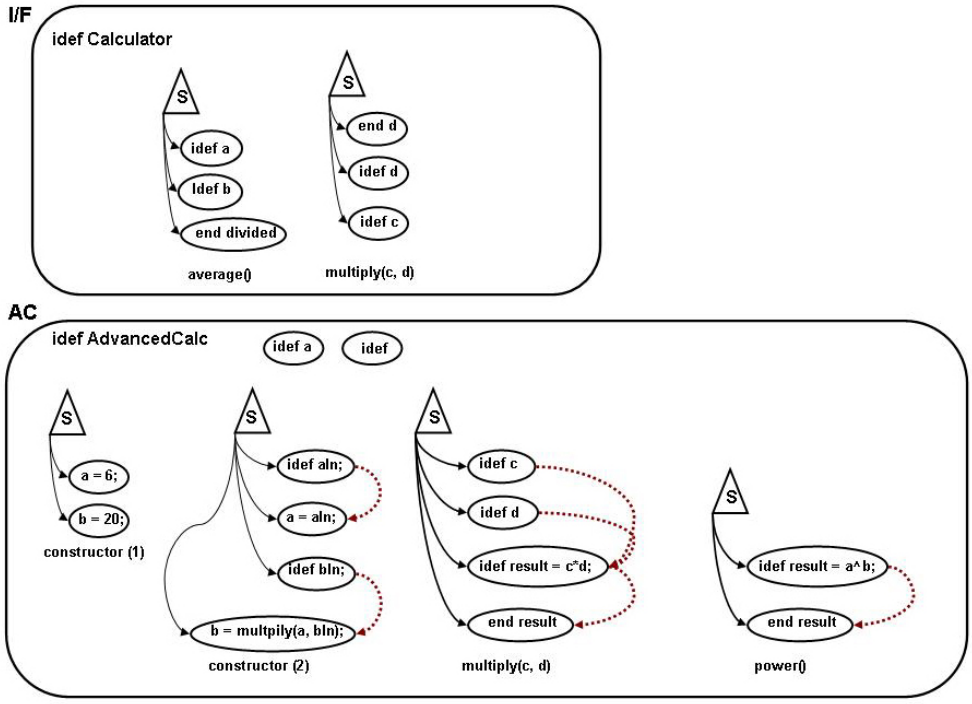


Figure 6.8: Example 3: SOOPDG for Figure 5.5, Calculator Interface and AdvancedCalc Class (3 of 3)

7

THE SIMULATION DEPENDENCE GRAPH

This chapter presents the Simulation Dependence Graph (SDG), a natural extension to the SOOPDG with applications in the modeling and simulation field. Specifically, we define the SDG and establish the foundation for a line of research investigating the use of the formalism in the representation and analysis of simulation systems. While this discussion is specific to simulation systems, the concepts have application in the fields of component based software.

Component based software development is an emerging topic in the software engineering discipline [Par03, Lau06]. The success of JavaBeans is a prominent example [Mic05] of its benefits. The concept of component re-use and issues associated with creating new applications through the composition of existing components is common to both fields of study. Specifically, notions discussed here such as syntactic and semantic interoperability transfer directly between the problem domains. Simulation composability and software component based development both require that appropriate components be identified for use, their functionality be measured against requirements, and the resulting system be stable [WGH03, Par03].

The SDG views computation at a coarser granularity than the SOOPDG. The elements of traditional PDGs and the SOOPDG roughly correspond to program statements and relationships between these statements and allow for analysis at the program level. In contrast, the elements of the SDG correspond roughly to

model features, model components, and the relationships between them. The SDG allows for analysis at the system level. Program analysis techniques developed using the PDG may be leveraged to allow static and dynamic analyses of simulation systems. Here we present basic definitions used in the development of the SDG, provide an example simulation, and discuss potential uses for this representation.

7.1 Background Definitions

Informally, a model is a mathematical approximation of a real world system representing a finite number of measurable attributes, or features of interest, of the real world system [WGH03]. The model specifies a set of attributes, coupled with mathematical functions that update the values of these attributes to represent sufficiently the state and behavior (state changes) of the real world system being investigated. A simulation is a process that investigates the behavior of the model [WGH03] through a series of updates to the attribute values. For our purposes, the model is implemented through software, and the simulation is the act of executing the software program(s). We represent the model implementation using the SDG in the same way that the PDG represents programs implementing computational algorithms. To represent a simulation, there should be a correspondence between the mathematical model specification in the modeling space and the implementation of the model in the simulation space [WGH03]. A graph representation meets this requirement, in that a rewriting semantics may be defined that corresponds to the original program semantics [Sel90a].

In this section we develop a set notation that acts as an intermediate representation in the transition from a model specification to an SDG representation

of the model. This set notation is based largely on that developed by Petty et al [PW03b] in their discussions about semantic composability of models. We begin by defining individual attributes and their update functions, and continue to define models and simulations.

We define a feature of a model, denoted as f, as a measurable aspect of a real world system represented in the model (Definition 37). We define an attribute, denoted as a, as a range of values (Definition 38) that may be assigned to a feature. This definition is similar to Petty's definition [PW03b], except that we do not restrict attribute values to integers. This definition is also in accordance with the requirement that simulations contain explicitly defined boundaries and constraints [WGH03]. We refer to an individual value residing in an attribute as an *attribute value*, and denote it as α. We further denote that an attribute vector, A, is a tuple of attributes (Definition 39), and an attribute value vector, A_V, is a tuple of specific values from each attribute contributing to the vector (Definition 41). An attribute update function, $F(A_V)$, is a process that receives a (potentially empty) vector of attribute values as input, and selects a value from the attribute it is associated with as output (Definition 42). Attribute update functions provide a mapping from a vector of attribute values to attribute values, $a \Leftarrow F(A_V)$.

Definition 37 (Feature) *A feature, f, is a representation in a model of a measurable aspect of the real world system being studied.*

Definition 38 (Attribute) *An attribute, a, is a range of values, that is not necessarily continuous. We define attributes through the following recursive rules.*

1. *i is a single value in $Z \cup \perp$.*
2. *r is a single value in $R \cup \perp$.*

3. $i_o - i_f$ *designates a range of values in* Z *from* i_o *to* i_f*, inclusive.*

4. $r_o - r_f$ *designates a range of values in* R *from* r_o *to* r_f*, inclusive.*

5. $a = \phi$ *designates an attribute containing no members.*

6. $a = \{\alpha | i_o \geq \alpha \geq i_f, \alpha,\ i_o, i_f \in Z\}$ *designates an attribute.*

7. $a = \{\alpha | r_o \geq \alpha \geq r_f, \alpha,\ r_o, r_f \in R\}$ *designates an attribute.*

8. *if* a_1 *and* a_2 *are attributes, then* $a = a_1 \cup a_2$ *is an attribute.*

Definition 39 (Attribute Vector) *An attribute vector,* A*, is an ordered tuple of attributes,* $A = (a_1, a_2, a_3, \ldots, a_k)$

Definition 40 (Attribute Value) *An attribute value,* α*, is an individual member of an attribute.*

Definition 41 (Attribute Value Vector) *An attribute value vector,* A_v*, is an ordered tuple of attribute values,* $A_v = (\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_k)$*, such that* $\alpha_i \in a_i$*, for* $1 \leq i \leq k$*.*

Definition 42 (Attribute Update Functions) *An attribute function,* $F_i(A_V)$*, takes an attribute value vector and produces an attribute value,* $\alpha \in a_i$*.*

We define a model, M, as a set of features with associated attributes and update functions (Definition 43). The values selected for all model attributes at any point in the course of a simulation represent the state of the simulation. The update functions serve to modify the current values selected from each attribute in the model as it transitions from state to state. Since models are represented in set format, model composition may be defined in terms of set union ($M_3 = M_1 \cup M_2$), and decomposition can be defined as set subtraction ($M_1 = M_3 - M_2$).

Definition 43 (Model) *A model, M, is a set of tuples of features, attributes and attribute update functions.*

$M = \{(f_1, a_1, F_1), (f_2, a_2, F_2), \ldots, (f_n, a_n, F_n)\}$.

This set notation form of model representation can be used to discuss model properties in a static format. We define two such model properties, completeness and ambiguity, useful in defining the notion of a simulation. Completeness refers to the capability of a model to supply input values for its update functions. A complete model contains all attributes required as input for all update functions in the model (Definition 44). Simulation systems are often constructed by composing models as simulation components. For example, an aircraft model may be composed from a geometric model defining airframe shape, an engine model defining engine performance across various flight regimes, and an environmental model describing the atmospheric effects on the aircraft. No single component may be complete, though the aircraft model may be complete once composed. The second property of interest to us is the notion of ambiguity. A model is ambiguous if there is more than one attribute and update function pair providing values for the same model feature of interest (Definition 45).

Definition 44 (Completeness) *A model, M, is complete if all attributes required to supply values to all update functions reside within the model. That is* $\forall F(A_V) \in M, \forall \alpha_i \in A_V, \exists (f_i, a_i, F(A_{Vi})) \in M$

Definition 45 (Ambiguity) *A model, M, is ambiguous if it contains more than one attribute and update function providing values for the same model feature. M is ambiguous if* $\exists i, j$*, such that* $(f_x, a_i, F_i(A_V)), (f_x, a_j, F_j(A'_V)) \in M$*, and* $i \neq j$.

A simulation is a process acting on a model that updates one or more attributes one or more times (Definition 46). We will require that the model be

complete, so that all inputs to all update functions reside within the model, and non-ambiguous, so that deterministic sources exist for inputs to update functions. In order to perform the simulation process we require some form of sequencing designating the order in which attributes are updated. Let A_s denote a special attribute vector that designates this sequence of attribute updates, then a simulation can be thought of as a function of the model and the sequencing attribute vector, $S = F(M, A_s)$.

Definition 46 (Simulation) *Given a complete, non-ambiguous model, M, and a sequencing attribute vector, A_s, we define a simulation, $S = F(M, A_s)$ to be a function operating on M by updating its attributes in accordance with the attribute update functions and the sequences designated by A_s. This process may be iterative and does not require termination criteria.*

This set notation form of a model and the simulation process is used to specify a model in a suitable format to transition to the SDG form.

7.2 Definition of the SDG

The Simulation Dependence Graph (SDG) is composed of a node set and three edge sets (Definition 47). The node set represents attributes of the real world entities being simulated, their allowable value ranges, and the functions that update the attribute values during the course of a simulation. Nodes are annotated as PUSH or PULL nodes, meaning that attribute updates are supplied to other nodes upon calculation or upon request, respectively. This allows for imperative and lazy model semantics to be represented. The three edge sets represent data dependences, entity memberships, and sequencing constraints. The data depen-

dence edges explicitly model input-output interactions between attributes; this information can be used to analyze and optimize system behavior in the same manner that modern optimizing compilers perform program data-flow analysis. The entity membership edges explicitly associate attributes belonging to the same physical entity; this information can be used to distribute entity-dependent behaviors and constraint across a set of nodes representing a single simulated object. The sequencing constraint edges explicitly represent the order in which attribute values are updated; these edges may be used to demonstrate the efficiency of one manner of scheduling tasks over another. The sequencing edges may model uni-processor or parallel processor environments.

Definition 47 (Simulation Dependence Graph) *The Simulation Dependence Graph,* $G = \{N, E_f, E_c, E_e\}$*, where:*

1. N *is the node set. Each node,* $n \in N$*, contains an attribute, its update function, and a tag designating the node as a PUSH or PULL node.*

2. E_f *is the set of flow dependence edges. Each edge,* $e_f \in E_f$*, identifies an ordered pair of nodes,* (n_i, n_j)*, such that* n_j *is flow dependent on* n_i*.*

3. E_c *is the set of sequencing edges where each edge,* (n_i, n_j)*, describes the order in which attribute updates are performed in a pairwise fashion.*

4. E_e *is the set of undirected entity edges containing edges of the form* (n_i, n_j)*, indicating that nodes* n_i *and* n_j *belong to the same entity.*

The node set, N, and flow edge set, E_f, are derived directly from the set notation form of the model. The sequencing edges, E_c, and entity edge set, E_e, are typically specified by the model designer.

7.3 Ants on a Log: A Simulation Example

We will use a simple model of ants walking back and forth along a log as an example of a simulation and corresponding SDG. The model is discussed below and defined in Figure 7.1, while the corresponding SDG is shown in Figure 7.2.

The Ants on a Log (AL) simulation has the following components:

1. Entities representing ants are walking back and forth on a level log.
2. Ants cannot pass each other; the log is effectively 1-D.
3. Ants turn to avoid falling off the log or colliding with other ants.

This simple model contains the key elements of all simulations: an environment (the log), agents (the ants), agents acting with the environment (ants - log interactions), and agents affected by other agents (ant-ant interactions).

This simple model can easily be made remarkably complex. For example, the model may include expanding or shrinking logs, passing zones, variable speed ants, territorial ants, and teams of ants acting in concert.

7.4 The SDG and Model Analysis

Model composability is important because low-level models are integrated to compose higher system-level simulations, and composition must be stable when parts are replaced [WGH03]. While composition in the computational sense implies the combination of simple functions to create complex behaviors, the Modeling and Simulation community uses the term to imply the ability to combine existing models into new application domains [KN00]. The term *interoperability* is also

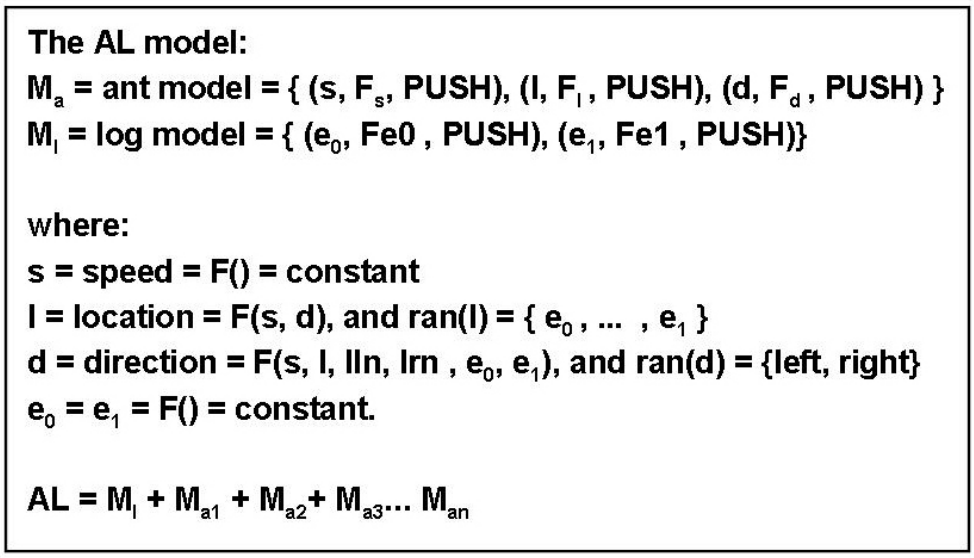


Figure 7.1: Definition of the Ants on a Log Simulation

used to describe this concept of composability. The notion that overall system behavior can be described or maintained purely through examination of model components is not valid, as models may produce surprising system level behaviors through emergent interactions of the component models [Par03]. It is also not possible, in the general case, to predict what effects replacement of model components will have on the overall model. Textual (syntactic) differences between models can be identified, but it is in general undecidable to determine semantic differences [Hor90].

The composability problem of replacing one model with an equivalent alternative has been addressed in various ways. The problem is typically broken into two components: syntactic and semantic composability [PW03a]. Syntactic composability can be approximated through examination of model structure, and function inputs and outputs. While this addresses the syntactic composability

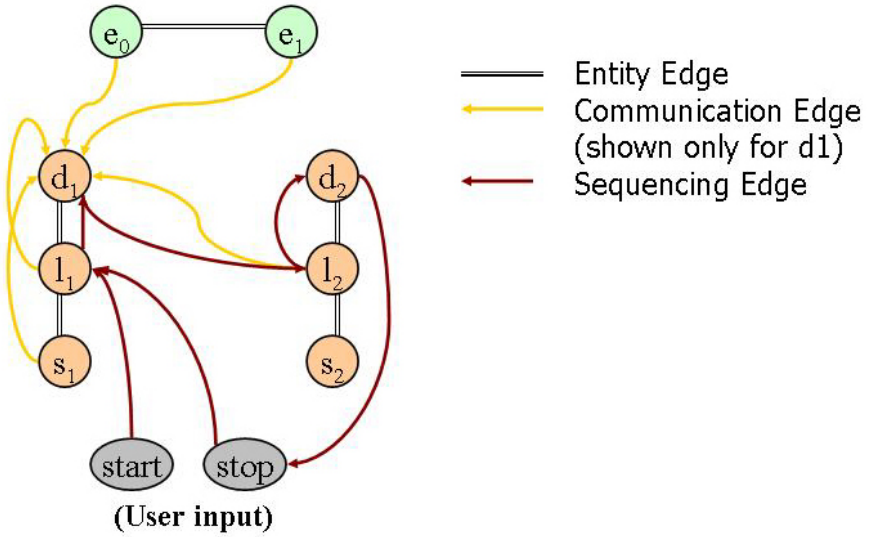


Figure 7.2: SDG for the Ants on a Log Simulation

of models, it does not assure that software representations of the models are composable.

Determination of semantic composability is not Turing computable in the general sense. Semantic composability may be difficult because different users may have different contexts [WGH03]. One proposed method attempts to determine if two model components are semantically equivalent through the use of meta-data added to the model. For example, editor tags are one way to supply meta-data information for an underlying model [Hor90]. This assumes the new model was created by editing an old one, and does not generalize to two models created in completely different processes. Another technique proposed for semantic composability is to provide an internally consistent library of potential component models that may be combined in various ways to meet specific

needs [KN00]. A third method is to estimate model behaviors through a form of profiling by examining actual run time behaviors or behavior predictions of two models [PW03a].

The SDG supports syntactic composability of models directly, and provides a new mechanism to address semantic composability. Both of these techniques require further development through future research efforts. Syntactic composability is addressed through duplication of graph structures across some interface. For example, a subcomponent may be removed from the overall model, leaving a "hole" in the graph, with dangling edges. Any replacing subcomponent must fill the "hole" such that the resulting model has no dangling edges. Semantic composability may be investigated using traditional PDG analysis techniques. We propose using the notion of a *program slice* as a measure of semantic composability.

A program slice identifies, for a given program point, all nodes in a graph affecting the computation at that point, or affected by the computation at that point [HRB90]. The notion of using program slices as a measure of semantic difference for traditional PDGs is discussed by Horwitz [Hor90]. For an SDG, we may build a slice for each node in the model to identify all nodes affecting, and affected by, a given node. Thus, two models (components) may be compared by comparing the slices generated for each node in each model (component). This provides a measurable comparison of the computations each model relies on, and the computations affected by each model. Ranges of values for each attribute may also be compared to allow comparison of the domain and ranges for alternate update functions, thus providing an indirect estimate of the range of model behaviors. A sample program slice is provided for the Ants on a Log Model in Figure 7.3.

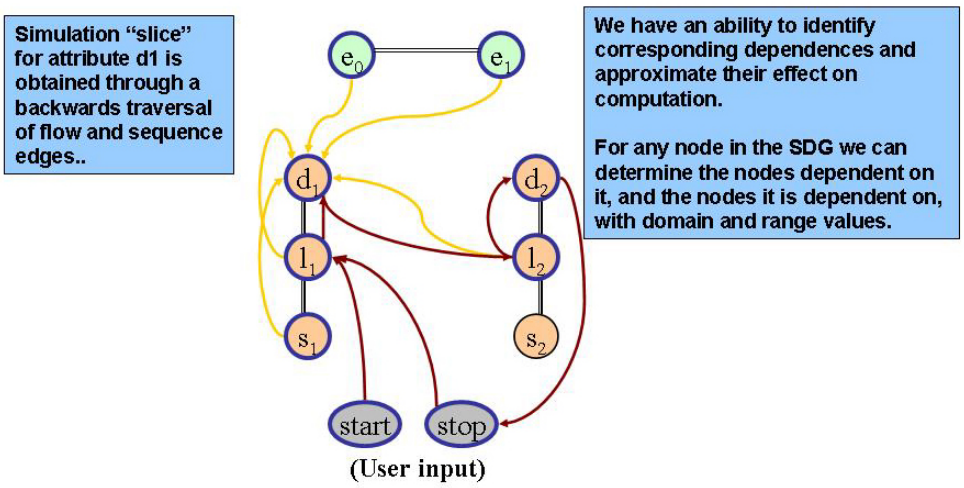


Figure 7.3: SDG Slice for the l_1 Node in the Ants on a Log PDG

While the SDG is still in its developmental stages, it leverages established PDG techniques to represent simulation systems in a manner allowing static and dynamic analysis of the systems. The analogy to program slicing is just one such instance of how techniques and insights developed for the PDG hold promise for formal reasoning about complex simulation systems.

8

SOOPDG EXTENSIONS AND FUTURE WORK

This thesis has presented a definition of the SOOPDG and described program analysis techniques that may be performed using this representation. Advantages over other Java and OO dependence based representations have also been presented. We plan to continue this research to more fully develop the SOOPDG as a formal model of computation in an OO environment and also as a tool to enhance program performance. The specific areas of interest for future work include inclusion of multi-threaded applications, development of a formal denotational semantics, and adaptation of the SOOPDG to run time optimization schemes.

8.1 Multi-threading and Unstructured Control

In this thesis, we limit the Java-like language J to a single thread. Actual Java applications are typically multi-threaded, which implies that incorporating multi-threaded behavior into the definition of J and the SOOPDG is a useful extension. The literature contains examples of static dependence-based analytical techniques developed for multi-threaded Java programs. These techniques typically generate program slices involving elements from multiple execution threads. Nanda, Krinke, and Hammer and Snelting [Nan01, Kri03, HS04] present slicing techniques for multi-threaded Java programs through direct extensions of System

Dependence Graphs and interprocedural slicing methods presented by Horwitz et al. [HRB90]. Zhao [Zha98] presents a slicing method capable of slicing multi-threaded Java programs by adapting Horwitz et al [HRB90] methods to the JSDG-Z representation. The current literature supports static representations of multi-threaded Java programs, with no presentations of rewriting semantics.

Extending the SOOPDG representation and rewriting semantics to support multi-threaded executions will allow the advantages demonstrated for the SOOPDG to be applied to a fuller range of applications. The SOOPDG intrinsically supports parallel operations of a single execution thread, as discussed in Chapter 5. The operations are deterministic as the sequences specified by dependence edges are respected [Sel89]. Though deterministic forms of multi-threaded programs can be created through explicit synchronization features such as *notify* and *wait* [Zha98], multi-threaded programs are not guaranteed to be deterministic. This presents a difficulty in establishing a rewriting semantics that correctly represents program execution and potential input-output behavior in a non-deterministic environment. This challenge may potentially be overcome by adapting techniques to represent explicitly parallel programs using existing PDG forms [SHW93, LMP99]. If a mechanism is not developed to represent the non-deterministic aspects of parallel processing, there is still utility in adapting the SOOPDG to deterministic, thread safe applications.

There is also value in extending the model to allow forms typically used in real world programs that introduce unstructured control flow. The widespread use of the "try-catch-throw" mechanism of exception handling is suficient motivation for the extension. Neither the J language nor the SOOPDG presented in this thesis currently supports "try-catch-throw." The J language is currently restricted to allowing at most one "return" statement in a method, and does not support

"break" statements. Real world Java methods often contain multiple "return" statements accommodating different control flow paths within the method. Similarly, programmers often use "break" statements to provide convenient exits from compound statements. These are commonly used elements and their inclusion in the model would increase the robustness and utility of the SOOPDG.

8.2 Formal SOOPDG Semantics

Parsons [Par92] developed a complete denotational semantics for the PDG to allow formal reasoning of program properties. The PDG form developed by Cartwright, Felleison and Parsons upheld the SFU property through use of the valve node [CF89, Par92]. The compositional nature of the valve node, as opposed to the ϕ-node used in the SSA PDG form, was instrumental in developing the semantics presented by Parsons. Since the SOOPDG has its basis in the SFU form it is amenable to an extension of the semantics to include the OO features. Extending Parsons' semantics to the SOOPDG allows formal reasoning regarding program transformations and maintenance of program meaning.

8.3 Dynamic Performance Improvement

We propose investigating the use of the SOOPDG to enhance performance dynamically. Current literature provides examples of research efforts focused on improving the performance of Java (and OO languages) by identifying parallelization and compilation opportunities in the run time environment. In this discussion we will differentiate between compilation to byte-code occurring prior

to program execution and so-called Just In Time (JIT) compilation occurring during execution by using the term "compilation" for the former and "JIT" for the latter. Successful application of automated parallelization and JIT techniques require that the same fundamental question be answered: "Does the performance benefit obtained outweigh the overhead cost of performing the technique?" The need to address this issue for automatic and dynamic parallelization techniques has been discussed for some time [Tre79, Den94, LH96, ZC91] and continues to be investigated [OH02]. The need to address this issue for JIT techniques is displayed in current literature as well [VE00, Wha01, SYN03, BV05]. Arnold et al [AFG05] provide an taxonomy of numerous techniques proposed to perform run-time optimizations.

Current techniques impose a division between compile-time and run-time environments [FO98, AFG05]. This makes both compilation and execution less efficient, as information currently accessible in only one environment is often required in the other for optimal decision making [FO98]. Decisions made during compilation must make conservative assumptions to compensate for the lack of information regarding the run time environment. Decisions made during program execution suffer from a lack of information regarding program structure and dependences normally uncovered during compilation. Existing dynamic optimization techniques rely on profiling of past execution history to estimate future optimization decisions, or instrumentation of the program to assess current optimization opportunities [AFG05, Wha01, SYN03, BV05]. A well known example of this is the Java HotSpot technology, which tracks program execution and performs JIT compilation of program sections when the executing program repeatedly executes that section above some threshold [PVC01]. Future research will adapt these techniques to the SOOPDG representation, and the efficacy of this will be verified empirically.

We also propose to investigate the utility of incorporating information regarding program structure and dependences into an intermediate form using the SOOPDG as a basis. The basic notion is to delay parallelization and JIT decisions to run time while carrying forward information pertinent to the optimization using a PDG-like representation. This will reduce the barrier between the compilation and run time environments. Deferred compilation and JIT schemes exist, but they are not predictive in nature and depend instead on run-time profiling [AFG05, VE00]. Similarly, dynamic parallelization schemes defer decisions to the run-time environment and then operate on profiling [VE99]. The representation is decorated with program dependence information and execution cost estimations to allow optimization decisions to be made in a predictive fashion prior to execution of the program segments in question, rather than operating inefficiently during a profiling phase. While this has been suggested in the past [FO98], there do not appear to be implemented examples. This scheme requires that an Information Preserving (IP) intermediate representation be developed to carry information discovered during compilation forward to the run time environment. We anticipate utilizing the SOOPDG as a basis and will refer to the information preserving form as the IPPDG. The IPPDG must be developed in conjunction with a run time compiler/interpreter that dynamically performs parallelization and/or JIT operations based on information embedded in the IPPDG, current data values, and characteristics of the current run time environment.

8.4 Extending the Simulation Dependence Graph

The SDG presented in this thesis requires further development and verification. Future work will include development of a more precise definition of rewriting

semantics and precise definition of system properties such as composability. The utility of the model will be verified through application to test simulation systems, both as a predictor of system performance and as a profiling tool intrinsic to the system run-time infrastructure. In addition, adapting the SDG concepts to more general software composability issues appears to be a fruitful line of research.

9

CONCLUSIONS

This thesis has developed and presented a new program dependence graph that represents OO languages, specifically a subset of the Java language. This representation, the SOOPDG, and its associated rewriting operations are capable of representing static program configurations, as well as computations performed by the program. An algorithm was presented to create an SOOPDG directly from program statements. This algorithm was shown to be of comparable complexity to algorithms developed for similar representations. We have also discussed the ability of the SOOPDG to represent computation through its rewriting semantics. The SOOPDG was shown to be amenable to standard analyses such as constant propagation, program slicing, and determination of class inheritance.

We have demonstrated that the SOOPDG has several advantages over existing representations. Specifically,

1. The SOOPDG represents OO constructs while introducing a relatively few number of new node types and no new edge types from those used in traditional PDGs. This results in cleaner semantics than other OO representation systems.

2. The size of the SOOPDG compares favorably with other representations, primarily due to the representation of method input/output, and also due to the use of higher order semantics.

3. The cost of building the SOOPDG is comparable to the cost of other representations, and has advantages when compared to those representations having larger sizes.

4. The SOOPDG supports dynamic binding directly through the SFU property and the use of higher order semantics. Called methods flow to the call site as "values." This simplifies reasoning regarding calling contexts and reduces graph size over that presented by Walkinshaw et al [WRW03].

5. The SOOPDG inherently supports parallel operations. This is in contrast to alternate models such as JSDG-Z and JSDG-W.

6. The SOOPDG supports the SFU property, which has advantages in semantic clarity and compiler operations over the SSA property used by other representations.

We have presented the Single Flow to Use (SFU) property that requires that, for each variable used in a program statement, exactly one value for the variable is made available to the statement at its execution. This is a new program property that is enforced during program execution, as opposed to the static representation. We have demonstrated how the SFU property is enforced through the use of strategically placed valve nodes, specified criteria for valve node placement, and demonstrated that the techniques presented by Parsons [Par92] meet those criteria.

We have also presented the SDG, a promising extension of the SOOPDG into the realm of modeling and simulation. We have discussed the ability of the SDG to represent simulation systems statically and proposed using a form of graph slicing to investigate model composability issues. We have also demonstrated the SDGs ability to represent simulation processes through graph semantics. The

SDG requires further development but appears to be a promising representation form for simulation systems.

In addition to forming the basis for the development of the SDG, the SOOPDG provides opportunities for future fruitful research in several areas. The SOOPDG may be used as a basis for run-time optimization schemes that take advantage of the information uncovered during the compilation phase. This information may be used to detect parallelization opportunities without relying on existing methods that require profiling obtained from previous program runs. The same techniques may be used to support dynamic compilation decisions, such as that used in the Java HotSpot technology.

LIST OF REFERENCES

[AFG05] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. "A Survey of Adaptive Optimization in Virtual Machines." *Proceedings of the IEEE*, **92**(2):449, 466, February 2005.

[AH03] M. Allen and S. Horwitz. "Slicing Java programs that throw and catch exceptions." *Proceedings of the ACM SIGPLAN 2003 Workshop on Partial Evaluation and Semantics Based Program Manipulation*, June 2003.

[ALS07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. AddisonWestley, Reading, Massachusetts, 2.0 edition, 2007.

[Ana99] C. Scott Ananian. *"The Static Single Information Form."*. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1999.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. "Detecting equality of Variables in programs." *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pp. 1,11, January 1988.

[BCH98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. "Practical Improvements to the Construction and Destruction of Static single Assignment Form." *Software - Practice and Experience*, pp. 859, 881, July 1998.

[BM92] Robert A. Ballance and Arthur B. Maccabe. "Program dependence graphs for the rest of us." *Technical Report 91-10*, 1992.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages." *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 257, 271, June 1990.

[BV05] Miihai Burcea and Michael Voss. "Managing Compilation Overheads in a Runtime Specializer for OpenMP." In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2005.

[CF89] Robert Cartwright and Mathis Felleison. "The semantics of program dependence." *Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 13, 27, 1989.

[CFR89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark M. Wegman, and F. Kenneth Zadeck. "An efficient method of computing static single assignment form." *Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages*, pp. 22, 35, January 1989.

[CFR91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark M. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph." *ACM Transactions on Programming Language and Systems (TOPLAS)*, **13**(4):451, 490, October 1991.

[CX01] Zhenqiang Chen and Baowen Xu. "Slicing Object-Oriented Java Programs." In *SIGPLAN Notices*, volume 36, pp. 33,40, New York, NY, 2001. ACM Press.

[CY96] Jien-Tsai Chan and Wuu Yang. "A Program Slicing System for Object-Oriented Programs." In *Proceedings of the 1996 International Computer Symposium*, pp. 422,429, December 1996.

[Den94] Jack B. Dennis. "Machines and models for parallel computing." *International Journal of Parallel Programming*, **22**(1):47, 77, 1994.

[FO98] Nickolas J. G. Falkner and Michael J. Oudshoorn. "Smarter Compilation: a step towards automated distribution." In *The Fifth IDEA Workshop*, February 1998.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and its Use in Optimization." *ACM Transactions on Programming Languages and Systems*, **9**(3):319, 349, July 1987.

[FYD06] Stephen Fink, Eran Yahav, Nurit Dor, and G. Ramalingam. "Effective typestate verification in the presence of aliasing." *ISSTA '06*, July 2006.

[GCH00] Manish Gupta, Jong-Deok Choi, and Michael Hind. "Optimizing Java Programs in the Presence of Exceptions." *Lecture Notes in Computer Science*, **1850**:422–446, 2000.

[GDD97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. "Call graph construction in object-oriented languages." In *ACM Conference on ObjectOriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1997.

[Hav93] Paul Havlak. "Construction of Thinned Gated Single-Assignment Form." *Proceedings of the Sixth International Workshop on Programming Languages and Compilers for Parallel Computing*, pp. 477 – 499, August 1993.

[Hor90] Susan Horwitz. "Identifying the Semantic and Textual differences Between Two Versions of a Program." In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 234,245, 1990.

[HR92] Susan Horwitz and Thomas Reps. "The use of program dependence graphs in software engineering." *Proceedings of the 14th International Conference on Software Engineering*, May 1992.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural slicing using dependence graphs." *ACM Transactions on Programming Languages and Systems*, **12**:26, 60, January 1990.

[HS04] Christian Hammer and Gregor Snelting. "An improved slicer for Java." In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 17–22, 2004.

[HU75] Matthew S. Hecht and Jeffrey D. Ullman. "A Simple Algorithm for Global Data Flow Analysis Problems." *SIAM Journal of Computing*, **4**(4):519 – 532, December 1975.

[Hug89] John Hughes. "Why Functional Programming Matters." *The Computer Journal*, **32**(2):98,107, 1989.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. "The Program Structure Tree: Computing Control Regions in Linear Time." In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 171, 185, 1994.

[KKP81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. "Dependence graphs and compiler optimizations." *Proceedings of the Eighth ACM Symposium on the Principles of Programming Languages*, 1981.

[KN00] Stephen Kaputis and Henry C. Ng. "Composable Simulations." In *Proceedings of the 2000 Winter Simulation Conference*, pp. 1577, 1584, December 2000.

[Kri89] E. V. Krishnamurthy. *Parallel Processing Principles and Practice*. AddisonWesley, Reading, Massachusetts, 1989.

[Kri03] Jens Krinke. "Context-Sensitive Slicing of Concurrent Programs.", 2003.

[Lau06] Kung-Kiu Lau. "Software Component Models." In *ACM Proceedings of the 28th International conference on Software Engineering*, 2006.

[LCH04] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. "A Compiler Framework for Speculative Optimizations." *ACM Transactions on Architecture and Code Optimization*, **1**(3):247, 271, 2004.

[LH96] Hans-Wolfgang Loidl and Kervin Hammond. "Making a packet: cost-effective communication for a parallel graph reducer." *Implementation of Functional Languages*, 1996.

[LH98] D. Liang and M. Harrold. "Slicing Object Using System Dependence Graph.", 1998.

[LMP99] Jaejin Lee, Samual P. Midkiff, and David A. Padua. "Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs." *Proceedings of the Seventh ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1999.

[Mic05] Sun Microsystems. *Enterprise Java Beans Specification, Version 3.0*. 2005.

[MMK94] B. A. Malloy, J. D. Mcgregor, A. Krishnaswamy, and M. Medlkonda. "An extensible program representation for object-oriented software." *ACM SIGPLAN Notices*, **29**(12):38–47, 1994.

[Mog91] Eugenio Moggi. "Notions of Computation and Monads." *Information and Computation*, **93**(1):55–92, 1991.

[Nan01] Mangala Gowri Nanda. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, India Institute of Technology, 2001.

[OH02] Michael Oudshoorn and Lin Huang. "Evolving toward an Optimal Scheduling Solution Through Adaptivity." *Journal of Parallel and distributed Computing*, **62**(7):1203, 1222, July 2002.

[OO84] K. Ottenstein and L. Ottenstein. "The Program Dependence Graph in a Software Development Environment." In *Proceedings of the ACM SIGSOFT SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 177, 184, 1984.

[Par92] Rebecca Parsons. *Semantic Program Dependence Graphs*. PhD thesis, Rice University, June 1992.

[Par03] Rebecca Parsons. "Components and the World of Chaos." *IEE Software*, pp. 83, 85, May/June 2003.

[PP96] Paul M. Peterson and David A. Padua. "Static and dynamic evaluation of data dependence analysis techniques." *IEEE Transactions on Parallel and Distributed Systems*, **7**(11):1121,1132, November 1996.

[PVC01] M. Paleczny, C. Vick, and C. Click. "The Java hotspot server compiler." In *in Proc. Usenix Java Virtual Machine Research and Technology Symp. (JVM01)*, pp. 1–12, 2001.

[PW03a] Mikel D. Petty and Eric W. Weisal. "A Composibility Lexicon." In *Proceedings of the Spring 2003 Software Interoperability Workshop*, pp. 181, 187, April 2003.

[PW03b] Mikel D. Petty and Eric W. Weisal. "A Formal Approach to Composibility." In *Proceedings of the 2003 Interservice/Industry Training, Simulation and Education Conference*, pp. 1763, 1772, December 2003.

[Rei78] John H. Reif. "Symbolic Program analysis in Almost Linear Time." *Conference Record of the Fifth ACM Symposium on the Principles of Programming Languages*, 1978.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, New York, New York, 1998.

[RKG04] Atanas Rountev, Scott Kagan, and Michael Gibas. "Static and dynamic analysis of call chains in java." In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1, 11, July 2004.

[RSE04] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. "Compiler Orchestrated Prefetching via Speculation and Predication." In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, pp. 189, 198, October 2004.

[RWF03] Jeffery von Ronne, Ning Wang, and Michael Franz. "Interpreting Programs in Static Single Assignment Form." Technical Report TR 03-19, University of California, Irvine, Irvine, CA, April 2003.

[Sel89] Rebecca Parsons Selke. "A rewriting semantics for program dependence graphs." *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pp. 12, 24, August 1989.

[Sel90a] Rebecca Parsons Selke. "Program dependence graphs: a formal treatment." Technical Report TR90-130, Rice University, Houston, TX, August 1990.

[Sel90b] Rebecca Parsons Selke. "Transforming program dependence graphs." Technical Report TR90-131, Rice University, Houston, TX, August 1990.

[SG95] Vugranam Sreedhar and Guang R. Gao. "A Linear time algorithm for placing phi-nodes." *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pp. 62–73, 1995.

[SHW93] Harini Srinivasan, James Hook, and Michael Wolfe. "Static Single Assignment for Explicitly Parallel Programs." *Conference record of the 20th Annual Symposium on Principles of Programming Languages*, pp. 260–272, 1993.

[SYN03] T. Suganuma, T. Yasue, and T. Nakatani. "A region-based compilation technique for a Java just-in-time compiler." *ACM SIGPLANNotices*, **38**(5):312, 323, May 2003.

[TGH92] Kevin B. Theobold, Guang R. Gao, and Laurie J. Hendron. "On the Limits of Program Paralellism and its Smoothness." Technical Report ACAPS Technical Memo 40, McGill University School of Computer Science, 1992.

[Tip95] F. Tip. "A survey of program slicing techniques." *Journal of programming languages*, **3**:121–189, 1995.

[Tre79] Philip C. Treleaven. "Exploiting Program concurrency in computing systems." *Computer*, pp. 42, 49, January 1979.

[VE99] Michael Voss and Rudolf Eigenmann. "Dynamically Adaptive Parallel Programs." In *ISHPC*, pp. 109–120, 1999.

[VE00] Michael Voss and Rudolf Eigenmann. "ADAPT: Automated De-Coupled Adaptive Program Transformation." In *International Conference on Parallel Processing*, pp. 163–, 2000.

[VE01] Michael J. Voss and Rudolf Eigemann. "High-level adaptive program optimization with ADAPT." *ACM SIGPLAN Notices*, **36**(7):93–102, 2001.

[WGH03] Gwendolyn Walton, Brian Goldiez, Ron Hofer, and David Kaup. "Mathematical Foundations for Modeling and Simulation." *Proceedings of the SPIE*, **5091**:310, 320, 2003.

[Wha01] J. Whaley. "Partial method compilation using dynamic profile information." *ACM SIGPLANNotices*, **36**(11):166, 179, November 2001.

[WRW03] Neil Walkinshaw, Marc Roper, and Murray Wood. "The Java system dependence graph." *Third IEEE International Workshop on Source Code Analysis and Manipulation*, September 2003.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, New York, 1991.

[Zha98] Jinjian Zhao. "Multithreaded Dependence Graphs for Concurrent Java Programs." *Proceedings of Workshop on Software Engineering and Database Systems, International computer Symposium*, 1998.

[Zha99] Jinjian Zhao. "Applying Program Dependence Analysis to Java Software." *Proceedings of the International Symposium on software Engineering for Parallel and Distributed Systems*, 1999.