DATAFLOW ANALYSIS OF SOFTWARE FRAGMENTS

BY ATANAS ROUNTEV

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

New Brunswick, New Jersey

August, 2002

ABSTRACT OF THE DISSERTATION

Dataflow Analysis of Software Fragments

by ATANAS ROUNTEV

Dissertation Director: Barbara Gershon Ryder

The goal of *dataflow analysis* is to determine properties of the run-time behavior of software systems by analyzing the software source code. Such analysis has a wide range of uses in software engineering tasks and their supporting tools. Traditionally, interprocedural dataflow analyses are designed as *whole-program analyses*: they take as input and process complete programs. This analysis model has been used extensively in the previous work on dataflow analysis.

The paradigm of whole-program analysis has several limitations. Such analysis cannot be applied to incomplete programs (e.g., library modules), to programs containing unanalyzable modules (e.g., programs built with precompiled components), and to large programs. These restrictions limit the usefulness of whole-program analysis in the context of real-world software development. To address this problem, in this thesis we propose the paradigm of fragment dataflow analysis.

Fragment dataflow analysis is an interprocedural dataflow analysis that is designed to analyze software fragments rather than complete programs. This analysis model is more flexible and can be used in situations in which whole-program analysis is not applicable. The first contribution of our work is a theoretical framework for constructing fragment dataflow analyses. This framework provides analysis designers with a sound basis for constructing fragment analyses and for reasoning about their properties.

Our second contribution is an approach for performing points-to analysis and side-effect analysis for C programs that are built with precompiled libraries modules. We define and evaluate fragment analyses that can be applied to library modules, as well as to the corresponding client modules. Our work enables the use of two fundamental dataflow analyses—points-to analysis and side-effect analysis—for C programs that cannot be handled by the previous work on whole-program analysis.

The third contribution of this work is an approach for performing class analysis for the purposes of testing of polymorphism in Java. Our work defines a method for constructing fragment class analyses in order to compute test coverage requirements. We present empirical results showing the precision and practicality of the analyses. This work is the first one to show how to construct high-quality coverage tools for testing of polymorphism in Java software.

Acknowledgements

I would like to thank my advisor and mentor Prof. Barbara Ryder for her constant support and guidance. She was always ready to help, and she taught me many things that influenced my professional and personal growth. I would also like to thank all members of the PROLANGS group for providing a great research environment. Dr. Bill Landi taught me many things, and our card games were a lot of fun. Prof. Tom Marlowe provided constant encouragement and many insightful comments. I am grateful to Prof. Phil Stocks for our long conversations and for the good times. I spent many hours talking with Matt Arnold about research and about many other things; we learned a lot from each other. I am also grateful to Ana Milanova for her patience and for all the discussions. Dr. Satish Chandra helped me broaden my research interests and explore new ideas. Finally, I would like to thank my parents for their unconditional support and encouragement.

Dedication

To my parents

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of *dataflow analysis* is to determine properties of the run-time behavior of software systems by analyzing the software source code. Dataflow analysis information has been traditionally used to enable a wide variety of compiler optimizations. In addition, this information has many possible software engineering applications. For example, dataflow analysis can be used during software maintenance to aid in understanding the analyzed software, and to assist in performing and evaluating software changes. Dataflow analysis can also check correctness properties in order to discover software faults. Analysis information can be used to define test coverage criteria and to evaluate the adequacy of test suites.

Interprocedural dataflow analyses are designed to analyze multiple procedures and to take into account the interactions among these procedures. Traditionally, interprocedural dataflow analyses are designed as whole-program analyses. A *whole-program analysis* takes as input a complete program and analyzes that entire program. The advantage of processing the entire program is that the analysis can track complex interactions among different parts of the program. This approach enables a wide variety of advanced analyses, and has been used extensively in the existing work on dataflow analysis.

Unfortunately, the paradigm of whole-program analysis has several limitations. First, such analysis cannot be applied to *incomplete programs*. For example, whole-program analysis cannot be used when a reusable component has to be analyzed in the absence of that component's clients. As another example, such analysis is not applicable when program components only become available at run time (e.g., through mechanisms such as dynamic class loading or run-time component discovery). Another problem for whole-program analysis comes from program components that are available, but *cannot be analyzed*. Typical examples are components for which source code is not available (e.g., precompiled libraries); other examples are components written in a different language than the rest of the program. Finally, analyzing the entire program limits the *scalability* of whole-program analysis for large programs. This problem is particularly relevant for more precise analyses, which tend to be relatively expensive, and therefore cannot be used for large programs.

The fundamental problem with whole-program analysis is that this analysis paradigm is too restrictive to accommodate a wide variety of real-world situations. Thus, the large body of existing work on whole-program analysis cannot be used directly to solve many important analysis problems in the context of optimizing compilers and software engineering tools. The real-world impact of dataflow analysis research can be broadened significantly if the limitations of whole-program analysis are addressed and resolved. To achieve this goal, in this thesis we propose the paradigm of fragment dataflow analysis.

Fragment dataflow analysis is an interprocedural dataflow analysis that analyzes software fragments rather than complete programs. The input to the fragment analysis is a fragment which consists of some arbitrary set of procedures that do not form a complete program. The analysis output is information about properties of the possible run-time behaviors of the fragment, based on the available knowledge about the environment in which the fragment will operate.

The conceptual difference between whole-program analysis and fragment analysis is that they are designed to solve different problems. Essentially, fragment analysis solves a problem that is more general than the problem solved by whole-program analysis. As a result, the model of fragment analysis is more flexible and can be used in situations in which whole-program analysis is not applicable. For example, fragment analysis can be used to analyze a software component in the absence of that component's clients and servers; in this case, the component is the analyzed fragment. In general, many important analysis problems can be solved using fragment dataflow analysis, even though they cannot be solved with the traditional model of whole-program analysis.

1.1 Contributions

The work presented in this thesis has three major contributions.

1.1.1 Theoretical Model of Fragment Analysis

The first contribution of our work is a theoretical framework for constructing fragment dataflow analyses. This framework provides the basis for designing fragment analyses and for reasoning about their properties. The key idea of our approach is to *derive* fragment analyses from existing whole-program analyses. This allows conceptual reuse of existing algorithmic techniques for whole-program analysis, and also enables physical reuse of existing implementations of whole-program analyses. The approach can be applied to a variety of flow-sensitive/flow-insensitive and context-sensitive/context-insensitive analyses.[1] Using our framework, analysis designers can adapt the large body of existing work on whole-program analysis to solve many problems that currently cannot be solved with whole-program analysis.

1.1.2 Points-to Analysis and Side-effect Analysis for C Programs Built with Precompiled Libraries

Our second contribution is an approach for performing points-to analysis and side-effect analysis for C programs that are built with precompiled library modules. *Points-to analysis* determines which memory locations may be pointed to by a given variable v (i.e., which addresses may be stored in v). *Side-effect analysis* determines which variables are potentially modified by the execution of a program statement. In the presence of pointers, side-effect analysis requires the output of a points-to analysis to disambiguate modifications through pointer dereferences. Both points-to analysis and side-effect analysis compute information that is of fundamental importance for many analyses and optimizations for software written in languages that use pointers (such as C) or object references (such as Java). In this thesis we consider the problem of performing these two analyses for C programs that are built with precompiled library modules. For such programs, traditional methods for

[1]Definitions of flow sensitivity and context sensitivity are presented in Chapter 2.

whole-program analysis cannot be applied because the source code for the entire program is never available.

In the context of this problem, we have developed an approach for performing flow-insensitive context-insensitive points-to analysis as well as side-effect analysis based on such points-to analysis. We first present a method for constructing *fragment analyses of a library module*; these analyses can be used without any available information about the clients of the library. We also show how to perform *fragment analyses of a client module* without having the source code of the used library modules; our method is based on a technique for automatic construction of summary information that precisely describes the possible effects of library modules. Finally, we present *empirical results* that confirm the practicality and effectiveness of our approach. This work enables optimizing compilers and software engineering tools to use two fundamental dataflow analyses—points-to analysis and side-effect analysis—for C programs that are built with precompiled library modules. For such programs, it is not possible to use the previous work on whole-program points-to analysis and side-effect analysis.

1.1.3 Class Analysis for Testing of Polymorphism in Java Software

The third contribution of our work is an approach for performing class analysis for the purposes of testing of polymorphism in Java. *Class analysis* is a fundamental dataflow analysis for object-oriented software that determines the possible classes of all objects that a reference variable may refer to. This information has a wide variety of applications in optimizing compilers and software engineering tools. We consider one specific use of class analysis in the context of a coverage tool for testing of polymorphism in Java software. *Polymorphism* is a basic object-oriented feature that allows the binding of an object reference to objects of different classes. Previous work on testing of object-oriented software [41, 40, 43, 11] proposes techniques that require exercising of all possible bindings of polymorphic references at call sites. For example, if a reference variable v may refer to instances of classes A, B, and C, adequate testing of call site "$v.m()$" should exercise each one of the three possible classes of the receiver object. However, existing work does not address the problem of measuring how well a given test suite covers all possible polymorphic bindings.

We have built a coverage tool for Java that reports which bindings of polymorphic receivers have been exercised by a given test suite. The tool uses class analysis to determine the coverage requirements (i.e., all possible bindings of reference variables). In this context we cannot use whole-program class analysis because the tool may be used for testing of partial programs. We have developed *a general method for constructing fragment class analyses* from existing whole-program class analyses. This method allows a large number of whole-program analyses from previous work to be incorporated in coverage tools for testing of polymorphism. We also present *empirical results* that compare several specific fragment class analyses and evaluate their suitability for our target problem. The results clearly show that appropriately chosen analyses have high precision and low cost, and therefore are good candidates for inclusion in real-world tools. Our work is the first one to show how to construct high-quality coverage tools for testing of polymorphism in Java software.

1.2 Thesis Organization

The rest of this thesis is organized as follows. The theoretical framework for constructing fragment dataflow analyses is presented in Chapter 2. Chapter 3 discusses our approach for performing points-to analysis and side-effect analysis for C programs that are built with precompiled libraries. Chapter 4 presents our method for constructing fragment class analyses for the purposes of testing of polymorphism in Java. Related work is discussed in Chapter 5. Finally, Chapter 6 summarizes the thesis and presents possible directions for future work.

Chapter 2

Theoretical Foundations

This chapter presents a theoretical framework for constructing fragment dataflow analyses from existing whole-program dataflow analyses. Our approach can be applied to a variety of flow-sensitive/flow-insensitive and context-sensitive/context-insensitive analyses. In the remainder of this thesis we present and evaluate specific fragment analyses that were designed with the help of these general theoretical techniques: Chapter 3 discusses flow-insensitive context-insensitive fragment points-to analyses for C and fragment side-effects analyses based on them, and Chapter 4 presents flow-insensitive fragment class analyses for Java. (The notions of flow sensitivity and context sensitivity will be discussed shortly.)

2.1 Whole-program Dataflow Analysis

This section presents a standard framework for whole-program interprocedural dataflow analysis [57, 39]. This framework serves as the basis for our general approach for constructing fragment dataflow analyses. Given a whole program to be analyzed, a *whole-program analysis* conceptually defines a tuple $<G, L, F, M, \eta>$, where

- $G = (N, E, \rho)$ is a directed graph with node set N, edge set E, and starting node $\rho \in N$. G is an interprocedural control flow graph, as described below.
- $<L, \leq, \wedge, \top>$ is a meet semi-lattice [39] with partial order $\leq$, meet operation $\wedge$, and greatest element $\top$. For the purposes of our work, we assume that L is finite.[1]
- $F \subseteq \{f \mid f : L \to L\}$ is a monotone function space (i.e., $x \leq y$ implies $f(x) \leq f(y)$)

[1]Our results can be trivially extended for infinite semi-lattices with finite height.

- $M : N \to F$ is an assignment of *transfer functions* to the nodes in G. The transfer function for node n will be denoted by f_n.

- $\eta \in L$ is the solution at the bottom of the starting node ρ

A *partially ordered set* is a set that has an ordering relation $\leq$ that is reflexive, anti-symmetric, and transitive. For any two elements x and y of a partially ordered set L, their *meet* (denoted by $x \wedge y$) is a unique element of L which is less than x and y in the partial order, and is maximal among all elements of L with this property; in general, $x \wedge y$ may not exist. A *meet semi-lattice* is a partially ordered set in which every two elements have a meet.

The program is represented by an *interprocedural control flow graph* (ICFG) which contains the control flow graphs for all procedures in the program. Each procedure has a single *entry node* and a single *exit node.* Node ρ is the entry node of the starting procedure. Nodes within each procedure represent statements from that procedure, and the edges in the graph represent potential flow of control between these statements. Each call statement is represented by a pair of nodes, a *call node* and a *return node.* There is an edge from the call node to the entry node of the called procedure; there is also an edge from the exit node of the called procedure to the return node in the calling procedure.

A path from node n_1 to node n_k is a sequence of nodes $p = (n_1, \ldots, n_k)$ such that $(n_i, n_{i+1}) \in E$. Let $f_p = f_{n_1} \circ f_{n_2} \circ \ldots \circ f_{n_k}$. A *realizable path* is a path on which every procedure returns to the call site which invoked it [57, 35, 49]; only such paths represent potential sequences of execution steps.[2] A *same-level realizable path* is a realizable path whose first and last nodes belong to the same procedure, and on which the number of call nodes is equal to the number of return nodes. Such paths represent sequences of execution steps during which the call stack may temporarily grow deeper, but never shallower that its original depth, before eventually returning to its original depth [49]. The set of all realizable paths from n to m will be denoted by $RP(n, m)$; the set of all same-level realizable paths

[2]Some realizable paths may be infeasible (i.e., cannot be followed during an actual program execution); a typical example is a program in which the boolean conditions of two different `if` statements always have the same value. In general, it is impossible to detect such infeasible paths, and most analyses do not attempt to filter out realizable paths that are infeasible.

from n to m will be denoted by $SLRP(n,m)$.

Definition 1 *For each $n \in N$, the **meet-over-all-realizable-paths** (MORP) solution at n is defined as $MORP(n) = \bigwedge_{p \in RP(\rho,n)} f_p(\eta)$.*

After conceptually defining the tuple $<G, L, F, M, \eta>$ based on the input program, a whole-program analysis computes a dataflow solution $S : N \to L$; the solution at the bottom of node n will be denoted by S_n. The solution is *safe* iff $S_n \leq MORP(n)$ for each node $n \in N$. A *safe analysis* computes a safe solution for each valid input program.

The analysis computes a solution by using some solution procedure, which is essentially an algorithm that takes as input the tuple $<G, L, F, M, \eta>$ and produces as output a solution $S : N \to L$. In the rest of the thesis we will refer to this algorithm as the *solution engine* of the analysis. For efficiency reasons, the solution engine typically employs approximation techniques that result in a solution that is less precise than the MORP solution (i.e., $S_n < MORP(n)$). With respect to these approximations, two major design choices are the *flow sensitivity* and the *context sensitivity* of the analysis. Informally, flow-sensitive analyses take into account the flow of control in the program, while flow-insensitive analyses ignore that flow. Context-sensitive analyses distinguish among the different invocation contexts of a procedure, and context-insensitive analyses do not make this distinction.

2.1.1 Context-Insensitive Analysis

The solution engine of a *flow-sensitive, context-insensitive* analysis constructs a system of equations of the form

$$S_\rho = \eta, \quad S_n = \bigwedge_{m \in Pred(n)} f_n(S_m) \tag{2.1}$$

where $Pred(n)$ is the set of predecessor nodes for n. This system has the form $S = H(S)$, where $S : N \to L$ and $H : (N \to L) \to (N \to L)$. To solve this system, the engine starts with an initial solution S^0 for which $S^0_\rho = \eta$ and $S^0_n = \top$ for $n \neq \rho$. The final solution is the limit of the sequence $\{S^i | S^i = H(S^{i-1})\}$, and can be computed using some form of fixed-point iteration.[3]

The solution engine of a *flow-insensitive, context-insensitive analysis* ignores the flow of control in the program (i.e., it assumes that statements may be executed in arbitrary order), and computes a single solution $S \in L$ for the entire program.[4] Intuitively, this approach can be thought of as merging all nodes in G. Conceptually, this creates a graph with a single node, with a corresponding transfer function $f_{all} : L \rightarrow L$ of the form $f_{all} = (\bigwedge_{n \in N} f_n) \wedge id$. This transfer function summarizes all possible effects of all transfer functions for individual program statements. The inclusion of the identity function id in the above definition is necessary to encode the fact that the analysis does not perform any "kills"—that is, because of the flow insensitivity, dataflow facts that have already been created cannot be removed. After all nodes have been merged, the solution engine defines the equation $S = f_{all}(S)$ and then solves it using fixed-point iteration as follows: $S^0 = \eta, S^i = f_{all}(S^{i-1})$.

2.1.2 Context-Sensitive Analysis

The problem with context-insensitive analysis is that the effects of different invocations of the same procedure are merged and are propagated to all callers of that procedure. Context-sensitive analysis attempts to avoid this potential source of imprecision by distinguishing between the different calling contexts of a procedure. There have been various approaches that introduce different elements of context sensitivity in dataflow analysis. For example, early work by Sharir and Pnueli [57] presents two approaches to context sensitivity: "functional approach" and "call string approach". In the functional approach, the calling context is approximated by a lattice element that encodes the state of the program at the entry node of the called procedure. The call string approach approximates calling context by a chain of k enclosing call sites. Both techniques have served as the basis for a large number of subsequent context-sensitive analyses.

[3]Since N and L are finite and H is monotone, this sequence has a well-defined limit.

[4]Some analyses (e.g., [33]) compute separate solutions for each procedure; our results can be trivially extended to such analyses. Whenever we discuss flow- and context-insensitive analyses in the rest of the thesis, we refer to analyses that compute a single solution for the entire program.

2.2 Fragment Dataflow Analysis

This section describes the overall structure of fragment dataflow analysis and its relationship with whole-program analysis. The techniques presented in this section allow designers of dataflow analyses to *derive* fragment analyses from existing whole-program analyses. Using these techniques, the large existing body of work on whole-program analysis can be adapted to solve a variety of problems that cannot be solved with the traditional model of whole-program analysis.

2.2.1 Structure of Fragment Analysis

The input to a fragment analysis contains the source code for some *software fragment* Fr. For the purposes of this discussion, we will assume that the fragment is some arbitrary set of procedures that does not form a complete program. In general, the fragment could potentially be a part of many different whole programs. The input to the fragment analysis also contains *whole-program information* I that represents the available knowledge about the whole programs that could contain Fr. The available whole-program information depends on the particular context in which the fragment analysis is performed. For example, I could encode some knowledge about preconditions that will be satisfied by the callers of the fragment. As another example, I could be summary information about the possible effects of the callees of the fragment. We will use $\mathit{WholePrg}(Fr, I)$ to denote the set of all valid whole programs that contain Fr and for which I is true; typically, $\mathit{WholePrg}(Fr, I)$ is an infinite set.

Interprocedural Control Flow Graph

Consider a fragment Fr and the control flow graphs of the procedures in that fragment. We will denote by *BoundaryEntries* the set of entry nodes for all procedures in Fr that could be called by non-fragment procedures. More precisely, *BoundaryEntries* contains all entry nodes e such that for some whole program from $\mathit{WholePrg}(Fr, I)$, node e has a predecessor call node that is located inside a non-fragment procedure. Similarly, set *BoundaryCalls* contains all call nodes c in the fragment such that for some whole program from $\mathit{WholePrg}(\mathit{Fr}, I)$, node c has a successor entry node that is outside of the fragment.

Given Fr and I, the fragment analysis starts by creating an interprocedural control flow graph $G' = (N', E', \rho')$. G' contains the control flow graphs of all fragment procedures, as well as the interprocedural edges representing calls between fragment procedures (as described in Section 2.1). In addition, G' contains the control flow graphs for several *placeholder procedures*. Intuitively, the placeholder procedures serve as representatives for the unknown procedures from the rest of the program. For each placeholder procedure, there is a control flow graph that contains *placeholder nodes* and edges between these nodes. Informally, the role of the placeholder nodes is to represent the possible effects of the unknown statements from the rest of the program; a more formal description will be presented shortly.

In addition to the intraprocedural edges inside the placeholder procedures, there could be interprocedural edges representing (i) calls between placeholder procedures, and (ii) calls between placeholder procedures and fragment procedures. For each fragment node $e \in \mathit{BoundaryEntries}$, G' should contain at least one interprocedural edge from a call node in some placeholder procedure to entry node e (plus the corresponding exit-return edge). Similarly, for each fragment node $c \in \mathit{BoundaryCalls}$, G' should contain an interprocedural edge from c to the entry node of some placeholder procedure (plus the corresponding exit-return edge).

If the fragment contains the starting procedure of the program, the starting node ρ' is the entry node of this fragment procedure. Otherwise, ρ' is the entry node of some placeholder procedure.

Example. In Chapter 3 we consider the problem of analyzing a library before we have any information about the possible clients of this library. In the context of this problem, we use a single placeholder procedure `ph_proc` that represents the unknown procedures from client code that could be built on top of the library. Inside this placeholder procedure, we use a variety of placeholder nodes that represent the possible effects of different categories of statements from the unknown client code. The starting node ρ' is the entry node of `ph_proc`. $\diamondsuit$

Lattice and Transfer Functions

Given Fr and I, the fragment analysis defines a finite meet semi-lattice L' with partial order $\leq$, meet operation $\wedge$, and greatest element $\top'$. In addition, the analysis defines a monotone function space $F' \subseteq \{f \mid f : L' \rightarrow L'\}$ and assigns transfer functions to the nodes in G' by defining a map $M' : N' \rightarrow F'$. Note that N' contains not only nodes inside the fragment, but also placeholder nodes inside placeholder procedures. The transfer functions associated with these placeholder nodes encode the possible effects of unknown statements from the rest of the program. Finally, the analysis also defines a value $\eta' \in L'$ as the solution at the bottom of the entry node ρ'.

Computing a Solution

Once the fragment analysis has defined the tuple $< G', L', F', M', \eta' >$, it computes a dataflow solution $S' : N' \rightarrow L'$; the solution at the bottom of node n will be denoted by S'_n. To compute this solution, the fragment analysis can use solution procedures similar to the approaches used by whole-program analyses. In fact, we are only interested in fragment analyses that have been *derived* from existing whole-program analyses. For such fragment analyses, we assume that the designers of the analysis have chosen L' and F' that are similar in structure to L and F from some whole-program analysis (or from an entire category of whole-program analyses), and because of this, the solution engine of that whole-program analysis can be reused by the fragment analysis. Therefore, after conceptually defining $< G', L', F', M', \eta' >$, the fragment analysis will run *the solution engine of some existing whole-program analysis.* From the point of view of that engine, it is irrelevant that tuple $<G', L', F', M', \eta'>$ is not based on some whole program, but instead is constructed by the fragment analysis. As long as the tuple has the appropriate structure and properties, the whole-program solution engine can be used to compute a fragment analysis solution (the correctness of this solution is discussed in Section 2.2.3). This approach allows conceptual reuse of existing algorithmic techniques for whole-program analysis, and also enables physical reuse of existing implementations of whole-program analyses.

2.2.2 Fragment Analysis Design

The previous section describes the general structure of fragment analysis, but intentionally does not discuss several issues that have to be addressed by the designer of a fragment analysis. The first issue is what kind of whole-program information I to use. This depends on the context in which the fragment analysis is used—that is, the process/tool/system that employs the analysis. Clearly, we cannot expect to be able to provide a general characterization of I. In the rest of this thesis, we present several specific problems in the context of which we define the appropriate form of whole-program information. For example, we consider the problem of analyzing software libraries before we have access to the client components of these libraries. In the context of this problem, the only available information is the list of library procedures that could be called by client code, as well as the list of library variables that could be read/written by client code. As another example, we also consider the problem of analyzing a library client without having access to the library source code. For this problem, one possible solution is to use an I that describes the potential effects of the library code on the client.

Another issue is what kind of lattice and transfer functions to use. As described earlier, our goal is to be able to reuse existing solution procedures from whole-program analyses, and therefore the structure of L' and F' should allow this reuse. From our experience, given an existing whole-program analysis with L and F, it is trivial to define L' and F' of the same form as L and F, thus enabling the use of the solution engine from the whole-program analysis. The only non-trivial issue is how to define the parts of L' and F' that represent entities from outside the fragment (e.g., what kind of transfer functions to associate with placeholder nodes). In general, this depends on the available whole-program information I and on the kind of whole-program analysis we start with. In the next section we present techniques that can be used by analysis designers to ensure that they in fact have chosen L' and F' appropriately. In the remaining chapters of the thesis we present specific choices of lattices and transfer functions in the context of particular fragment analyses.

2.2.3 Fragment Analysis Safety

Consider a safe whole-program analysis from which we have derived a fragment analysis. To be able to reason about the safety of the fragment analysis, analysis designers have to define the relationship between the semi-lattice L' in the fragment analysis and all semi-lattices L_P constructed by the corresponding whole-program analysis for the programs $P \in \mathit{WholePrg}(\mathit{Fr}, I)$. For each such L_P, analysis designers need to define an *abstraction relation* $\alpha_P \subseteq L_P \times L'$. This relation encodes the notion of safety and is used to prove that the analysis is safe; it is never explicitly constructed or used during the actual analysis. If $(x, x') \in \alpha_P$, we will write $\alpha_P(x, x')$.

Intuitively, the abstraction relation α_P defines the relationship between the knowledge represented by values from L_P and the knowledge represented by values from L'. If $\alpha_P(x, x')$, the knowledge associated with x' "safely abstracts" the knowledge associated with x; thus, α_P is similar in nature to the abstraction relations used in abstract interpretation [15]. Relation α_P is defined by the designer of the fragment analysis, and depends on the original whole-program analysis, on the available information about the rest of the program, and on the intended uses of the fragment analysis solution. In the remaining chapters of this work we present specific definitions of abstraction relations in the context of several fragment analyses.

Definition 2 *A solution produced by a fragment analysis for an input pair* (Fr, I) *is* **safe** *iff* $\alpha_P(\mathit{MORP}_P(n), S'_n)$ *for each* $P \in \mathit{WholePrg}(\mathit{Fr}, I)$ *and each node* $n \in N'$ *inside the fragment. Here* S'_n *denotes the fragment analysis solution at* n*, and* $\mathit{MORP}_P(n)$ *is the MORP solution at* n *for the whole-program analysis of* P*, as defined earlier. A* **safe fragment analysis** *produces a safe solution for each valid input pair* (Fr, I)*.*

2.2.4 General Correctness Conditions

We have defined a set of sufficient conditions that ensure the safety of a fragment analysis according to the above definition. These conditions can be used by analysis designers in two ways. First, after a designer has defined her analysis, she can prove its safety. Second, while designing her fragment analysis, she can use the conditions as a guide to define different elements of the analysis, in particular the lattice and the transfer functions. While designing the specific fragment analyses presented later in the thesis, we found it very helpful to use the correctness conditions as a guide in making design choices.

Intuitively, the first condition ensures that a safe approximation in L', as defined by the partial order $\leq$, is also a safe abstraction according to α_P. The second condition ensures that if $x' \in L'$ safely abstracts the effects of each individual realizable path ending at a node n, it also safely abstract the MORP solution at n. Formally, for any $P \in \mathit{WholePrg}(\mathit{Fr}, I)$ and its α_P, any $x, y \in L_P$ and any $x', y' \in L'$, the following should be true:

Condition 1: If $\alpha_P(x, x')$ and $y' \leq x'$, then $\alpha_P(x, y')$

Condition 2: If $\alpha_P(x, x')$ and $\alpha_P(y, x')$, then $\alpha_P(x \wedge y, x')$

The next condition ensures that the solution η' at the starting node in the fragment analysis safely abstracts the solution η_P at the starting node in the whole-program analysis. That is, for any program $P \in \mathit{WholePrg}(\mathit{Fr}, I)$ the following should be true:

Condition 3: $\alpha_P(\eta_P, \eta')$

We should ensure that the transfer functions for fragment nodes in the fragment analysis safely abstract the corresponding transfer functions in the whole-program analysis. Intuitively, this means that the semantics associated with node $n \in \mathit{Fr}$ in the fragment analysis safely models the semantics associated with n in the whole-program analysis. We first introduce the following notation: for any function f in the function space of the whole-program analysis, and any function $f' \in F'$, $\alpha_P(f, f')$ holds if and only if $\alpha_P(x, x')$ implies $\alpha_P(f(x), f'(x'))$ for any $x \in L_P$ and $x' \in L'$. In other words, we generalize the abstraction relation to transfer functions.

Let $f_{n,P}$ be the transfer function for node $n \in \mathit{Fr}$ in the whole-program analysis for some program $P \in \mathit{WholePrg}(\mathit{Fr}, I)$, and f'_n be the transfer function for that same node in the fragment analysis. For any such n, the following should be true:

Condition 4: $\alpha_P(f_{n,P}\ , f'_n)$

The next two conditions guarantee that the possible effects of nodes outside of the fragment are taken into account by the fragment analysis. First, we need to consider realizable paths that represent calls from the rest of the program to the fragment. For any $P \in \mathit{WholePrg}(Fr, I)$ and any node $n \in \mathit{BoundaryEntries}$, let $\mathit{OutRP}_P(\rho, n)$ be the set of all realizable paths $q = (\rho, \ldots, m, n)$ in the ICFG of P such that $m \notin Fr$. Then the following condition should hold:

Condition 5: For any path $q \in \mathit{OutRP}_P(\rho, n)$ there exists a realizable path $q' = (\rho', \ldots, n)$ in G' such that $\alpha_P(f_q, f'_{q'})$

Finally, we also have to consider the effects of calls from the fragment to the rest of the program. For any $P \in \mathit{WholePrg}(Fr, I)$ and any node $n \in \mathit{BoundaryCalls}$ with a corresponding return node r, let $\mathit{OutSLRP}_P(n, r)$ be the set of all same-level realizable paths $q = (n, m, \ldots, r)$ in the ICFG of P such that $m \notin Fr$. Each such path represents the possible effects of a call from the fragment to some procedure that is external to the fragment. In this case, the following should hold:

Condition 6: For any path $q \in \mathit{OutSLRP}_P(n, r)$ there exists a same-level realizable path $q' = (n, \ldots, r)$ in G' such that $\alpha_P(f_q, f'_{q'})$

The safety of the fragment analysis is guaranteed by the following claim:

Theorem 1 *If the fragment analysis uses the solution engine of a safe whole-program analysis, and if Conditions 1–6 hold, then the resulting fragment analysis solution is safe according to Definition 2.*

Proof Outline. Because of the safety of the whole-program solution engine, the solution S'_n computed by the fragment analysis for a node n is an approximation of $\mathit{MORP}'(n) = \bigwedge_{q' \in RP(\rho', n)} f_{q'}(\eta')$, where $RP(\rho', n)$ is the set of realizable paths $(\rho', \ldots, n)$ in G'. In other words, $S'_n \leq \mathit{MORP}'(n)$.

Suppose we can show that $\alpha_P(\mathit{MORP}_P(n), \mathit{MORP}'(n))$ for each node $n \in Fr$. Because of Condition 1, this would imply that $\alpha_P(\mathit{MORP}_P(n), S'_n)$, which guarantees the safety of the fragment analysis.

To show that $\alpha_P(MORP_P(n), MORP'(n))$, it is enough to prove that for each path $q \in RP(\rho, n)$ in the ICFG for P, there exists a path $q' \in RP(\rho', n)$ in G' such that $\alpha_P(f_q, f'_{q'})$. Because of Condition 3, this implies $\alpha_P(f_q(\eta_P), f'_{q'}(\eta'))$; in turn, using Conditions 1 and 2, it is trivial to show that $\alpha_P(MORP_P(n), MORP'(n))$. The existence of such q' for each q can be proven by induction on the number k of fragment nodes in q. The base case of the induction ($k = 1$) holds due to Condition 5; the inductive step can be shown using Conditions 4–6. $\diamondsuit$

2.2.5 Relaxed Correctness Conditions

Conditions 5 and 6 from Section 2.2.4 are designed to ensure that the fragment analysis correctly models the possible effects of non-fragment nodes in the whole-program analysis. Using these conditions (together with Conditions 1–4), we can prove that $\alpha_P(MORP_P(n), MORP'(n))$ for each node $n \in Fr$. Since the fragment analysis uses the solution engine of a safe whole-program analysis, the fragment analysis solution is guaranteed to be a safe approximation of $MORP'(n)$, and therefore safe according to Definition 2.

In some cases, instead of proving safety with respect to *any* whole-program solution engine, we are only interested in proving safety with respect to a *particular* engine (or category of engines). In this case, Conditions 5 and 6 may be too strong, because they are designed to guarantee that $MORP'(n)$ is safe (which implies safety with respect to all engines), while we are only interested in proving the safety of the solution S'_n computed with that particular engine. In other words, even if $\alpha_P(MORP_P(n), MORP'(n))$ does *not* hold, we may still be able to prove that $\alpha_P(MORP_P(n), S'_n)$. In this case, we can relax Conditions 5 and 6 to take into account the properties of the engine under consideration. In particular, we have defined relaxed correctness conditions that are appropriate for flow-insensitive, context-insensitive solution engines, and another set of relaxed conditions that are useful for flow-sensitive, context-insensitive solution engines.

There are two advantages in defining and using these relaxed conditions. First, they provide analysis designers with more freedom in making design choices: fragment analyses that cannot be proven to be safe using the general conditions from Section 2.2.4 can now be proven safe using engine-specific knowledge. Second, these relaxed conditions are simpler than the general conditions, and therefore are easier to use by analysis designers while designing the fragment analysis and when proving analysis safety.

Flow-Insensitive Context-Insensitive Analysis

Suppose that the fragment analysis uses the solution engine of a flow-insensitive, context-insensitive whole-program analysis, and that this engine can be described using the model presented in Section 2.1: $S^0 = \eta, S^i = f_{all}(S^{i-1})$, with a final solution $S^* = \lim_{i \to \infty} S^i$. Note that the actual engine may use a variety of methods to compute S^*, not just the simple fixed-point iteration described by $S^i = f_{all}(S^{i-1})$. For example, instead of applying f_{all}, the engine may apply individual transfer functions in particular order that ensures fast convergence. In fact, the analysis may even compute an approximation of S^* in order to reduce analysis time. While there could be a wide variety of solution procedures, we assume that the engine is conceptually based on the above model, and therefore computes a solution $S' \leq S^*$. Essentially, the above definition of S^* provides a generic model that allows us to reason about the properties of an entire class of solution engines.[5]

If a fragment analysis uses a whole-program solution engine that fits the above model, it should satisfy the following relaxed versions of Conditions 5 and 6 from Section 2.2.4:

Condition 7: For any path $q \in \mathit{OutRP}_P(\rho, n)$ there exists a set of nodes $\{m_1, \ldots, m_r\}$ in graph G' with a corresponding set of transfer functions $\{f'_{m_1}, \ldots, f'_{m_r}\}$ such that $\alpha_P(f_q\, ,\, f'_{m_1} \circ \ldots \circ f'_{m_r})$. The same is true for any path $q \in \mathit{OutSLRP}_P(n, r)$.

Since the solution engine ignores the flow of control in G', we can define the correctness condition in terms of sets of nodes in G', rather than in terms of realizable paths in G' as in Conditions 5 and 6. Given Conditions 1–4 and 7, it is straightforward to prove that a flow-insensitive context-insensitive fragment analysis is safe according to Definition 2 (assuming its solution engine can be described using the above model). The proof considers the solution S^* defined earlier, and shows that for each $P \in \mathit{WholePrg}(\mathit{Fr}, I)$, each node

[5]For example, in Chapter 3 we define such a generic model to describe several different flow-insensitive context-insensitive points-to analyses.

$n \in Fr$, and each realizable path $q = (\rho, \dots, n)$ in P, it is true that $\alpha_P(f_q(\eta_P), S^*)$. This together with Conditions 1 and 2 guarantees that $\alpha_P(MORP_P(n), S')$. To prove that $\alpha_P(f_q(\eta_P), S^*)$, it is enough to show that there exists some set of functions $\{f'_i | f'_i \in F'\}$ such that $\alpha_P(f_q, f'_1 \circ \dots \circ f'_k)$; this together with the obvious $S^* \leq (f'_1 \circ \dots \circ f'_k)(\eta')$ shows the desired result. The existence of such $\{f'_i\}$ is straightforward to prove based on Conditions 4 and 7.

Flow-Sensitive Context-Insensitive Analysis

Suppose that the fragment analysis uses the solution engine of a flow-sensitive, context-insensitive whole-program analysis, and that this engine can be described using the model from Section 2.1. Recall that this model defines the system of equations (2.1) of the form $S = H(S)$, where $S : N \to L$ and $H : (N \to L) \to (N \to L)$. Let $S^* = \lim_{i \to \infty} S^i$, where $S^i = H(S^{i-1})$. This is just a model of the operation of the solution engine: the actual engine may use a different computational procedure (e.g., ordered applications of transfer functions using a worklist), and may introduce approximations to reduce analysis cost. However, we assume that the engine is conceptually based on the above model, and therefore computes a solution $S' \leq S^*$. As before, our goal is to define a generic model that can be used to reason about the properties of an entire class of solution engines.

If a fragment analysis uses a whole-program solution engine that can be described with the above model, we can define the following relaxed versions of Conditions 5 and 6 from Section 2.2.4:

Condition 8: For any path $q \in OutRP_P(\rho, n)$ there exists a path $q' = (\rho', \dots, n)$ in G' such that $\alpha_P(f_q, f'_{q'})$. Similarly, for any path $q \in OutSLRP_P(n, r)$, there exists a path $q' = (n, \dots, r)$ in G' such that $\alpha_P(f_q, f'_{q'})$.

Since the solution engine takes into account *all* paths in G' (both realizable and non-realizable), we can define the correctness condition in terms of arbitrary paths in G', rather than in terms of realizable paths in G' as in Conditions 5 and 6. Given Conditions 1–4 and 8, it can be proven that a flow-sensitive, context-insensitive fragment analysis is safe according to Definition 2. The proof shows that for each program $P \in WholePrg(Fr, I)$, each node $n \in Fr$, and each realizable path $q = (\rho, \ldots, n)$ in P, it is true that $\alpha_P(f_q(\eta_P), S_n^*)$. This can be proven by showing that there exists a path $q' = (\rho', \ldots, n)$ in G' such that $\alpha_P(f_q, f'_{q'})$; this together with $S_n^* \leq f'_{q'}(\eta')$ shows the desired property. The existence of such a path q' can be proven based on Conditions 4 and 8.

Chapter 3

Points-to Analysis and Side-effect Analysis for Programs Built with Precompiled Libraries

Large programs are typically built from separate modules. Such modular development enables people to understand and manage complex software systems. This development model also allows practical compilation: instead of (re)compiling large programs from scratch, compilers can perform separate compilation of individual modules. Modular development allows sharing of modules between programs: for example, an already compiled library module can be reused with no development effort or compilation cost. Modularity also enables better distribution of the implementation effort—different modules can be developed by different teams, at different times and in separate locations.

Traditional *whole-program dataflow analysis* cannot be used directly in the context of a modular development process. To make existing analyses useful in realistic compilers and software productivity tools, analysis techniques must be adapted to handle such modular development. In this chapter we investigate one instance of this problem. Our work considers points-to analysis and side-effect analysis for programs built with *precompiled library modules*. Reusable modules are routinely employed in order to reduce development costs. Such modules are designed to be used by a variety of (as yet unknown) clients, and are typically packaged as precompiled libraries that are statically or dynamically linked with the client code. To simplify the discussion, for most of the chapter we consider programs containing two modules: a *library module* that is developed and compiled independently of any particular client, and a *client module* that uses the functionality provided by the library module. (Section 3.4 discusses analysis of programs built with multiple library modules.) In the context of such programs, we propose fragment points-to analyses and fragment side-effect analyses that can be used to analyze the library module and the client module in separation from each other.

3.1 Whole-program Points-to Analysis and Side-effect Analysis

Modification side-effect analysis (MOD) determines, for each program statement, the set of variables whose values may be modified by executing that statement. The complementary USE analysis computes similar information for the uses of variable values. Such information plays a fundamental role in optimizing compilers and software productivity tools: it enables a variety of other analyses (e.g., reaching definitions analysis, live variables analysis, etc.), which in turn are needed for code optimization and for program understanding, restructuring and testing. For brevity, we only discuss MOD analysis; all techniques trivially apply to USE analysis, because the two analysis problems are essentially identical.

Side-effect analysis for languages like C is difficult because *pointers* allow indirect memory accesses and indirect procedure calls. Typically, MOD analysis uses the output of a *points-to analysis* to resolve pointer dereferences. Various whole-program points-to analyses have been developed [35, 33, 21, 4, 68, 60, 69, 56, 36, 25, 17, 23, 12, 32, 54]. In this chapter we focus on the category of flow- and context-insensitive analyses [4, 60, 69, 56, 17, 32]. Such analyses ignore the flow of control between program points and do not distinguish between different calling contexts of procedures. As a result, analyses from this category are efficient and can be used in production-strength tools without introducing substantial time/space overhead.

Several combinations of a whole-program MOD analysis and a whole-program flow- and context-insensitive points-to analysis have been investigated [55, 34, 54]. Similarly to [55, 34], we consider a whole-program MOD analysis based on Andersen's whole-program flow- and context-insensitive points-to analysis for C [4]. Even though we investigate these specific analyses, our techniques also apply to similar whole-program flow- and context-insensitive points-to analyses (e.g., [60, 56, 17]), because these analyses can be thought of as approximate algorithms for performing Andersen's analysis.

In this section we define a conceptual model of Andersen's whole-program points-to analysis and a whole-program MOD analysis based on it. In the remaining sections of

Program → GlobalDecl* Procedure*
GlobalDecl → **global** VarList
Procedure → **proc** Var(Formals)
  [**returns** Var] **{** Body **}**
Formals → ϵ | VarList
Body → LocalDecl* StLocalDecl* Stmt*
LocalDecl → **local** VarList
StLocalDecl → **static local** VarList
Stmt → Call | IndirectCall | Assignment
Call → [Var =] Var(Actuals)

IndirectCall → [Var =] (*Var)(Actuals)
Actuals → ϵ | VarList
Assignment → Var = Var | Var = **&**Var |
  Var = *Var | *Var = Var |
  Var = NptrExpr
NptrExpr → **const** | **unop** Var |
  Var **binop** Var
VarList → Var | Var **,** VarList
Var → **id**

Figure 3.1: Grammar for the program representation. Terminals are shown in boldface. A procedure may have a special return variable (defined by **returns**) that is assigned the return value of the procedure. NptrExpr is an expression with non-pointer values.

this chapter we describe fragment analyses that are derived from these whole-program analyses.

3.1.1 Program Representation

For the purpose of this work, we assume that the program representation is defined by the grammar in Figure 3.1. This representation has a simplified form similar to the kinds of program representations that are typically supplied as input to points-to and side-effect analyses. Control-flow statements (`if`, `while`, etc.) are irrelevant with respect to flow-insensitive analyses and are not included in the representation. Because of the weak type system of C (due to typecasting and union types), we assume that type information is not used in the points-to and MOD analyses[1], and therefore the program representation is untyped. Similarly to [60, 56, 55, 22, 25, 17, 23, 34, 32], structures and arrays are represented as monolithic objects without distinguishing their individual elements. We assume that calls to *malloc* and other heap-allocating functions are replaced by statements of the form "$x = \&heap_i$", where $heap_i$ is a variable unique to the allocation site. Figure 3.2 shows a sample program with two modules: module *Client* containing procedures *main* and *div*, and module *Lib* containing procedures *exec* and *neg*.

[1]This analysis approach is the simplest to implement and therefore most likely to be the first one employed by realistic compilers and tools. Our techniques can be easily adapted to approaches that use some form of type information.

```
global g
proc main() {
  local x,y,z,w
  1: x = 1
  2: y = 2
  3: z = &x
  4: g = &y
  5: w = &div
  6: exec(z,w)
}
proc div(a,b) {
  local c,d,e
  7: c = *a
  8: d = *b
  9: e = c / d
  10: *a = e
}
```

Client

```
proc exec(p,fp) {
  local s,u,q,t
  11: s = 3  12: u = 4
  13: t = p
  14: (*fp)(g,t)
  15: q = &s  16: neg(q)
  17: q = &u  18: neg(q)
  19: *t = u
  20: g = t
}
proc neg(r) {
  local i,j
  21: i = *r
  22: j = -i
  23: *r = j
}
```

Lib

Var	$Pt(v)$
z	x
g	x, y
w, fp	div
p, t, b	x
a	x, y
q, r	s, u

Stmt	$Mod(s)$
1	x
6	g, x, y
10	x, y
14	x, y
16,18	s, u
19	x
23	s, u

Figure 3.2: Program with two modules *Client* = $\{main, div\}$ and *Lib* = $\{exec, neg\}$, and the points-to and MOD solutions computed by the whole-program analyses.

3.1.2 Points-to Analysis

Let V be the set of variables in the program representation, as defined by the last production in Figure 3.1. We classify the elements of V as (i) *global variables*, (ii) *procedure variables*, which occur immediately after the **proc** terminal and denote the names of procedures, (iii) *local variables*, including formals and return variables, and (iv) *heap variables* introduced at heap-allocation sites. The analysis constructs a *points-to graph* in which nodes correspond to variables from V. A directed edge (v_1, v_2) shows that one of the memory locations represented by v_1 may contain the address of one of the memory locations represented by v_2 (i.e., v_1 may point to v_2).

We can define a lattice of points-to graphs $L = \mathcal{P}(V \times V)$, where $\mathcal{P}(X)$ denotes the power set of X. As usual, in this lattice the partial order $\leq$ is the "is-superset-of" relation $\supseteq$, the meet operation $\wedge$ is set union, and the greatest element is $\emptyset$. Each statement in the program defines a *transfer function* $f : L \to L$ that encodes the points-to effects of the statement. For example, the transfer function for statement "$p = q$" is $f(G) = G \cup \{(p, x) \mid (q, x) \in G\}$. As another example, the transfer function for "$*p = q$" is $f(G) = G \cup \{(x, y) \mid (p, x) \in G \wedge (q, y) \in G\}$. We can define similar transfer functions for all categories of statements described by the `Stmt` non-terminal from the grammar in Figure 3.1.

Conceptually, the analysis starts with an empty graph and applies transfer functions until a fixed-point solution is obtained. Figure 3.2 shows the points-to solution for the sample program; $Pt(v)$ denotes the points-to set of v. The process of computing a points-to solution can be modeled using the approach from Section 2.1. Formally, we can define a compound transfer function $f_{all} : L \rightarrow L$ of the form $f_{all} = (\bigwedge_{n \in N} f_n) \wedge id$. This function summarizes the effects of all transfer functions for individual program statements. Andersen's analysis computes a solution $S^* = \lim_{i \rightarrow \infty} S^i$ where $S^0 = \emptyset, S^i = f_{all}(S^{i-1})$. As discussed in Section 2.2.5, this is just a model of the operation of the analysis: the actual analysis engine need not construct or directly apply f_{all}.

In fact, the above formal model can also be used to reason about other flow- and context-insensitive points-to analyses—for example, the analyses by Steensgaard [60], Shapiro and Horwitz [56], and Das [17]. These analyses can be thought of as computing an approximation of S^* (i.e., the computed solution is $S' \leq S^*$). Since our approach for constructing fragment analyses (presented later) is defined in terms of the above model, our techniques trivially apply not only to Andersen's analysis, but also to other analyses that fit in this model (e.g., [60, 56, 17]).

3.1.3 Side-effect Analysis

We define a conceptual algorithm for whole-program MOD analysis that computes a set of potentially modified variables $Mod(s) \subseteq V$ for each program statement s. The elements of $Mod(s)$ represent memory locations whose values after the execution of s may be different from their values before the execution of s. The algorithm is derived from similar MOD algorithms [55, 34, 54] by adding two *variable filters* that allow certain variables to be excluded from *Mod* sets. This filtering improves the precision of the MOD analysis by compensating for some of the imprecision introduced by the flow- and context-insensitivity of the underlying points-to analysis.

Variable Filters

Our first filter is based on the following observation: a variable v should be included in $\mathit{Mod}(s)$ only if v represents memory locations whose lifetime is *active* immediately before and immediately after the execution of s. For example, if v is a non-static local variable in procedure P, it represents memory locations whose lifetime is active only for procedures that are directly or transitively called by P. We define a relation $\mathit{active}(s, v)$ that holds only if

- v is a global variable, a static local variable, or a heap variable, or
- v is a non-static local variable in procedure P_1, s belongs to procedure P_2, and either P_2 is reachable from P_1 in the program call graph, or P_1 and P_2 are the same procedure

Our second filter is applied only to $\mathit{Mod}(s)$ of a call statement s. Suppose that s invokes procedure P. Among all locations whose lifetime is active with respect to s, only a subset is actually *accessible* by P and the procedures transitively called by P. Intuitively, an active location is accessible only if it can be referenced either directly, or through a series of pointer dereferences. Only accessible variables may be modified by the call to P, and only they should be included in $\mathit{Mod}(s)$.

An active global variable is always directly accessible by P and the procedures transitively called by P. Similarly, an active static local could be directly accessible if its defining procedure is P or a callee of P. However, an active non-static local can only be accessed indirectly through pointer dereferences; any *direct* reference to this local in P or in the callees of P actually accesses a different run-time instance of the local. Finally, heap variables can only be accessed through pointers because the program does not contain any direct references to such variables. Based on these observations, for each call statement s we define a relation $\mathit{accessible}(s, v)$ that holds only if $\mathit{active}(s, v)$ holds and

- v is a global variable or a static local variable, or
- $v \in \mathit{Pt}(a)$, where a is an actual parameter of s, or

- $v \in Pt(w)$ and $accessible(s, w)$ holds for some $w \in V$

For example, variable u from Figure 3.2 is active for all statements in procedures *exec*, *div*, and *neg*. With respect to calls 16 and 18, u is accessible because it belongs to the points-to set of actual q; however, u is not accessible for call 14.

In our conceptual MOD algorithm, a variable v is added to $Mod(s)$ only if $active(s, v)$ holds. In addition, v is included in $Mod(s)$ for a call statement s only if $accessible(s, v)$ holds. We use these filters in our algorithm formulation in order to define a more precise semantics for the MOD analysis. This allows us to improve the quality of our fragment analyses, as described later.

Analysis Algorithm

The MOD algorithm is described in Figure 3.3. Input map *SynMod* stores the syntactic modifications that occur in program statements. Each *syntactic modification* is a pair (v, d), where v is variable that occurs on the left-hand side of an assignment or a call, and $d \in \{D, I\}$ indicates whether the modification is direct or indirect. For example, in Figure 3.2, $SynMod(s)$ is (x, D) for statement 1 and (a, I) for statement 10.

The first analysis phase (lines 1–5 in Figure 3.3) resolves indirect calls and constructs a safe approximation of the program call graph. The second phase (lines 6–13) computes initial *Mod* sets without taking into account the effects of procedure calls. This phase also initializes map *ProcMod*, which stores the sets of variables modified by each procedure. The third phase (lines 14–18) is a fixed-point computation based on the program call graph; it uses $ProcMod(P)$ to update the *Mod* sets of calls to P and the *ProcMod* sets for the calling procedures. Figure 3.2 shows the computed *Mod* sets for some of the statements in the sample program.

Filters *active* and *accessible* are used whenever a variable is added to a *Mod* set (lines 12 and 17). In addition, we filter out *direct* modifications of non-static local variables (lines 9–10), because the lifetime of the modified memory location terminates when the procedure returns. This filtering allows the analysis to compute more precise information for non-static local variables declared in recursive procedures, by distinguishing among

```
input     $Stmt$: set of statements             $Proc$: set of procedures
          $SynMod$: $Stmt \rightarrow V \times \{D, I\}$    $Pt$: $V \rightarrow \mathcal{P}(V)$
output    $Mod$: $Stmt \rightarrow \mathcal{P}(V)$
declare   $Called$: $Stmt \rightarrow \mathcal{P}(Proc)$     $ProcMod$: $Proc \rightarrow \mathcal{P}(V)$

[1]   foreach $s \in Stmt$ do
[2]       if s is a direct call to procedure $P$ then
[3]           $Called(s) := \{P\}$
[4]       if s is an indirect call through pointer $q$ then
[5]           $Called(s) := \{ P \mid \text{procedure } P \in Pt(q)\}$

[6]   foreach $s \in Stmt$ do
[7]       if $SynMod(s) = (v, D)$ then
[8]           $Mod(s) := \{v\}$
[9]           if $v$ is global or static local then
[10]              add $\{v\}$ to $ProcMod(EnclosingProc(s))$
[11]      if $SynMod(s) = (v, I)$ then
[12]          $Mod(s) := \{x \mid x \in Pt(v) \wedge active(s, x) \}$
[13]          add $Mod(s)$ to $ProcMod(EnclosingProc(s))$

[14]  while changes occur in $Mod$ or $ProcMod$ do
[15]      foreach call statement $s \in Stmt$ do
[16]          foreach $P \in Called(s)$ do
[17]              $Mod(s) := Mod(s) \cup \{x \mid x \in ProcMod(P) \wedge accessible(s, x)\}$
[18]          add $Mod(s)$ to $ProcMod(EnclosingProc(s))$
```

Figure 3.3: Whole-program MOD analysis.

different recursive instantiations of such variables.

3.2 Fragment Analysis of Library Modules

In this section we consider the problem of analyzing a library module in isolation from its client modules. For example, suppose that a library vendor wants to use an optimizing compiler to produce a highly optimized library binary that will eventually be used by many (unknown) different clients. In this case the compiler cannot use whole-program analysis, and it needs to use some form of fragment analysis (where the analyzed fragment is the library). Similarly, if a library developer wants to use a software engineering tool for understanding/restructuring/testing of the library module, this tool needs to use fragment analysis.

We consider one particular instance of this problem: performing Andersen's points-to analysis and the corresponding MOD analysis (as defined in Section 3.1) on a given library module, under the assumption that many different clients of this module will be written in the future. We assume that the only knowledge we have about the possible future clients of the library is the list of library procedures/variables that may be directly referenced by client modules—that is, the interface of the library module. This list is the whole-program information I (as defined in Section 2.2) that is part of our analysis input. Intuitively, given the library and I, our fragment analyses have to model *all possible* points-to relationships and MOD relationships that may hold with respect to *at least one* of the possible client modules of the library. To achieve this, the analyses have to make conservative assumptions about the possible effects of the unknown code from the client modules.

Given a library module *Lib*, consider a complete program P containing *Lib* and some client module. Let $\mathit{WholePrg}(\mathit{Lib})$ be the (infinite) set of all such complete programs. We use V_P to denote the variable set of any such P, as defined in Section 3.1.2. Let $V_L \subseteq V_P$ be the set of all variables that occur in statements in *Lib* (this set is independent of any particular P). Also, let $V_{exp} \subseteq V_L$ be the set of all variables that may be explicitly referenced by client modules; we refer to such variables as *exported*. Exported variables are either globals that could be directly accessed by library clients, or names of procedures that could be directly called by client code.

Example. We use module *Lib* from Figure 3.2 as our running example; for convenience, the module is shown again in Figure 3.5. For this module, $V_L = \{exec, p, fp, s, u, t, g, q, neg, r, i, j\}$. For the purpose of this example, we assume that $V_{exp} = \{g, exec\}$. Note that the complete program in Figure 3.2 is just one of the (infinitely many) elements of $\mathit{WholePrg}(\mathit{Lib})$. $\diamondsuit$

3.2.1 Placeholders

In our fragment analyses, the possible effects of the unknown code from client modules are represented by introducing *placeholder statements*. The placeholder statements are located

```
global ph_var
proc ph_proc(f$_1$,..,f$_n$) returns ph_proc_ret {
  ph_var = f$_i$  $(1 \leq i \leq n)$        ph_var = &v  (v$\in V_{exp}$)       ph_var = &ph_var
  ph_var = *ph_var               *ph_var = ph_var              ph_var = &ph_proc
  ph_var = (*ph_var)(ph_var,..,ph_var)  (with $m$ actuals)
  ph_proc_ret = ph_var
}
```

Figure 3.4: Placeholder procedure and placeholder statements.

inside a *placeholder procedure* `ph_proc` that represents all procedures in all possible client modules. The statements use a *placeholder variable* `ph_var` that represents all global, local, and heap variables $v \in (V_P - V_L)$ for all $P \in \mathit{WholePrg}(\mathit{Lib})$. The placeholder procedure and the placeholder statements are shown in Figure 3.4. Intuitively, each statement represents different kinds of statements that could occur in client modules: for example, "`ph_var=&v`" for $v \in V_{exp}$ models statements of the form "`u=&v`" where $u \in (V_P - V_L)$ for some complete program P.

The indirect call through `ph_var` represents all calls originating from client modules. In the fragment analyses, the targets of this call could be (i) `ph_proc`, (ii) the procedures from V_{exp}, or (iii) any library procedure whose address is taken somewhere in *Lib*. To model all possible formal-actual pairs, the number of actuals m should be equal to the maximum number of formals for all possible target procedures. Similarly, all callbacks from the library are represented by indirect calls to `ph_proc`; thus, the number of formals n in `ph_proc` should be equal to the maximum number of actuals used at indirect calls in the library.

Example. Consider module *Lib* from Figure 3.5. For the purpose of this example assume that $V_{exp} = \{g, \mathit{exec}\}$. The library has one indirect call with two actuals; thus, `ph_proc` should have two formals ($n = 2$). The indirect call through `ph_var` has possible targets *exec* and `ph_proc`, and should have two actuals ($m = 2$). $\diamondsuit$

3.2.2 Computing a Solution

The fragment analyses start by creating the placeholders described above. These placeholders are then added to the library, the result is treated as a complete program, and

```
global g
proc exec(p,fp) {                                            proc neg(r) {
  local s,u,q,t    11: s = 3          12: u = 4               local i,j
  13: t = p        14: (*fp)(g,t)     15: q = &u              21: i = *r
  16: neg(q)       17: q = &s         18: neg(q)              22: j = -i
  19: *t = u       20: g = t                                  23: *r = j
}                                                            }
```

$Pt(v) = \emptyset$ for $v \in \{s, u, i, j, neg\}$
$Pt(v) = \{s, u\}$ for $v \in \{q, r\}$
$Pt(v) = \{ph\_var, ph\_proc, g, exec\}$ for every other v in *Lib* and in `ph_proc`

$Mod(s) = \{ph\_var, g\}$ for $s \in \{14, 19\}$ and for the indirect call through *ph_var*
$Mod(23) = \{s, u\}$
$Called(s) = \{ph\_proc, exec\}$ for $s = 14$ and for the indirect call through *ph_var*

Figure 3.5: Module *Lib* ($V_{exp} = \{g, exec\}$) and the fragment analysis solutions.

the solution engines of the whole-program analyses (presented in Section 3.1) are applied to it. This approach is simple to implement by reusing already existing implementations of the whole-program analyses; such reuse is an important practical advantage. The fragment analysis solutions computed with this approach are guaranteed to be *safe abstractions* of all possible points-to and MOD relationships in all possible complete programs $P \in WholePrg(Lib)$; this issue is discussed in detail in the next section.

Example. Figure 3.5 shows the points-to solution computed by the fragment points-to analysis. The solution represents all possible points-to pairs in all complete programs. For example, $ph\_var \in Pt(p)$ shows that p may point to some unknown variable declared in some client module. Similarly, $ph\_proc \in Pt(p)$ indicates that p may point to an unknown procedure from some client module. The first phase of the MOD analysis (lines 1–5 in Figure 3.3) computes a call graph that represents all possible calls in all complete programs. For example, $Called(14) = \{ph\_proc, exec\}$ represents the possible callback from *exec* to some client module, as well as the possible recursive call of *exec*. The remaining phases of the MOD algorithm compute a solution that represents all possible *Mod* sets in all complete programs. Some of the *Mod* sets for *Lib* are shown in Figure 3.5. For example, $ph\_var \in Mod(19)$ shows that statement "`*t=u`" may modify the value of some unknown variable from some client module. ♢

Conceptually, the solutions computed by the fragment analyses can be used in two ways. First, the analyses can determine potential interactions between the library module and unknown client modules (e.g., which library statements may modify client variables or may invoke client procedures). This information is necessary for optimizing compilers and software engineering tools that process the library independently of its clients, and therefore need to take into account all possible library-client interactions. Second, the fragment analyses can prove that certain elements of the library do not interact with the client modules: for example, that regardless of the effects of unknown client code, a given library variable never points to a client variable. Essentially, the complement of the analysis solution can be used to prove the *absence* of library-client interactions. In Section 3.3 we use this information when creating library summaries that describe the possible effects of the library. By identifying the absence of interactions with the potential clients, we can remove redundant information from the library summary.

3.2.3 Analysis Safety

Let V' be the set of all variables in the analyzed library and in the placeholder procedure. The fragment points-to analysis defines a lattice of points-to graphs $L' = \mathcal{P}(V' \times V')$. The analysis also associates transfer functions $f' : L' \rightarrow L'$ with all fragment statements and placeholder statements; these functions are defined similarly to the whole-program transfer functions from Section 3.1.2.

Consider a complete program $P \in \mathit{WholePrg}(\mathit{Lib})$. Recall that we use V_P to denote the variable set of any such P, and $V_L \subseteq V_P$ to denote the set of all variables that occur in statements in *Lib*. For any such P, we can define an abstraction relation $\alpha_P \subseteq V_P \times V'$ as follows:

- $\alpha_P(v, v)$ for any $v \in V_L$
- $\alpha_P(v, \mathit{ph\_var})$ for any global, local, or heap variable $v \in (V_P - V_L)$
- $\alpha_P(v, \mathit{ph\_proc})$ for any procedure variable $v \in (V_P - V_L)$

This definition encodes the idea that in the fragment analysis, all fragment variables are represented by themselves, all unknown non-fragment procedures are represented by `ph_proc`, and all unknown non-fragment variables are represented by `ph_var`. We can generalize α_P to lattice elements (i.e., points-to graphs) as follows: $\alpha_P(x, x')$ iff for each points-to pair $(v, u) \in x$, there exists a points-to pair $(v', u') \in x'$ such that $\alpha_P(u, u')$ and $\alpha_P(v, v')$.

Based on this abstraction relation, the clients of the fragment points-to analysis can infer that certain points-to relationships may hold in the context of whole programs that contain the library. For example, if the fragment analysis reports that some fragment variable v points to the placeholder variable `ph_var`, this means that in some whole program v may point to some unknown non-fragment variable. Of course, we have to make sure that the fragment analysis solution represents *all* possible points-to relationships that may exist in some whole program. More specifically, we have to show that the solution is safe according to Definition 2 from Section 2.2.3. To prove this, we can use the correctness conditions described in Section 2.2.4 and Section 2.2.5. For convenience, these conditions are shown again below.

Condition 1: If $\alpha_P(x, x')$ and $y' \leq x'$, then $\alpha_P(x, y')$

Condition 2: If $\alpha_P(x, x')$ and $\alpha_P(y, x')$, then $\alpha_P(x \wedge y, x')$

For Condition 1, $y' \leq x'$ means that $x' \subseteq y'$, and therefore all points-to pairs from x are represented by corresponding pairs in y'. Similarly, for Condition 2, $x \wedge y$ is $x \cup y$, and clearly there are points-to pairs in x' that represent all pairs in $x \cup y$.

Condition 3: $\alpha_P(\eta_P, \eta')$

Since the starting values $\eta_P = \eta' = \emptyset$, this condition is trivially satisfied.

Let $f_{n,P}$ be the transfer function for node $n \in Lib$ in the whole-program analysis, and f'_n be the transfer function for that same node in the fragment analysis. For any such n, the following should be true:

Condition 4: $\alpha_P(f_{n,P}\ , f'_n)$

The proof of this claim considers the different kinds of transfer functions. For example, for statement "$p = q$", the transfer function (both in the whole-program analysis and in the fragment analysis) is $f(G) = G \cup \{(p,x)|(q,x) \in G\}$. Suppose we have two points-to graphs $G_P \in L_P$ and $G' \in L'$ such that $\alpha_P(G_P, G')$. Therefore, for every $(q,x) \in G_P$ there exists $(q,x') \in G'$ such that $\alpha(x,x')$. Clearly, for every new $(p,x) \in f_{n,P}(G_P)$, there exists a corresponding $(p,x') \in f'_n(G')$. The proof for other kinds of statements can be done in a similar fashion.

Since the points-to analysis is flow-insensitive and context-insensitive, we can use the following correctness condition from Section 2.2.5:

Condition 7: For any path $q \in \mathit{OutRP}_P(\rho, n)$ there exists a set of nodes $\{m_1, \ldots, m_r\}$ in graph G' with a corresponding set of transfer functions $\{f'_{m_1}, \ldots, f'_{m_r}\}$ such that $\alpha_P(f_q\, ,\, f'_{m_1} \circ \ldots \circ f'_{m_r})$. The same is true for any path $q \in \mathit{OutSLRP}_P(n, r)$.

Intuitively, this condition ensures that the fragment analysis correctly represents the effects of paths that lie outside of the fragment. For our purposes, we can prove this condition by showing that the following property holds: for each individual node $n \notin \mathit{Lib}$ in the whole-program analysis, there exists a set of nodes (i.e., statements) $\{m_1, \ldots, m_r\}$ in the fragment analysis such that $\alpha_P(f_{n,P}\, ,\, f'_{m_1} \circ \ldots \circ f'_{m_r})$; clearly, this implies Condition 7. To prove this property, we have to consider the kinds of statements that could occur outside of the fragment. For example, some client module could contain a statement "`v=&g`", where v is a non-fragment variable and g is an exported library global variable. In the fragment analysis, we have a corresponding placeholder statement "`ph_var=&g`" (see Figure 3.4). It is easy to show that α_P holds between the transfer functions for these two statements. In general, it can be proven that for each possible non-fragment statement n in the whole-program analysis, there exists a group of placeholder statements $\{m_1, \ldots, m_r\}$ from Figure 3.4 such that $\alpha_P(f_{n,P}\, ,\, f'_{m_1} \circ \ldots \circ f'_{m_r})$.

As discussed in Section 2.2.4 and Section 2.2.5, Conditions 1–4 and 7 guarantee the safety of the fragment points-to analysis of the library module. The safety of the fragment MOD analysis is straightforward to show, based on the algorithm from Figure 3.3. More precisely, it can be proven that for every statement $n \in \mathit{Lib}$, we have $\alpha_P(\mathit{Mod}_P(n), \mathit{Mod}'(n))$. The proof of this property is based on the safety of the fragment points-to analysis.

3.3 Fragment Analysis of Client Modules

In this section we consider the problem of analyzing a client module that is built on top of an existing reusable library module. We are particularly interested in a scenario where the library module is produced by a vendor that does not provide the source code for the library. This problem is important in the context of component-based software, where systems are built from different components that are developed and assembled by different organizations. Such components are typically distributed in binary form, and their source code is rarely available.

In order to analyze a client module (e.g., in the context of an optimizing compiler or a software engineering tool), it is necessary to use some form of fragment analysis because the source code of the used library module is not available. In this thesis we consider *summary-based fragment analysis* of the client module. Such analysis takes as input the client module (which is the analyzed fragment), as well as summary information about the library module. This summary information describes the possible effects of the library module on the client module (i.e., this is the whole-program information I discussed in Section 2.2.1). Such summary information can be constructed by the library vendor by analyzing the library source code, and can be made available to the clients of the library together with the library object code. Library clients can use this information to perform summary-based fragment analysis of their client modules.

In this section we present an approach for automatic construction of library summary information for the purposes of performing fragment points-to analysis and fragment side-effect analysis of client modules. We discuss fragment analyses based on Andersen's points-to analysis and the MOD analysis described in Section 3.1. Our techniques trivially apply to other flow- and context-insensitive points-to analyses (e.g., [60, 56, 17]) and to MOD analyses based on them.

3.3.1 Summary Construction

Previous work on context-sensitive points-to analysis uses summary information computed for each program procedure [35, 54, 68, 31, 10, 12]. In these approaches, the effects of a procedure P are represented by a *summary function* that encodes the cumulative effects of the transfer functions of all statements from P and from all procedures transitively called by P. Some analyses construct *partial* functions that are defined only for some input values [35, 54, 68]; others use *complete* functions defined for all possible inputs [31, 10, 12]. The latter approach can be used to produce summary information for a library module, by computing and storing the complete summary functions for all exported procedures [31].

The above approaches have one major disadvantage: the implicit assumption that a called procedure can be analyzed either *before* its callers are analyzed, or *while* its callers are being analyzed. This assumption can be easily violated—for example, when a library module calls another library module, there are no guarantees that any summary information will be available for the callee. In the presence of callbacks (e.g., through function pointers in C), the library module may call client modules that do not even exist at the time when the summary is being constructed. For example, for module *Lib* from Figure 3.5, the effects of *exec* cannot be expressed by a summary function because of the indirect call to some unknown client module. In practice, callbacks are often used to increase the flexibility of reusable libraries by allowing client code to define, extend, or modify the behavior of the library. A classic example is a generic sorting procedure that takes as input a pointer to a comparison function defined in the client code. The library modules used as data in our experiments had several examples of callback usage.

Our approach for summary construction is conceptually different from approaches based on complete summary functions. We use summary information that is similar in form to the program representation defined in Figure 3.1; for example, the summary contains statements like those from Figure 3.1. Conceptually, we use elementary transfer functions instead of cumulative summary functions. This approach has two advantages. First, the summary can be constructed completely independently of any callers and callees of the library; therefore, unlike previous work, our approach can handle callbacks. Second, the summary construction algorithm is inexpensive and simple to implement, and is independent of any particular analysis or analysis implementation. Thus, unlike previous approaches, the analysis needed to construct the library summary does not have to be the

1. Variable summary

   $Procedures = \{exec, neg\}$ $\qquad$ $Locals(exec) = \{p, fp, s, u, q, t\}$

   $Globals = \{g\}$ $\qquad$ $Locals(neg) = \{r, i, j\}$

2. Points-to summary

```
proc exec(p,fp)     q=&s       *t=u           i=*r
t=p                 neg(q)     g=t            *r=j
(*fp)(g,t)          q=&u       proc neg(r)
```

3. Mod summary

   $SynMod(exec) = \{(t, I), (g, D)\}$ $\qquad$ $SynMod(neg) = \{(r, I)\}$

   $SynCall(exec) = \{(fp, I), (neg, D)\}$ $\qquad$ $SynCall(neg) = \emptyset$

Figure 3.6: Basic summary for module *Lib*.

same as the analysis that uses this summary to analyze the client module.

The summary information is designed to be *precision-preserving*. With respect to the needs of library clients, the solutions computed by the summary-based fragment analyses are the same as the solutions that would have been computed if the standard whole-program analyses were possible. This ensures the best possible cost and precision for the subsequent users of the fragment analysis solutions.

Basic Summary

The basic summary is the simplest summary information produced by our approach. Figure 3.6 shows the basic summary for module *Lib* from Figure 3.5. The summary has three parts. The *variable summary* contains information about relevant library variables. The *points-to summary* contains all library statements that are relevant to points-to analysis; statements of the form "Var = NptrExpr" (see Figure 3.1) are not included. The `proc` declarations are used by the subsequent points-to analysis to model the formal-actual pairings at procedure calls. The *Mod summary* contains syntactic modifications and syntactic calls for each library procedure. A syntactic modification (defined in Section 3.1.3) is a pair (v, D) or (v, I) representing a direct or indirect modification. A syntactic call is a similar pair indicating a direct or indirect call through v. The Mod summary does not include direct modifications of non-static local variables—as discussed in Section 3.1.3, such modifications can be filtered out by the MOD analysis.

3.3.2 Summary-based Fragment Analysis of Client Modules

In the summary-based fragment analysis, the client module (which is the analyzed fragment) is combined with the library summary, and the result is analyzed as if it were a complete program. Thus, already existing implementations of the whole-program analyses from Section 3.1 can be reused with only minor adjustments, which makes the approach simple to implement. For example, the MOD solution engine (described in Figure 3.3) can treat each library procedure P as containing a single placeholder statement with *Called* set constructed from $\mathit{SynCall}(P)$ and with multiple syntactic definitions determined by $\mathit{SynMod}(P)$. Clearly, if the summary-based analyses are provided with the basic summary, the computed points-to and MOD solutions are the same as the solutions that would have been produced by the standard whole-program analyses.

The basic summary is easy to construct and use. However, with this summary, the cost of the summary-based analyses is essentially the same as the cost that would have been incurred if the standard whole-program analyses were possible. This cost can be reduced by performing some analysis work in advance and by encoding the results in the summary. In the next section we describe several summary optimizations that achieve this goal.

3.3.3 Summary Optimizations

In this section we describe three techniques for optimizing the basic summary. The resulting *optimized summary* has two important features. First, the summary is precision-preserving: with respect to library clients, the fragment analysis solutions computed with the optimized summary are the same as the solutions that would have been computed with the basic summary. Second, as demonstrated by our experiments, the cost of the summary-based analyses is significantly reduced when using the optimized summary, compared to using the basic summary.

Variable Substitution

Variable substitution is a technique for reducing the cost of points-to analysis by replacing a set of variables with a single representative variable [50]. In this work we use a particular

1. Variable summary

   $\mathit{Procedures} = \{\mathit{exec}, \mathit{neg}\}$ $\quad$ $\mathit{Locals}(\mathit{exec}) = \{\mathit{fp}, s, u\}$ $\quad$ $\mathit{Reps} = \{\mathit{rep}_1, \mathit{rep}_2\}$

   $\mathit{Globals} = \{g\}$ $\quad$ $\mathit{Locals}(\mathit{neg}) = \{i, j\}$

2. Points-to summary

```
proc exec(rep1,fp)    rep2=&s     *rep1=u     i=*rep2
(*fp)(g,rep1)         rep2=&u     g=rep1      *rep2=j
```

3. Mod summary

   $\mathit{SynMod}(\mathit{exec}) = \{(\mathit{rep}_1, I), (g, D)\}$ $\quad$ $\mathit{SynMod}(\mathit{neg}) = \{(\mathit{rep}_2, I)\}$

   $\mathit{SynCall}(\mathit{exec}) = \{(\mathit{fp}, I), (\mathit{neg}, D)\}$ $\quad$ $\mathit{SynCall}(\mathit{neg}) = \emptyset$

Figure 3.7: Optimization through variable substitution.

precision-preserving substitution that allows us to produce a more compact summary, which in turn reduces the cost of the subsequent summary-based analyses without any loss of precision.

Two variables are *equivalent* if they have the same points-to sets. The substitution is based on mutually disjoint sets of variables $V_1, \ldots, V_k$ such that for each set V_i (i) all elements of V_i are equivalent, (ii) no element of V_i has its address taken (i.e., no element is pointed to by other variables), and (iii) no element of V_i is exported. Each V_i has associated a *representative variable* rep_i. We optimize the points-to summary by replacing all occurrences of a variable $v \in V_i$ with rep_i. In addition, in the Mod summary, every pair (v, I) is replaced with (rep_i, I); this ensures that the subsequent MOD analysis will use the appropriate points-to sets to resolve indirect modifications and indirect calls.

Example. Consider module *Lib* from Figure 3.5 and its basic summary in Figure 3.6. Using an algorithm described later, we identify sets $V_1 = \{p, t\}$ and $V_2 = \{q, r\}$. After the substitution, the summary can be simplified by eliminating trivial statements. For example, "`t=p`" is transformed into "`rep1=rep1`", which can be eliminated. Call "`neg(rep2)`" can also be eliminated. Since this is the only call to *neg* (and *neg* is not exported), declaration "`proc neg(rep2)`" can be removed as well. Figure 3.7 shows the resulting summary, derived from the basic summary in Figure 3.6. $\diamondsuit$

It can be proven that the above optimization is precision-preserving with respect to client modules. More precisely, after this optimization, for any variable x that occurs in some client module, the fragment points-to analysis of that module computes a points-to set $Pt(x)$ that is exactly the same as the set $Pt(x)$ that would have been computed if the basic summary were used. Similarly, for any statement s in the client module, the computed set $Mod(s)$ is the same as the set $Mod(s)$ produced with the basic summary. These properties can be easily proven using an earlier result from [50].

We identify sets of equivalent variables $V_1, \ldots, V_k$ using a linear-time algorithm that extends a similar algorithm from [50]. We start by constructing a *subset graph* $G_{\subseteq}$. The nodes in the graph represent points-to sets, while the edges represent subset relationships between these points-to sets. For each variable $v \in V_L$, the subset graph contains node $n(v)$ and node $n(*v)$. If the address of v is taken anywhere in the library, the subset graph also contains node $n(\&v)$. Node $n(v)$ represents the points-to set $Pt(v)$, node $n(*v)$ represents the union of $Pt(x)$ for all $x \in Pt(v)$, and node $n(\&v)$ represents the points-to set $\{v\}$. Graph edges represent subset relationships between points-to sets: for every edge (n_1, n_2), we have $Pt(n_1) \subseteq Pt(n_2)$. $G_{\subseteq}$ contains edges that represent all subset relationships that can be directly inferred from statements in the library:

- For each assignment "$p = \&x$", $G_{\subseteq}$ contains $(n(\&x), n(p))$ and $(n(x), n(*p))$.
- For each assignment "$p = q$", $G_{\subseteq}$ contains $(n(q), n(p))$ and $(n(*q), n(*p))$.
- For each assignment "$p = *q$", $G_{\subseteq}$ contains $(n(*q), n(p))$
- For each assignment "$*p = q$", $G_{\subseteq}$ contains $(n(q), n(*p))$
- For each pair of an actual a and a corresponding formal f in a direct call, $G_{\subseteq}$ contains edges $(n(a), n(f))$ and $(n(*a), n(*f))$
- For each direct call where r is the return variable of the called procedure and p is assigned the return value at the call site, $G_{\subseteq}$ contains $(n(r), n(p))$ and $(n(*r), n(*p))$

The subset graph represents all subset relationships that can be *directly* inferred from individual library statements. Relationships created indirectly through pointer dereferences are not represented. For example, if the address of a variable v is taken, there could be indirect assignments to v; the subset relationships created by such assignments are not represented by any of v's incoming edges.

A *direct node* $n(v)$ corresponds to a variable v for which we can track directly all values assigned to v. The points-to set of a direct node depends *only* on the points-to sets of its predecessor nodes in the subset graph. A variable node $n(v)$ is direct only if all of the following conditions hold: (i) v is not an exported global or a formal of an exported procedure, (ii) the address of v is not taken, (iii) v is not a formal of a procedure whose address is taken, and (iv) v is not assigned the return value of an indirect call.

After constructing the subset graph, the algorithm identifies all strongly connected components (SCC) in $G_{\subseteq}$ and builds the corresponding SCC-DAG condensation. Since edges in $G_{\subseteq}$ encode subset relationships, all nodes in the same SCC represent the same points-to set, and therefore the corresponding variables are equivalent. In addition, multiple SCCs may represent the same points-to set. To identify such sets of equivalent SCCs, the algorithm performs a topological sort order traversal of the SCC-DAG. During this traversal, an integer label is assigned to each visited SCC. If a SCC x contains only direct nodes and if all of its predecessor SCCs have the same integer label, that same label is assigned to x. In all other cases, x is assigned a fresh label. At the end of the traversal, any two SCCs that have the same label are guaranteed to represent the same points-to set, and therefore the variables in these SCCs are equivalent.

Example. Consider module *Lib* from Figure 3.5. Because of statement "$t = p$", the subset graph contains an edge $(n(p), n(t))$. Two of the SCCs in $G_{\subseteq}$ are $\{n(p)\}$ and $\{n(t)\}$. Since $n(t)$ is a direct node and SCC $\{n(t)\}$ has a single predecessor SCC $\{n(p)\}$ in the SCC-DAG, $\{n(t)\}$ will be assigned the same label as $\{n(p)\}$. Thus, the algorithm determines that p and t are equivalent variables and reports set $V_1 = \{p, t\}$. Similarly, the algorithm detects and reports set $V_2 = \{q, r\}$. These two sets are then used to perform variable substitution as shown in Figure 3.7. $\diamondsuit$

In addition to producing a more compact summary, we use the computed substitution to reduce the cost of the fragment points-to analysis of the library module described in Section 3.2. During the analysis, every occurrence of $v \in V_i$ in library statements is treated as an occurrence of rep_i. In the final solution, the points-to set of v is defined to be the same as the points-to set computed for rep_i. It is easy to show that this technique produces the same points-to sets as the original analysis.

Statement Elimination

After variable substitution, the points-to summary is simplified further by removing statements that have no effect on client modules. A library variable $v \in V_L$ is *client-inactive* if v is a non-static local in procedure P and there is no path from P to placeholder procedure `ph_proc` in the call graph computed by the fragment MOD analysis of the library module.[2] This call graph represents all possible call graphs for all complete programs; thus, for any such v, $active(s, v)$ (defined in Section 3.1.3) is false for all statements s in all client modules. Intuitively, a client-inactive variable can never be live on the run-time stack when client code is being executed.

Let $Reach(u)$ be the set of all variables reachable from u in the points-to graph computed by the fragment points-to analysis of the library module.[3] A library variable $v \in V_L$ is *client-inaccessible* if (i) v is not a global, static local, or procedure variable, and (ii) v does not belong to $Reach(u)$ for any global or static local variable u, including `ph_var`. Based on the safety properties from Section 3.2.3, it is easy to show that in this case $accessible(s, v)$ (defined in Section 3.1.3) is false for all call statements s in all client modules. Intuitively, a client-inaccessible variable can never be accessed during the execution of call statements located in client modules.

In the summary-based MOD analysis, any variable that is client-inactive or client-inaccessible will not be included in any *Mod* set for any statement in a client module. We refer to such variables as *removable.* The optimization eliminates from the points-to summary certain statements that are only relevant with respect to removable variables. As a result of this elimination, the summary-based points-to analysis computes a solution in which some points-to pairs (p, v) are missing. However, we can guarantee that for any such missing pair, v is a removable variable. Clearly, variables that occur in client modules cannot point to removable variables. Furthermore, it is easy to show that the removal

[2]The call graph is computed at lines 1–5 in Figure 3.3.

[3]Variable w is reachable from u if there exists a path from u to w containing at least one edge.

of such points-to pairs (p, v) does not affect *Mod* sets for statements in client modules. Therefore, the optimization preserves the safety of the summary-based fragment points-to analysis and fragment MOD analysis of the client component.

The optimization is based on the fact that certain variables can only be used to access removable variables. Variable v is *irrelevant* if $Reach(v)$ contains only removable variables. Certain statements involving irrelevant variables can be safely eliminated from the points-to summary:

- "$p = \&q$", if q is removable and irrelevant
- "$p = q$", "$p = *q$", and "$*p = q$", if p or q is irrelevant
- calls "$p = f(q_1, \ldots, q_n)$" and "$p = (*fp)(q_1, \ldots, q_n)$", if all of $p, q_1, \ldots, q_n$ are irrelevant

Intuitively, the removal of such statements does not "break" points-to chains that end at non-removable variables. This guarantees the safety of the optimization: it can be proven that the points-to solution computed after statement elimination differs from the solution computed without statement elimination only by points-to pairs (p, v) in which v is a removable variable.

Example. Consider module *Lib* and the fragment analysis solutions from Figure 3.5. Variables $\{r, i, j\}$ are client-inactive because the call graph does not contain a path from *neg* to *ph_proc*. Variables $\{p, fp, s, u, q, t, r, i, j\}$ are client-inaccessible because they are not reachable from g or *ph_var* in the points-to graph. Variables $\{s, u, i, j\}$ have empty points-to sets and are irrelevant. Variables $\{q, r, rep_2\}$ can only reach s and u and are also irrelevant. Therefore, the following statements can be safely removed from the points-to summary in Figure 3.7: `rep2=&s`, `rep2=&u`, `*rep1=u`, `i=*rep2`, and `*rep2=j`. $\diamondsuit$

Identifying client-inaccessible and irrelevant variables requires various reachability computations in the points-to graph computed by the fragment analysis of the library module. We reduce the cost of these traversals by merging `ph_var` with all of its successor nodes in the graph: during the reachability computations, all successors of `ph_var` are considered merged with `ph_var`, and any edges incident to these nodes are redirected to `ph_var`. It

1. Variable summary

    $Procedures = \{exec, neg\}$ $\quad Locals(exec) = \{fp\}$ $\quad Reps = \{rep_1\}$

    $Globals = \{g\}$ $\quad Locals(neg) = \emptyset$

2. Points-to summary

```
proc exec(rep1,fp)   (*fp)(g,rep1)    g=rep1
```

3. Mod summary

    $SynMod(exec) = \{(rep_1, I), (g, D)\}$ $\quad SynMod(neg) = \emptyset$

    $SynCall(exec) = \{(fp, I), (neg, D)\}$ $\quad SynCall(neg) = \emptyset$

Figure 3.8: Final optimized summary.

is easy to show that after this optimization, the traversals of the points-to graph identify exactly the same client-inaccessible and irrelevant variables as with the original points-to graph.

Modification Elimination

This optimization removes syntactic modifications that are irrelevant with respect to client modules. Syntactic modification (v, I) can be removed from the Mod summary if $Pt(v)$ contains only removable variables. Clearly, this optimization does not affect the *Mod* sets of statements in client modules. In Figure 3.7, (rep_2, I) can be removed because rep_2 points only to removable variables s and u. The final optimized summary for our running example is shown in Figure 3.8.

3.4 Programs Built with Multiple Libraries

Up to this point, we have considered programs that are built with one library module and one client module. Consider now a complete program containing one client module and multiple library modules $L_1, \ldots, L_n$. In the simplest case, the library modules are independent: no module references variables exported by other modules. The fragment analyses and the summary optimizations presented earlier can be trivially extended to handle this case.

Suppose that module L_1 references variables exported by another module L_2—either by accessing exported globals, or by calling exported procedures. We say that such variables are *imported* by L_1. The fragment analyses and the summary construction for L_1 must be extended to handle the additional interactions with L_2.

Two approaches can be used in this situation: (i) the analyses of L_1 can be performed completely independently of L_2, or (ii) L_1 can be analyzed by using an existing library summary for L_2. The first approach is needed if no summary information about L_2 is available—for example, if no summary construction capabilities existed at the time when L_2 was compiled. The second approach can be applied if a library summary for L_2 had been constructed in advance and had been stored together with L_2's binary.

3.4.1 Case 1: A summary for L_2 is available

If a precomputed summary for L_2 is available, it can be combined with L_1 and the result can be treated as a single library to which the fragment analyses from Section 3.2 are applied. In addition, the techniques from Section 3.3.3 can be applied to this "combined library" to identify variables from L_1 that are equivalent, removable, or irrelevant. This information can be used to optimize the basic summary for L_1.

3.4.2 Case 2: A summary for L_2 is not available

In the case when no summary for L_2 is available, the fragment analyses of L_1 are obtained by three adjustments of the fragment analyses from Section 3.2. First, a placeholder statement "`ph_var=&v`" should be introduced for any imported global `v`. Second, all direct calls to imported procedures should be replaced by calls to placeholder procedure `ph_proc`. Finally, statements "`v=&P`", where `P` is an imported procedure, should be replaced with "`v=&ph_proc`". The last two adjustments ensure that direct and indirect calls to imported procedures are modeled by calls to `ph_proc`.

When no summary for L_2 is available, the summary optimizations from Section 3.3.3 need two adjustments when constructing an optimized summary for L_1. First, variable substitution should not be performed for imported variables; this ensures that variables that occur in both libraries will be properly matched by the subsequent analyses of the

Program	#LOC	#Statements	Library	#LOC	#Statements
bzip2-0.9.0c	6.3K	8453	libbz2	4.5K (71%)	7263 (86%)
tiff2ps-3.4	20.9K	24618	libtiff-3.4	19.6K (94%)	20688 (84%)
cjpeg-5b	22.7K	20610	libjpeg-5b	19.1K (84%)	17343 (84%)
gasp-1.2	26.0K	12611	libiberty	11.2K (43%)	6259 (50%)
unzip-5.40	27.6K	27001	zlib-1.1.3	8.0K (29%)	9005 (33%)
fudgit-2.41	29.8K	27903	readline-2.0	14.8K (50%)	10788 (39%)
gnuplot-3.7.1	66.3K	50522	libgd-1.3	22.2K (34%)	2965 (6%)
povray-3.1	133.9K	112369	libpng-1.0.3	25.7K (19%)	25322 (23%)

Table 3.1: Data programs and libraries. Last two columns show absolute and relative library size.

client module. Second, the algorithm for detecting equivalent variables (described in Section 3.3.3) should take into account that a variable node $n(v)$ is not direct if v is an imported global variable, or if v is assigned the value returned by a call to an imported procedure. No adjustments are needed for the rest of the summary optimizations.

3.5 Empirical Results

For our experiments, we implemented the fragment points-to analyses of the library module and the client module, as well as the summary construction techniques from Section 3.3.3. Our implementation of Andersen's analysis is based on the BANE toolkit for constraint-based analysis [22]; the analysis is performed by generating and solving a system of set-inclusion constraints. We measured (i) the cost of the fragment points-to analysis of the library, (ii) the cost of constructing the optimized summary, (iii) the size of the optimized summary, and (iv) the cost of the summary-based fragment points-to analysis of the client module.

Table 3.1 describes our C data programs. Each program contains a well-defined library module that is designed as a general-purpose library and is developed independently of any client applications. For example, `unzip` is an extraction utility for compressed archives that uses the general-purpose data compression library `zlib`. Similarly, `fudgit` is a fitting program that uses the GNU Readline Library to provide command line interface. We added to each library module a set of stubs representing the effects of standard C library functions (e.g., `strcpy`, `cos`, `rand`); the stubs are summaries produced by hand from the

Library	T_{pt} (sec)	T_{sub} (sec)	T_{elim} (sec)
libgd	3.9	0.4	0.1
libiberty	5.8	0.5	0.1
libbz2	4.1	0.8	0.1
zlib	6.5	1.0	0.1
readline	10.2	1.3	0.1
libjpeg	15.1	2.5	0.1
libtiff	23.4	3.6	0.2
libpng	19.8	4.0	0.2

Table 3.2: Cost of library analysis and summary construction. T_{pt} is the running time of the fragment points-to analysis, T_{sub} is the time to compute the variable substitution, and T_{elim} is the time to perform statement elimination and modification elimination.

specifications of the library functions.[4] During the fragment analyses and the summary construction, the stubs were treated as part of the library module.

The first part of Table 3.1 shows our data programs and their sizes (including the library modules) in terms of number of lines of source code. The third column shows the number of pointer-related statements in the program representation; this number is the size of the input for the points-to analysis. The second part of the table shows the sizes of the library modules, as an absolute value and as percentage of the corresponding measurement for the whole program. For example, for `povray`, 19% (25.7K) of the source code lines are in the library module, and the remaining 81% (108.2K) are in the client module.

For our first set of experiments, we measured the cost of the fragment points-to analysis of the library module, as well as the cost of constructing the optimized summary. The results from these experiments are shown in Table 3.2.[5] Column T_{pt} shows the running time of the fragment points-to analysis of the library module. Column T_{sub} contains the time needed by the algorithm from Section 3.3.3 to compute the variable substitution. Column T_{elim} shows the time needed to identify removable and irrelevant variables (as described in Section 3.3.3) in order to perform statement elimination and modification elimination.

[4]In the future it would be interesting to investigate how our approach can be used to produce summaries for the standard libraries.

[5]These and all subsequent experiments were performed on a 360MHz Sun Ultra-60 with 512MB memory. The reported times are the median values out of five runs.

Library	S_{basic} (KB)	S_{opt} (KB)	Reduction	S_{obj} (KB)	S_{opt} / S_{obj}
libgd	67.7	25.8	62%	159.2	16%
libiberty	128.9	43.9	66%	195.4	22%
libbz2	145.7	37.1	75%	53.6	69%
zlib	75.9	40.3	47%	70.6	57%
readline	216.5	68.0	69%	193.5	35%
libjpeg	365.5	100.6	72%	132.6	76%
libtiff	455.4	110.0	76%	765.7	14%
libpng	562.0	117.2	79%	221.9	53%

Table 3.3: Summary sizes. S_{basic} is the size of the basic summary and S_{opt} is the size of the optimized summary. S_{obj} is the size of the library object code.

Clearly, the cost of the fragment points-to analysis and the cost of the summary construction are low. Even for the larger libraries (around 20K LOC) the running time is practical. These results indicate that both the fragment points-to analysis and the summary construction algorithm are good candidates for inclusion in realistic compilers and software engineering tools.

Our second set of experiments investigated the difference between the basic summary and the optimized summary. We first compared the sizes of the two summaries, as shown in Table 3.3. The fourth column in the table shows the reduction in summary size due to the optimizations described in Section 3.3.3. This reduction was between 47% and 79% (69% on average), which indicates that the optimizations can be very effective in producing a more compact summary. The last two columns in Table 3.3 compare the size of the optimized summary and the size of the library object code. The summary size was between 14% and 76% (43% on average) of the size of the object code, which shows that the space overhead of storing the summary is practical.[6]

Next, we measured the differences between the basic summary and the optimized summary with respect to the cost of the summary-based fragment points-to analysis of the client module. The results are shown in Table 3.4. The order of the programs in the table is based on the relative size of the library, as shown by the percentages in the last column

[6]The sizes depend on the file format used to store the summaries. We use a simple text-based format; more optimized formats could further reduce summary size.

Program	T_{basic} (sec)	Δ_T	S_{basic} (MB)	Δ_S
gnuplot	47.3	4%	79.6	5%
povray	111.6	17%	146.7	16%
unzip	21.0	25%	39.5	16%
fudgit	39.8	21%	59.5	17%
gasp	9.7	32%	18.6	26%
cjpeg	15.7	59%	32.1	56%
tiff2ps	22.5	61%	37.5	59%
bzip2	5.4	69%	13.0	54%

Table 3.4: Cost of the fragment points-to analysis of the client module. T_{basic} is the analysis time with the basic summary, and Δ_T is the reduction in analysis time when using the optimized summary. S_{basic} and Δ_S are the corresponding measurements for analysis memory.

of Table 3.1. Column T_{basic} shows the analysis time when using the basic summary. Column Δ_T shows the reduction in analysis time when using the optimized summary. The reduction is proportional to the relative size of the library. For example, in `bzip2` the majority of the program is in the library, and the cost reduction is significant. In `gnuplot` only 6% of the pointer-related statements are in the library, and the analysis cost is reduced accordingly. Column Δ_S shows the reduction in the memory usage of the analysis. Similarly to Δ_T, the space reduction is proportional to the relative size of the library.

The results from these experiments clearly show that the optimizations from Section 3.3.3 can have significant beneficial impact on the size of the summary and on the cost of the subsequent summary-based points-to analysis.

Chapter 4

Class Analysis for Testing of Polymorphism in Java

In this chapter we discuss the design and use of fragment analysis for the purposes of testing of polymorphism in Java software.

4.1 Testing of Polymorphism

Testing of object-oriented software presents new challenges due to features such as inheritance, polymorphism, dynamic binding, and object state [6]. Programs contain complex interactions among sets of collaborating objects from different classes. These interactions are greatly complicated by object-oriented features such as *polymorphism*, which allows the binding of an object reference to objects of different classes. While this is a powerful mechanism for producing compact and extensible code, it creates numerous fault opportunities [6].

Code that uses polymorphism can be hard to understand and therefore fault-prone—for example, understanding all possible interactions between a message sender and a message receiver under all possible bindings can be challenging for programmers. The sender of a message may fail to meet all preconditions for all possible bindings of the receiver [7]. A subclass in an inheritance hierarchy may violate the contract of its superclasses; clients that send polymorphic messages to this hierarchy may experience inconsistent behavior. For example, an inherited method may be incorrect in the context of the subclass [46], or an overriding method may have preconditions and postconditions different from the ones for the overridden method [6]. In deep inheritance hierarchies, it is easy to forget to override methods for lower-level subclasses [16]; clients of such hierarchies may experience incorrect behavior for some receiver classes. Changes in receiver classes may cause tested

```
class A { public void m() {..} }
class B extends A { public void m() {..} }
class C extends A {..}
A a;
.....
c_i: a.m(); //  a may refer to instances of A, B, or C
            //  RC(c_i) = {A,B,C}     TM(c_i) = {A.m,B.m}
```

Figure 4.1: Example of *RC* and *TM* coverage criteria

and unchanged client code to fail [7].

4.1.1 Coverage Criteria

Various techniques for testing of polymorphic interactions have been proposed in previous work [64, 41, 40, 43, 11, 7, 3]. These approaches require testing that exercises *all possible polymorphic bindings* for certain elements of the tested software. Such requirements can be encoded as coverage criteria. A *coverage criterion* is a program-based (structural, white-box) test adequacy criterion [70] that defines testing requirements in terms of the coverage of particular elements in the structure of the tested software. Coverage criteria can be used to evaluate the adequacy of the performed testing and can also provide guidelines for additional testing that leads to higher coverage.

In our work we focus on two coverage criteria for testing of polymorphism. The *receiver-classes* criterion (denoted by *RC*) requires exercising of all possible classes of the receiver object at a call site [41, 40, 43, 11]. The *target-methods* criterion (denoted by *TM*) requires exercising of all possible bindings between a call site and the methods that may be invoked by that site [64, 7].[1] Clearly, *RC* subsumes *TM*. For example, consider the Java classes in Figure 4.1, and suppose that reference variable `a` may refer to instances of classes `A`, `B`, or `C`. The *RC* criterion requires testing of call site `a.m()` with each of the three possible classes of the receiver object. Similarly, the *TM* criterion requires testing that invokes each of the two possible target methods (i.e., both `A.m` and the overriding `B.m`).

[1]Other coverage criteria for polymorphism are also possible. For example, in addition to *RC*, [41] proposes coverage of all possible classes for the senders and the parameters of a message. Our work can be trivially extended to handle such criteria.

The testing requirements encoded by the above criteria have been advocated by several authors [64, 41, 40, 43, 11, 7]. For example, Binder points out that "just as we would not have high confidence in code for which only a small fraction of the statements or branches have been exercised, high confidence is not warranted for a client of a polymorphic server unless all the message bindings generated by the client are exercised" [7]. There is existing evidence that such criteria are better suited for detecting object-oriented faults that the traditional statement and branch coverage criteria [11].

4.1.2 Coverage Tools and Class Analysis

The use of coverage criteria is essentially impossible without coverage tools. A *coverage tool* (1) analyzes the tested software to determine what elements need to be covered, (2) inserts instrumentation that allows coverage tracking, (3) executes the test cases, and (4) reports the degree of coverage and the elements that have not been covered. In order to determine what software elements need to be covered, a coverage tool has to use some form of *source code analysis*. Such an analysis computes the set of elements for which coverage should be tracked, and determines the kind and location of all necessary code instrumentation.

For simple criteria such as statement and branch coverage, the necessary source code analysis is trivial. However, the *RC* and *TM* criteria require a more complex analysis. For each call site, this analysis should determine the possible classes of the receiver object and the possible target methods. The simplest approach for achieving this goal is to traverse the class hierarchy. For example, for the call site `a.m()` in Figure 4.1, the type of reference variable `a` is `A`. By transitively traversing all subclasses of `A`, we can determine that the set of possible receiver classes is {`A`, `B`, `C`}, and the set of possible target methods is {`A.m`, `B.m`}. While not explicitly stated, it appears that all previous work [64, 41, 40, 43, 11, 7] uses this approach to determine the coverage requirements.

It is well known that using the class hierarchy to determine possible receiver classes is often overly conservative. For example, suppose that call `a.m()` is immediately preceded by statement `a = new B()`. In this case, source code analysis based on the class hierarchy will produce infeasible testing requirements for receiver classes {`A`,`C`} and for target method `A.m`. This problem can be addressed by using *class analysis*. Class analysis is a static program analysis that determines the classes of all objects to which a given reference variable may point. While initially developed in the context of optimizing compilers for object-oriented languages, class analysis also has a variety of applications in software engineering tools. We are interested in using class analysis to determine the coverage requirements for testing of polymorphism in object-oriented software. Thus, for each call site $x.m()$, we will use the class analysis solution for x to determine the possible receiver classes and the possible target methods.

All class analyses are conservative—that is, they are guaranteed to report all classes that could be actually observed at run time. In addition, class analyses may (and do) report infeasible classes. More precise analyses produce fewer infeasible classes, but tend to be more expensive. There is a large body of work on various class analyses with different tradeoffs between cost and precision [44, 1, 2, 47, 19, 5, 28, 18, 10, 48, 52, 61, 65, 62, 38, 51, 27, 42]. However, there has been no previous work on using these analyses for the purposes of testing of polymorphism.

4.1.3 Open Problems

The goal of our work is to investigate the use of class analysis for computing the *RC* and *TM* criteria in coverage tools. In this context, there are two open problems that need to be addressed: how to perform *class analysis of partial programs*, and how to ensure that *the class analysis is precise*.

Analysis of Partial Programs

The large body of existing work on class analysis focuses on the problem of performing *whole-program class analysis*. However, testing is rarely done only on completed programs—many testing activities are performed on partial programs. Any realistic coverage tool should be able to work on partial programs, and therefore cannot incorporate a whole-program class analysis. Clearly, what is needed here is some kind of *fragment* *class analysis*. Later in this chapter we show how to construct such fragment analyses from existing whole-program class analyses, based on the general methodology described in Chapter 2.

Analysis Precision

We believe that analysis precision is a critical issue for the use of class analysis in realistic coverage tools. Less precise analyses compute less precise coverage criteria—in other words, some of the coverage requirements may be impossible to achieve. For example, a less precise analysis may report a large set of possible receiver classes at a call site, while in reality only a small subset of these classes is actually possible. Thus, regardless of the testing effort, high coverage can never be achieved. In the presence of such imprecision, the coverage metrics become hard to interpret: is the low coverage due to inadequate testing, or is it due to analysis imprecision? This problem seriously compromises the usefulness of the coverage metrics. In addition, the person that creates new test cases may spend significant time and effort trying to determine the appropriate test cases, before realizing that it is impossible to achieve the required coverage. This situation is unacceptable because human time and attention are much more expensive than computing time.

In order to justify the use of a particular class analysis in a coverage tool, we need to ensure that few (if any) infeasible requirements are generated by that analysis. Previous work on class analysis only addresses the issue of *relative* analysis precision (e.g., how much smaller is the solution computed by analysis Y, compared to the solution computed by analysis X). However, we are interested in *absolute* analysis precision: what part of the analysis solution is infeasible? To answer this question, in our experiments we determined manually the highest possible coverage achievable with actual test cases. The difference between the required and the actual coverage is due to analysis imprecision. We believe that this kind of precision metric is absolutely necessary for justifying the use of a particular class analysis in a coverage tool; however, to the best of our knowledge, such metrics of absolute precision are not available in any previous work on class analysis.

4.2 A Coverage Tool for Java

We have built a test coverage tool for Java that supports the *RC* and *TM* coverage criteria. In the context of this tool we have implemented and evaluated several class analyses. In the future we plan to use the tool as the basis for investigations of other problems related to the testing of polymorphism, and more generally, problems related to the testing of object-oriented software.

The input of the tool contains a set *Cls* of classes that will be tested, as well as a set *Int* of methods and fields from classes in *Cls*. These methods and fields define the *interface* of *Cls* that is currently being tested.[2] In general, *Int* could contain a small subset of all fields and methods from *Cls*; this corresponds to the case when the testing only targets a subset of the functionality provided by *Cls*. A *test suite for Int* is any arbitrary Java class that tests *Int* (i.e., calls interface methods and reads/writes interface fields) and does not access any methods/fields from *Cls* that are not in *Int*. We denote by $\mathit{AllSuites}(\mathit{Int})$ the set of all possible test suites for *Int*. We assume that *Cls* is closed with respect to *Int*: for any arbitrary test suite $S \in \mathit{AllSuites}(\mathit{Int})$, any class that could be referenced during the execution of S is included in *Cls*. In other words, we consider test suites that only test interactions among classes from the given set *Cls*. In general, classes from *Cls* could potentially interact with unknown classes from outside of *Cls* (e.g., with unknown future subclasses of $C \in \mathit{Cls}$); however, at the time the testing is performed, such interactions cannot be exercised and therefore we do not consider test suites whose execution involves such external classes.

In addition to *Cls* and *Int*, the tool takes as input one particular test suite $T \in \mathit{AllSuites}(\mathit{Int})$. As output, the tool reports the coverage achieved by T with respect to the *RC* and *TM* criteria.

There are four tool components. The *analysis component* processes the classes in *Cls* and computes the requirements according the *RC* and *TM* criteria (i.e., for each call site c, it produces sets $RC(c)$ and $TM(c)$). More precisely, the analysis answers the following

[2]We use "interface" to refer to the software engineering concept of an interface, not the `interface` construct in Java. For our purposes, a Java `interface` is treated as an abstract Java class.

```
package station;
public abstract class Link
      { public abstract void transmit(String message); }
class NormalLink extends Link { ...  }
class PriorityLink extends Link { ...  }
class SecureLink extends Link { ...  }
class LoggingLink extends Link { ...  }

public class Station {
   private Link link = new NormalLink();
   private int message_id = 0;
   public void sendMessage(String m) {
  c1: link.transmit(message_id++ + " " + m);
      if (message_id == 10) link = new PriorityLink(); }
   public void report(Link l) { c2: l.transmit("id = " + message_id); } }

public class Factory {
   private boolean secure = false;
   public Link getLink() {
      if (secure) return new SecureLink();
      else        return new NormalLink(); }
   public void setSecure() { secure = true; } }
```

Figure 4.2: Package `station` with two polymorphic call sites c_1 and c_2.

question: For each call site, what may be the receiver classes and target methods with respect to all possible $S \in \mathit{AllSuites}(\mathit{Int})$? In other words, if it is possible to write some test suite that tests *Int* and exercises a call site $c \in \mathit{Cls}$ with some receiver class X or some target method m, the analysis should include X in $RC(c)$ and m in $TM(c)$. These computed coverage requirements are supplied to the *instrumentation component*, which inserts instrumentation at call sites to record the classes of the receiver objects at run time (using the reflection mechanism in Java). Instrumentation is only inserted at polymorphic call sites—i.e., sites c for which $RC(c)$ is not a singleton set. The instrumented code is supplied to the *test harness* which automatically runs the given test suite T. The results of the execution are processed by the *reporting component*, which determines the actual coverage achieved at call sites.

Example. Consider package `station` in Figure 4.2. Class `Station` models a station that connects to the rest of the system using a variety of links. Initially, messages are

```
package harness;
public abstract class TestSuite
   { public abstract void run(); }

package stationtest;
import station.*;
public class StationTestSuite extends harness.TestSuite {
  public void run() {
    Station s = new Station();
    Factory f = new Factory();
    Link l;
    for (int i=0; i < 10; i++) {
       s.sendMessage("message " + i);
       l = f.getLink();
       s.report(l); } } }
```

Figure 4.3: Simplified test suite for package `station`. This suite achieves only 50% RC coverage for call sites c_1 and c_2 from class `Station`.

transmitted using a normal-priority link. After certain number of messages have been processed, the station starts using a high-priority link. In addition, the station may be required to report its current state on some link provided from the outside. External code may use class `Factory` to gain access to normal or secure links.

Suppose that we are interested in testing the functionality that package `station` provides to non-package client code. In this case *Int* contains `Station.sendMessage`, `Station.report`, `Factory.getLink`, `Factory.setSecure`, and `Link.transmit` (plus the constructors of `Station` and `Factory`). Given the package and *Int*, the tool computes sets $RC(c_i)$ and $TM(c_i)$ for the call sites in `Station`. For example, using one of the class analyses presented later in the chapter, the analysis component may produce sets $RC(c_1) = \{\texttt{NormalLink}, \texttt{PriorityLink}\}$ and $RC(c_2) = \{\texttt{NormalLink}, \texttt{SecureLink}\}$ with the corresponding sets $TM(c_i)$. Given this information, the instrumentation component inserts instrumentation at the two call sites. At run time this instrumentation records the classes of the receiver objects using method `Object.getClass`.

Suppose that the tool is used to evaluate the test suite from package `stationtest` shown in Figure 4.3. The test harness automatically loads and executes the test suite, and then the reporting component provides the coverage results to the tool user. In this particular case, the test suite achieves 50% coverage for call site c_1 because the site is never executed with receiver class `PriorityLink`. Similarly, the coverage for c_2 is 50% because receiver class `SecureLink` is not exercised. Note that the suite achieves 100% statement and branch coverage for class `Station`, but this is not enough to achieve the necessary coverage of the polymorphic calls inside the class. To achieve 100% coverage for c_1 and c_2, we need to add at least one more iteration to the loop in `StationTestSuite`, and we also need to introduce a call `f.setSecure()`.

4.3 Fragment Class Analysis

In this section we describe a general method for constructing fragment class analyses for the purposes of testing of polymorphism in Java. This method can be applied to a large number of existing whole-program class analyses [2, 5, 28, 18, 48, 61, 62, 65, 38, 51, 27, 42] in order to derive fragment class analyses from them. The fragment analyses constructed with this method can be used in coverage tools to compute the requirements of the *RC* and *TM* coverage criteria.

Our approach is designed to be used with existing (and future) whole-program flow-insensitive class analyses. *Flow-insensitive* class analyses do not take into account the flow of control within a method, which makes them less costly than flow-sensitive analyses. The approach is applicable both to context-insensitive and to context-sensitive analyses. *Context-insensitive analyses* do not attempt to distinguish among the different invocation contexts of a method. This category includes Rapid Type Analysis (RTA) by Bacon and Sweeney [5], the XTA/MTA/FTA/CTA family of analyses by Tip and Palsberg [65], Declared Type Analysis and Variable Type Analysis by Sundaresan et al. [62], the p-bounded and p-bounded-linear-edge families of class analyses due to DeFouw et al. [18, 27], 0-CFA [58, 27], 0-1-CFA [28], Steensgaard-style points-to analyses [48, 38], and Andersen-style points-to analyses [61, 38, 51]. Our approach can be applied to all of these context-insensitive whole-program class analyses.

Context-sensitive analyses attempt to distinguish among different invocation contexts

Program $\rightarrow$ MainMethod ClassDecl$^+$
MainMethod $\rightarrow$ **main() {** Body **}**
ClassDecl $\rightarrow$ **class** ClassId [**extends** ClassId]
    { MemberDecl* **}**
MemberDecl $\rightarrow$ FieldDecl | ConstructorDecl |
    MethodDecl
FieldDecl $\rightarrow$ Type FieldId
ConstructorDecl $\rightarrow$ ClassId**()** **{** Body **}**
MethodDecl $\rightarrow$ (Type|**void**) MethodId(FormalDecl*)
    { Body **}**
FormalDecl $\rightarrow$ Type FormalId
Body $\rightarrow$ LocalDecl* Stmt$^+$

LocalDecl $\rightarrow$ Type LocalId
Var $\rightarrow$ LocalId | FormalId | **this**
Stmt $\rightarrow$ **if (..)** Stmt$^+$ **else** Stmt$^+$
    | **switch (..)** Stmt$^+$
    | **while (..)** Stmt$^+$
    | Var = Var
    | Var = Var**.**FieldId
    | Var**.**FieldId = Var
    | [Var =] Var**.**MethodId(Var*)
    | Var = **new** ClassId()
    | Var = (Type) Var
    | ReturnVarId = Var
Type $\rightarrow$ ClassId

Figure 4.4: Grammar for the simplified Java-like language. Terminals are shown in boldface. Each method has an auxiliary variable (represented by ReturnVarId) that is assigned all values returned by the method.

of a method. As a result, such analyses are potentially more precise and more expensive than context-insensitive analyses. In *parameter-based* context-sensitive class analyses, calling context is modeled by using some abstraction of the values of the actual parameters at a call site. *Call-chain-based* context-sensitive class analyses represent calling context using a vector of k enclosing call sites. Our approach can be applied both to parameter-based analyses (e.g., the Cartesian Product algorithm due to Agesen [2], the Simple Class Set algorithm by Grove et al. [28], and the parameterized object-sensitive analyses by Milanova et al. [42]) and to call-chain-based analyses (e.g., the standard k-CFA analyses [58, 27], as well as the k-1-CFA analyses by Grove et al. [28, 27]).

4.3.1 Simplified Language

For the purpose of this presentation, we consider the Java-like language described in Figure 4.4. We use this simplified language to make the formal description shorter and more readable. Our approach can be trivially extended to handle other language features of Java (e.g., non-default constructors, static methods/fields, interfaces, arrays, etc.). The actual implementations of fragment class analyses for our experiments handle the entire Java language.

4.3.2 Structure of Fragment Class Analysis

Recall that the input to the tool is a set of classes *Cls*, as well as a set *Int* of methods and fields that defines the interface of *Cls* that is currently being tested. A test suite for *Int* is any arbitrary Java class that tests *Int* (i.e., calls interface methods and reads/writes interface fields) and does not access any methods/fields from *Cls* that are not in *Int*. Let *AllSuites*(*Int*) be the set of all possible test suites for *Int*.

The goal of the tool is to compute the requirements according to the *RC* and *TM* criteria (i.e., for each call site c, to produce sets $RC(c)$ and $TM(c)$). For this, it needs to answer the following question: For each method call, what may be the receiver classes and target methods with respect to all possible $S \in AllSuites(Int)$? More precisely, if it is possible to write some test suite that tests *Int* and exercises a call site $c \in Cls$ with some receiver class X or some target method m, the tool should include X in $RC(c)$ and m in $TM(c)$.

To compute $RC(c)$ and $TM(c)$, the tool needs to use fragment class analysis. We define an entire family of such class analyses in the following way: first, we create *placeholders* that simulate the effects of the unknown code from all possible test suites. The placeholders are added to the tested classes, the result is treated as a complete program, and *the solution engine of some whole-program class analysis* is applied to it. In Section 4.5 we prove the correctness of this method with respect to all whole-program class analyses listed in the beginning of this section. It is important to note that the created placeholders are *not* designed to be executed as an actual test suite; they are only used for the purposes of the fragment class analysis.

4.3.3 Placeholders

In our approach we create a placeholder `main` method that contains a variety of placeholder statements, as shown in Figure 4.5. For each class $X \in Cls$, there is a placeholder variable *ph_X* that serves as a representative for all unknown external reference variables of type X. Different placeholder statements represent different kinds of statements that could occur in the unknown code. For example, `ph_X = new X()` represents the fact that the

```
main() {
    // placeholder variable ph_X for every class X ∈ Cls
    X ph_X;
    // for every class X whose constructor is in Int
    ph_X = new X();
    // for every field f ∈ Int declared in class X with type Y
    ph_Y = ph_X.f; ph_X.f = ph_Y;
    // for every method m ∈ Int declared in class X with signature W m(Y,..,Z)
    ph_W = ph_X.m(ph_Y,..,ph_Z);
    // for every subclass Y of class X
    ph_X = ph_Y; ph_Y = (Y)ph_X;
}
```

Figure 4.5: Placeholder method and placeholder statements.

unknown external code may create instances of X and assign them to reference variables of type X. Some of the placeholder statements represent the effect of accessing fields and methods from *Int*. Finally, the last two categories of placeholder statements represent the possible effects of assigning variables of one type to variables of another type (including the possible effects of casting).

Example. Consider package `station` in Figure 4.2. Suppose that we are interested in testing the functionality that this package provides to non-package client code. In this case *Int* contains `Station.sendMessage`, `Station.report`, `Factory.getLink`, `Factory.setSecure`, and `Link.transmit` (plus the constructors of classes `Station` and `Factory`). Given the package and *Int*, the fragment analysis creates the placeholders shown in Figure 4.6. These placeholders are added to `station` and the result is analyzed using the engine of some whole-program class analysis. In Section 4.4 we present examples of the solutions computed by two such whole-program analysis engines.

4.4 Precision of Fragment Class Analysis

The approach presented above allows us to construct safe fragment class analyses from a large number of existing (and future) whole-program class analyses. The quality of the information produced by the fragment analyses depends on the underlying whole-program analysis engine.

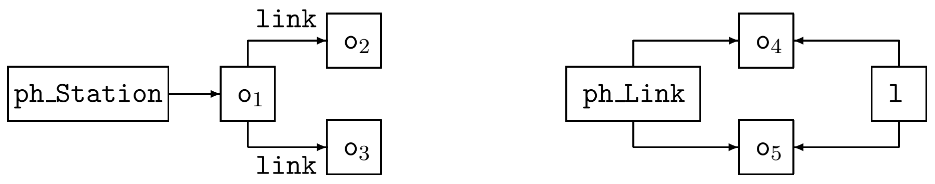
Consider package `station` in Figure 4.2. If we simply examine the class hierarchy to

```
import station;
main() {
      Station ph_Station;
      Factory ph_Factory;
      Link ph_Link;
      String ph_String;
      ph_Station = new Station();
      ph_Factory = new Factory();
      ph_String = new String();
      ph_Station.sendMessage(ph_String);
      ph_Station.report(ph_Link);
      ph_Link = ph_Factory.getLink();
      ph_Factory.setSecure();
      ph_Link.transmit(ph_String);
}
```

Figure 4.6: Placeholders for package `station` in the case when *Int* contains all methods visible outside of the package.

determine the possible receiver objects at call sites, we would have to conclude that $RC(c_i)$ contains all four subclasses of `Link`, which is too conservative and will result in infeasible testing requirements. In fact, the tool will never report more than 50% coverage for the two call sites in `Station`, even if in reality the achieved coverage is 100%.

Now suppose that we add the placeholders from Figure 4.6 and we run the engine of Rapid Type Analysis (RTA) [5]. RTA is a popular whole-program class analysis that performs class analysis and call graph construction in parallel. It maintains a worklist of methods reachable from `main`, and a set of classes instantiated in reachable methods. In the final solution, the set of classes for a variable `v` is the set of all instantiated subclasses of the declared type of `v`. In this example, RTA determines that class `Factory` is instantiated in `main`. This implies that call site `ph_Factory.getLink()` may be executed with an instance of `Factory`, which means that method `getLink` is reachable from `main`. By processing the body of `getLink`, RTA determines that `NormalLink` and `SecureLink` are instantiated. Similarly, because `Station` is instantiated in `main`, the analysis determines that `sendMessage` is reachable, which implies that `PriorityLink` may also be instantiated. At the end, RTA determines that the only instantiated subclasses of `Link` are `NormalLink`, `PriorityLink`, and `SecureLink`, and therefore $RC(c_i)$ contains only these three classes.



$o_1 \Rightarrow$ `new Station()` in `main`
$o_2 \Rightarrow$ `new NormalLink()` in `Station`
$o_3 \Rightarrow$ `new PriorityLink()` in `Station`
$o_4 \Rightarrow$ `new SecureLink()` in `Factory`
$o_5 \Rightarrow$ `new NormalLink()` in `Factory`

Figure 4.7: Some points-to edges computed by Andersen's analysis.

Unlike analysis of the class hierarchy, RTA is capable of filtering out infeasible receiver class `LoggingLink`. Still, some imprecision remains because infeasible class `SecureLink` is reported for c_1 and infeasible class `PriorityLink` is reported for c_2.

As another example, suppose that we use the solution engine of Andersen's whole-program points-to analysis for Java [61, 38, 51]. This engine constructs a *points-to graph* in which the nodes represent reference variables and objects, and the edges represent points-to relationships between the nodes. Figure 4.7 shows some of the edges in the points-to graph computed for our example. (Full description of the analysis and the computed points-to graph is beyond the scope of this presentation.) From this graph it is clear that $RC(c_1)$ contains `NormalLink` and `PriorityLink`, while $RC(c_2)$ contains `NormalLink` and `SecureLink`. Because the analysis engine is more powerful than RTA, it is capable of filtering out the additional infeasible receiver classes. Table 4.1 summarizes the solutions computed by the different analyses.

It is important to note that any class analysis could potentially compute infeasible classes. In this particular case, every receiver class reported by Andersen's analysis is feasible, but in general this need not be true. To realistically evaluate the quality of different analyses for the purposes of testing of polymorphism, for every reported receiver class we need to determine if this class is actually feasible—that is, if there exists some test suite that exercises this receiver class. Only analyses that report few (if any) infeasible classes should be used in coverage tools; otherwise, the coverage metrics become hard to interpret, and tool users may waste time and effort trying to satisfy infeasible testing requirements. We obtained such precision metrics during the experiments presented in

	NormalLink		PriorityLink		SecureLink		LoggingLink	
	c_1	c_2	c_1	c_2	c_1	c_2	c_1	c_2
Class Hierarchy	●	●	●	●	●	●	●	●
RTA	●	●	●	●	●	●		
Andersen	●	●	●			●		
Feasible	●	●	●			●		

Table 4.1: Sets $RC(c_1)$ and $RC(c_2)$ computed by the fragment class analyses. The last row shows the receiver classes that are actually feasible.

Section 4.6.

4.5 Safety of Fragment Class Analysis

The goal of this section is to prove the safety of the fragment analyses constructed with our approach. In particular, we are interested in proving the safety of any fragment class analysis that uses the solution engine of one of the whole-program analyses listed in the beginning of Section 4.3. All of these whole-program analyses are instantiations of a general framework for class analysis defined by Grove et al. [28, 27].

To prove the safety of the corresponding fragment analyses, we use the following approach. First, we define two particular whole-program analyses that are instantiations of the framework from [28, 27]. The first analysis, denoted by $\mathcal{A}_p$, is a parameter-based context-sensitive analysis similar to Agesen's Cartesian Product algorithm [2]. The second analysis, denoted by $\mathcal{A}_c$, is a call-chain-based context-sensitive analysis similar to the k-1-CFA analysis from [28]. These two analyses are relatively precise instantiations of the framework from [28, 27] and they represent two points at the high end of the precision spectrum for context-sensitive class analysis (with parameter-based sensitivity in $\mathcal{A}_p$ and call-chain-based sensitivity in $\mathcal{A}_c$). As discussed below, by proving safety with respect to these two analyses, we can guarantee safety with respect to *any instantiation* of the framework from [28, 27] that is less precise than $\mathcal{A}_p$ or $\mathcal{A}_c$.

After defining $\mathcal{A}_p$ and $\mathcal{A}_c$, we prove that the corresponding fragment class analyses are safe. Let $\mathcal{A}'_p$ be the fragment class analysis that creates the placeholders from Figure 4.5 and then runs the solution engine of $\mathcal{A}_p$. Similarly, let $\mathcal{A}'_c$ be the fragment class analysis that uses the engine of $\mathcal{A}_c$. In Section 4.5.2 we prove that both $\mathcal{A}'_p$ and $\mathcal{A}'_c$ are safe fragment analyses.

Consider an arbitrary whole-program class analysis $\mathcal{A}$ that is less precise than $\mathcal{A}_p$ or $\mathcal{A}_c$—that is, $\mathcal{A}$ always computes a solution that is a *superset* of the solution computed by $\mathcal{A}_p$ or by $\mathcal{A}_c$. Based on the safety of $\mathcal{A}'_p$ and $\mathcal{A}'_c$, it is easy to show that the fragment analysis that uses the solution engine of $\mathcal{A}$ is also safe. Because of the properties of the framework from [28, 27], each of the whole-program analyses listed in the beginning of Section 4.3 is either less precise than $\mathcal{A}_p$, or less precise than $\mathcal{A}_c$; this implies the correctness of our approach for all of these existing whole-program analyses. Furthermore, this result means that in the future our approach can be applied to any new whole-program class analysis that is less precise than $\mathcal{A}_p$ or $\mathcal{A}_c$.

4.5.1 Whole-program Analyses $\mathcal{A}_p$ and $\mathcal{A}_c$

The whole-program class analyses are defined in terms of three sets. Set V contains all variables in the analyzed program, as defined by the non-terminal `Var` in Figure 4.4. Set O contains names for all objects created at object allocation sites (i.e., sites of the form `v = new C()`). For each allocation site we use a separate object name $o_i \in O$. Set F contains all fields in the program.

The analyses consider different abstractions of the calling context of a method. Analysis $\mathcal{A}_p$ defines and uses a set of contexts $\mathcal{C} = \{\epsilon\} \cup O \cup O^2 \cup O^3 \cup \ldots$. Each context is a tuple of object names. For a method that has formal parameters $f_1, f_2, \ldots, f_n$ (where f_1 is the implicit parameter `this`), context $(o_1, o_2, \ldots, o_n) \in \mathcal{C}$ represents invocations of the method when formal parameter f_i points to o_i. The empty context ϵ represents the invocation of `main`.

Analysis $\mathcal{A}_c$ represents calling context with a vector of at most k enclosing call sites. Let *CallSites* be the set of all call sites in the program. The set of contexts is defined as $\mathcal{C} = \{\epsilon\} \cup \mathit{CallSites} \cup \mathit{CallSites}^2 \cup \ldots \cup \mathit{CallSites}^k$. For any method m, context $(s_1, s_2, \ldots, s_n) \in \mathcal{C}$ represents invocations of m from call site s_1 when the method containing s_1 is invoked from call site s_2, etc. Again, the empty context ϵ represents the invocation of `main`.

To distinguish among invocations of the same method under different contexts, the analyses create multiple copies of formal parameters and local variables. Each variable $v \in V$ is replicated for each of the possible contexts of the method that declares v. We will use v^c to denote the replica of v for context $c \in \mathcal{C}$.

The analyses constructs *points-to graphs* containing two kinds of edges. Edge $(v^c, o) \in (V \times \mathcal{C}) \times O$ shows that variable v may point to object o when the method declaring v is invoked with context c. Edge $(o_i.f, o_j) \in (O \times F) \times O$ shows that field f of object o_i may point to object o_j. The elements in the analysis lattices L are points-to graphs. As usual, the partial order $\leq$ in L is defined as $G_1 \leq G_2$ iff $G_1 \supseteq G_2$, and the meet operation $\wedge$ is $G_1 \wedge G_2 = G_1 \cup G_2$.

The analyses associate a transfer function $f : L \rightarrow L$ with each statement in the program; this function encodes the semantics of the statement. In addition, for each method m the analyses maintain a set $\mathcal{C}_m \subseteq \mathcal{C}$ of contexts that have been observed at calls to m. In the beginning of both analyses, $\mathcal{C}_{main} = \{\epsilon\}$ and $\mathcal{C}_m = \emptyset$ for all other methods m. The transfer functions for assignment statements have the following form (m is the method containing the statement):

- for $p = new\ C()$: $f(G) = G \cup \bigcup_{c \in \mathcal{C}_m} \{(p^c, o_i)\}$, where o_i corresponds to $new\ C()$
- for $p = q$: $f(G) = G \cup \bigcup_{c \in \mathcal{C}_m} \{(p^c, o) \mid (q^c, o) \in G\}$
- for $p = q.f$: $f(G) = G \cup \bigcup_{c \in \mathcal{C}_m} \{(p^c, o) \mid (q^c, o_2) \in G \wedge (o_2.f, o) \in G\}$
- for $p.f = q$: $f(G) = G \cup \bigcup_{c \in \mathcal{C}_m} \{(o_1.f, o_2) \mid (p^c, o_1) \in G \wedge (q^c, o_2) \in G\}$
- for $p = (T)q$: $f(G) = G \cup \bigcup_{c \in \mathcal{C}_m} \{(p^c, o) \mid (q^c, o) \in G \wedge \mathit{compatible}(o, T)\}$

Each of the above transfer functions considers all contexts that have been observed at calls to the method m containing the statement. For each such context $c \in \mathcal{C}_m$, the corresponding context replicas of formal parameters and local variables are processed according to the semantics of the statement. A statement of the form "$p = new\ C()$" creates a new edge (p^c, o_i), where o_i is a unique object name corresponding to this particular object allocation site. Other assignment statements have similar effects by creating new edges.

For the last statement, an edge (p^c, o) is created only if object o could be casted to type T according to the casting rules in the Java language.

The transfer function for a call statement "$r = p.m(q_1, \ldots, q_n)$" encodes the context-sensitivity of the analysis. For the parameter-based analysis $\mathcal{A}_p$, the transfer function has the form

$$f(G) = G \cup \bigcup_{c \in \mathcal{C}_m} \{resolve(G, m, o_{rcv}, o_1, o_2, \ldots, o_n, r^c) \mid (p^c, o_{rcv}) \in G \wedge (q_i^c, o_i) \in G\}$$

where *resolve* is defined as follows:

```
$resolve(G, m, o_{rcv}, o_1, o_2, \ldots, o_n, r^c)$
    let $c_2 = (o_{rcv}, o_1, o_2, \ldots, o_n)$
    let $m_j(this, f_1, \ldots, f_n, ret) = dispatch(o_{rcv}, m)$
    add $c_2$ to $\mathcal{C}_{m_j}$
    return $\{(this^{c_2}, o_{rcv}), (f_1^{c_2}, o_1), \ldots, (f_n^{c_2}, o_n)\} \cup \{(r^c, o) | (ret^{c_2}, o) \in G\}$
```

At the call site, the analysis considers all possible tuples of objects that are pointed to by p and q_i; each such tuple creates a separate calling context. For each context, *resolve* determines the method m_j that is actually invoked at run time for receiver object o_{rcv}. The analysis then updates $\mathcal{C}_{m_j}$ and processes the necessary context copies of *this* and formal parameters f_i. Finally, the return value of m_j (stored in auxiliary variable *ret*, as described in Section 4.3.1) is propagated back to the call site.

The transfer function used by the call-chain-based analysis $\mathcal{A}_c$ has the form

$$f(G) = G \cup \bigcup_{c \in \mathcal{C}_m} \{resolve(G, m, c, s, o_{rcv}, q_1^c, q_2^c, \ldots, q_n^c, r^c) \mid (p^c, o_{rcv}) \in G\}$$

where $s \in CallSites$ is the call site. Function *resolve* is defined as follows:

```
$resolve(G, m, c, s, o_{rcv}, q_1^c, q_2^c, \ldots, q_n^c, r^c)$
    let $c_2 = prepend_k(s, c)$
    let $m_j(this, f_1, \ldots, f_n, ret) = dispatch(o_{rcv}, m)$
    add $c_2$ to $\mathcal{C}_{m_j}$
    return $\{(this^{c_2}, o_{rcv})\} \cup \{(f_i^{c_2}, o) | (q_i^c, o) \in G\} \cup \{(r^c, o) | (ret^{c_2}, o) \in G\}$
```

At the call site, the analysis processes separately each of the possible receiver objects and determines the method m_j invoked for each receiver o_{rcv}. Function $prepend_k(s, c)$

creates a new context by adding call site s to the beginning of call chain c. If the resulting call chain has $k+1$ elements (where k is a parameter of the analysis), the last element of the chain is removed. This is a standard approach for ensuring that the analysis only considers call chains with length at most k.

Both $\mathcal{A}_p$ and $\mathcal{A}_c$ start with an empty points-to graph and apply the transfer functions for program statements until no more edges can be added to the graph. For any variable $v \in V$, the final class analysis solution is

$$\mathit{Classes}(v) = \{\, X \mid \mathit{DeclMethod}(v) = m \wedge c \in \mathcal{C}_m \wedge (v^c, o) \in G_{\mathit{final}} \wedge \mathit{ClassOf}(o) = X\}$$

4.5.2 Safety of Fragment Analyses $\mathcal{A}'_p$ and $\mathcal{A}'_c$

Let $\mathcal{A}'_p$ be the fragment class analysis that creates the placeholders from Figure 4.5 and then runs the solution engine of whole-program analysis $\mathcal{A}_p$. Similarly, let $\mathcal{A}'_c$ be the fragment class analysis that uses the solution engine of $\mathcal{A}_c$. In this section we prove the safety of these two fragment analyses.

Consider an arbitrary test suite $S \in \mathit{AllSuites}(\mathit{Int})$. Recall that by definition S is an arbitrary Java class that tests *Int* (i.e., calls interface methods and reads/writes interface fields) and does not access any methods/fields from *Cls* that are not in *Int*. Without loss of generality, we assume that S only contains method `main`; clearly, if S contains any other methods, it is trivial to inline them in `main`.

For any variable v declared in *Cls*, let $\mathit{Classes}_{\mathcal{A}_p}(v)$ be the solution for v computed by $\mathcal{A}_p$ for the whole program containing *Cls* and test suite S. Below we prove that the solution $\mathit{Classes}_{\mathcal{A}'_p}(v)$ computed by $\mathcal{A}'_p$ is a superset of $\mathit{Classes}_{\mathcal{A}_p}(v)$. Similarly, we prove that $\mathit{Classes}_{\mathcal{A}_c}(v) \subseteq \mathit{Classes}_{\mathcal{A}'_c}(v)$. Therefore, the two fragment analyses are safe: if it is possible to write some test suite that tests *Int* and exercises a call site $c \in \mathit{Cls}$ with some receiver class X, it is guaranteed that $\mathcal{A}'_p$ and $\mathcal{A}'_c$ will include X in the coverage criterion $RC(c)$.

Consider an arbitrary whole-program class analysis $\mathcal{A}$ that is less precise than $\mathcal{A}_p$. Clearly, for the fragment analysis $\mathcal{A}'$ derived from $\mathcal{A}$, the solution $\mathit{Classes}_{\mathcal{A}'}(v)$ is a superset of the solution computed by $\mathcal{A}'_p$. This implies the safety of $\mathcal{A}'$: if it is possible to write some test suite that tests *Int* and exercises a call site $c \in \mathit{Cls}$ with some receiver class X, $\mathcal{A}'$ will include X in $RC(c)$. Similarly, if whole-program analysis $\mathcal{A}$ is less precise than $\mathcal{A}_c$, the corresponding fragment analysis $\mathcal{A}'$ is safe.

Whole-program analyses $\mathcal{A}_p$ and $\mathcal{A}_c$ are instances of the general framework for class analysis defined by Grove et al. [28, 27]. All whole-program analyses listed in the beginning of Section 4.3 are also instances of that framework, and in fact are less precise instances than $\mathcal{A}_p$ and $\mathcal{A}_c$. Therefore, our approach for constructing fragment class analyses can be safely applied to *all of these existing whole-program analyses*. Furthermore, the approach can be applied to any future whole-program analysis that is less precise than $\mathcal{A}_p$ or $\mathcal{A}_c$ (e.g., to other instances of Grove's framework).

Parameter-based Fragment Analysis

To show the safety of $\mathcal{A}'_p$, we need to prove that $\mathit{Classes}_{\mathcal{A}_p}(v) \subseteq \mathit{Classes}_{\mathcal{A}'_p}(v)$ for any variable v declared in *Cls*. We start by defining several abstraction relations between elements of $\mathcal{A}_p$ and elements of $\mathcal{A}'_p$.

Let V_{Cls} be the set of all variables in *Cls*, and V' be V_{Cls} together with the placeholder variables. Similarly, let O_{Cls} be the set of all object names corresponding to object allocation sites in *Cls*, and O' be O_{Cls} together with the object names for allocation sites in placeholder statements. The fragment analysis uses a set of contexts $\mathcal{C}' = \{\epsilon\} \cup O' \cup (O' \times O') \cup \ldots$. The lattice elements in the fragment analysis are points-to graphs with edges $(v^{c'}, o_i) \in (V' \times \mathcal{C}') \times O'$ and $(o_i.f, o_j) \in (O' \times F) \times O'$, where F' is the set of all fields in *Cls*.

We can define several abstraction relations between elements of $\mathcal{A}_p$ and elements of $\mathcal{A}'_p$. Let V and O be the set of variables and the set of object names in $\mathcal{A}_p$, respectively. The following relations hold:

- $\alpha(v, v)$ for any $v \in V_{\mathit{Cls}}$
- $\alpha(v, ph\_X)$ for any $v \in (V - V_{\mathit{Cls}})$ of type X
- $\alpha(o, o)$ for any $o \in O_{\mathit{Cls}}$

- $\alpha(o, o')$ iff $o \in (O - O_{Cls})$, $o' \in O'$ is created at a placeholder statement, and $ClassOf(o) = ClassOf(o')$

- $\alpha(c, c')$ for $c = (o_1, \ldots, o_n) \in \mathcal{C}$ and $c' = (o'_1, \ldots, o'_n) \in \mathcal{C}'$ iff $\alpha(o_i, o'_i)$ for $1 \leq i \leq n$

- $\alpha(e, e')$ for $e = (v^c, o) \in (V \times \mathcal{C}) \times O$ and $e' = (w^{c'}, o') \in (V' \times \mathcal{C}') \times O'$ iff $\alpha(v, w) \wedge \alpha(c, c') \wedge \alpha(o, o')$

- $\alpha(e, e')$ for $e = (o_1.f, o_2) \in (O \times F) \times O$ and $e' = (o'_1.f, o'_2) \in (O' \times F') \times O'$ iff $\alpha(o_1, o'_1) \wedge \alpha(o_2, o'_2)$

- $\alpha(G, G')$ iff for every edge e in points-to graph G there exists an edge e' in points-to graph G' such that $\alpha(e, e')$

Intuitively, these definitions encode the idea that in the fragment analysis $\mathcal{A}'_p$, variables and object names from *Cls* are represented by themselves, while all unknown external variables and object names are represented by placeholder variables and placeholder object names, respectively. This abstraction is generalized for contexts, context replicas of variables, points-to edges, and points-to graphs.

Suppose we can prove that α holds between the final points-to graphs computed by the two analyses. It is easy to see that this implies $Classes_{\mathcal{A}_p}(v) \subseteq Classes_{\mathcal{A}'_p}(v)$ for any variable v declared in *Cls*, which guarantees analysis safety. To show that α holds between the final points-to graphs, we prove a particular property that relates the transfer functions in the whole-program analysis with the transfer functions in the fragment analysis. This property has the following form: suppose that

- α holds between points-to graphs G and G'

- for each method m and for each reaching context $c \in \mathcal{C}_m$ in the whole-program analysis, there exists a reaching context $c' \in \mathcal{C}'_m$ in the fragment analysis such that $\alpha(c, c')$

Then, for every transfer function f in the whole-program analysis, there exists a set of transfer functions $\{f'_1, \ldots, f'_k\}$ in the fragment analysis such that

- $\alpha(f(G), (f'_1 \circ \ldots \circ f'_k)(G'))$
- for any new reaching context c added to some $\mathcal{C}_m$ due to the application of f, the corresponding application of f'_i results in a set $\mathcal{C}'_m$ that contains a context c' such that $\alpha(c, c')$

Intuitively, the property ensures that the effects of any transfer function application in the whole-program analysis can be "simulated" by the fragment analysis, both in terms of creating new points-to edges and in terms of introducing new reaching contexts. A straightforward corollary of this property is that α holds between the final points-to graphs computed by $\mathcal{A}_p$ and $\mathcal{A}'_p$.

To prove the above property, we distinguish two cases. First, consider any statement in *Cls* with a transfer function f in the whole-program analysis and a transfer function f' in the fragment analysis. It is straightforward to show that $\alpha(f(G), f'(G'))$. Furthermore, it is easy to prove that for any call statement in *Cls*, the new reaching contexts created by the whole-program analysis are matched by corresponding calling contexts created by the fragment analysis.

Next, consider how the whole-program analysis processes a statement that is located outside of *Cls*. For each such statement, in the fragment analysis there exists a set of placeholder statements that "simulate" the effects of the external statement. For example, suppose that *Cls* contains a class A and a subclass B, and that the external statement is "$a = new\ B()$", where a is some external variable of type A. The effects of this statement are represented by the sequence of placeholder statements "$ph\_B = new\ B();\ ph\_A = ph\_B;$". In general, it can be proven that for each external statement with a transfer function f, there exist placeholder statements with transfer functions $f'_1, \ldots, f'_k$ such that $\alpha(f(G), (f'_1 \circ \ldots \circ f'_k)(G'))$. Furthermore, if f adds a new reaching context c to some $\mathcal{C}_m$, the application of f'_i adds to $\mathcal{C}'_m$ a context c' such that $\alpha(c, c')$.

Call-chain-based Fragment Analysis

The safety proof for $\mathcal{A}'_c$ is very similar to the proof for $\mathcal{A}'_p$. The analysis uses a set of contexts $\mathcal{C}' = \{\epsilon\} \cup \mathit{CallSites}' \cup (\mathit{CallSites}' \times \mathit{CallSites}') \cup \ldots$, where $\mathit{CallSites}'$ denotes the set of all call sites in *Cls* and in the placeholder method `main`. The abstraction relation for contexts can be defined as

- $\alpha(s, s)$ for any call site $s \in \mathit{Cls}$
- $\alpha(s, s')$ iff call site $s \notin \mathit{Cls}$ and s' is a placeholder call site with the same static target method as s
- $\alpha(c, c')$ for $c = (s_1, \ldots, s_n) \in \mathcal{C}$ and $c' = (s'_1, \ldots, s'_n) \in \mathcal{C}'$ iff $\alpha(s_i, s'_i)$ for $1 \leq i \leq n$

The abstraction relations for other analysis entities (variables, object names, etc.) are the same as for $\mathcal{A}'_p$. The correctness proof is also similar: we prove the same property as before, which implies that α holds between the final points-to graphs computed by $\mathcal{A}_c$ and $\mathcal{A}'_c$. Therefore, $\mathit{Classes}_{\mathcal{A}_c}(v) \subseteq \mathit{Classes}_{\mathcal{A}'c}(v)$ for any variable v declared in *Cls*, which guarantees the safety of the fragment analysis.

4.6 Empirical Results

For our experiments we used a set of Java packages including the standard packages `java.text` and `java.util.zip`, as well as the publicly available packages `gnu.math` (from `www.gnu.org/software/kawa`) and `com.lowagie.text` from the `iText` library for creating PDF files (`www.lowagie.com`). We then defined and performed several *testing tasks*. The goal of each task was to write a test suite that exercised some particular functionality provided by these packages. For example, one task exercised the functionality related to identifying boundaries in text (i.e., word boundaries, line boundaries, etc.), as provided by a set of classes from `java.text`. As another example, a task was designed to exercise the functionality from `java.util.zip` related to ZIP files. The first three columns in Table 4.2 briefly describe the testing tasks and the functionality they exercise.

For each task, we determined the set *Int* of interface methods and fields for the tested functionality. We then computed the set of methods that were directly or transitively reachable from the interface methods; in this reachability computation, the targets of method calls were resolved conservatively by considering the class hierarchy. Only such reachable methods could potentially be executed when the tested functionality is eventually

Task	Package	Functionality	#Classes	#PolySites
task1	`java.text`	boundaries in text	12	12
task2	`java.text`	formatting of numbers/dates	13	79
task3	`java.text`	text collation	12	2
task4	`java.util.zip`	ZIP files	8	5
task5	`java.util.zip`	ZIP output streams	8	18
task6	`gnu.math`	complex numbers	8	194
task7	`com.lowagie.text`	paragraphs in PDF docs	24	199
task8	`com.lowagie.text`	lists in PDF docs	24	169

Table 4.2: Description of testing tasks. Last two columns show the number of directly related classes and the number of polymorphic sites in these classes, according to the class hierarchy.

exercised by some client code. Among all classes that contained reachable methods, only some were *directly related* to the tested functionality; the rest of the classes were servers of these directly related classes. We considered each call site that was in a directly related class and for which there was more than one possible receiver class, according to the class hierarchy. Let *PolySites* denote the set of all such call sites. Table 4.2 shows the number of directly related classes for each task and the number of call sites in *PolySites*.

For each task we wrote a test suite that exercised the tested functionality, and we used the tool to execute the suite and to determine what receiver classes were exercised for each call site from *PolySites*. Our goal was to write a test suite that exercised *all possible receiver classes* for each call site. Substantial effort was put into writing the test suites. For each task, two people (working independently of each other) thoroughly examined the code and wrote tests that exercised all possible receiver objects. For each call site, the sets of exercised receiver classes obtained by the two people were carefully compared to ensure that there were no differences. As a result, for each task we had a test suite that exercised all possible receiver classes and target methods for each call site in *PolySites*.

Once we had test suites that exercised all possible classes/methods, we measured the coverage statistics reported by the tool for these suites. These statistics were based on the output of the fragment class analysis used by the tool: the analysis computed a set of possible classes/methods for each $c \in PolySites$, and the tool reported what percentage of these classes/methods was actually exercised by the test suite. In general, this reported coverage may be less than 100% because the analysis produces RC and TM requirements that are overestimates of the coverage that could be actually achieved—that is, the analysis may report infeasible receiver classes and infeasible target methods. Clearly, the goal of tool designers should be to use a class analysis that produces few infeasible classes/methods. As a precision metric we used the coverage that was reported by the tool for our test suites (which in reality exercise all possible classes/methods). When using a more precise analysis, the tool reports higher coverage; in the best case, the tool would report 100% coverage.

For our experiments we evaluated three fragment class analyses. All three analyses were designed using the general approach presented in Section 4.3: we first created the placeholders from Figure 4.5, and then we ran the solution engine of a whole-program class analysis. The first fragment class analysis (denoted by RTA_f) was derived from Rapid Type Analysis (RTA) [5]. RTA is a popular whole-program class analysis that performs class analysis and call graph construction in parallel. It maintains a worklist of methods reachable from `main`, and a set of classes instantiated in reachable methods. In the final solution, the set of classes for a variable `v` is the set of all instantiated subclasses of the declared type of `v`. RTA represents a point at the lower end of the cost/precision spectrum of class analysis.

The second fragment class analysis (denoted by AND_f) was derived from a whole-program points-to analysis for Java [51] which is based on Andersen's points-to analysis for C [4]. This whole-program analysis is a context-insensitive version of analyses $\mathcal{A}_p$ and $\mathcal{A}_c$ from Section 4.5,[3] and represents a point at the high end of the cost/precision spectrum for flow- and context-insensitive class analyses. Even though the analysis has cubic worst-case complexity, there are efficient implementation techniques that make it practical [51].

The third fragment class analysis (denoted by 0-CFA_f) is derived from a variation of the whole-program points-to analysis from [51]. In this variation, the whole-program analysis engine creates a single object name for all object allocation sites for a given class C—i.e.,

[3]More precisely, it is an instance of $\mathcal{A}_c$ for $k = 0$.

Task	Hierarchy		RTA_f		0-CFA_f		AND_f	
	C_{RC}	C_{TM}	C_{RC}	C_{TM}	C_{RC}	C_{TM}	C_{RC}	C_{TM}
task1	100%	100%	100%	100%	100%	100%	100%	100%
task2	67%	63%	67%	63%	76%	72%	76%	72%
task3	50%	100%	50%	100%	100%	100%	100%	100%
task4	31%	63%	45%	71%	100%	100%	100%	100%
task5	18%	21%	88%	92%	100%	100%	100%	100%
task6	76%	85%	76%	85%	97%	98%	98%	98%
task7	10%	15%	32%	48%	82%	93%	87%	93%
task8	5%	9%	18%	29%	62%	62%	62%	62%

Table 4.3: Reported coverage. More precise analyses result in higher reported coverage.

instead of having a separate object name o_i for each `new` expression as in [51], there is a single object name o_C for all expressions "`new C`". This version of the analysis is essentially equivalent to the 0-CFA class analysis [58, 18, 27],[4] and even though it is potentially less precise than the points-to analysis from [51], it still has cubic worst-case complexity and belongs at the high end of the cost/precision spectrum for flow- and context-insensitive class analyses.

Inside our coverage tool we used these three fragment class analyses to compute the *RC* and *TM* coverage requirements. We then ran our test suites (which in reality exercise all possible classes/methods), and we computed the achieved coverage with respect to *RC-RTA*$_f$, *TM-RTA*$_f$, etc. More precisely, for each analysis, we computed the sum S_1 of the number of possible receiver classes over all sites in *PolySites* as determined by the analysis, as well as the sum S_2 of the number of actually observed receiver classes at these sites. The tool reported the ratio $C_{RC} = S_2/S_1$ as a coverage metric for the *RC* criterion. A similar ratio C_{TM} was computed for the *TM* criterion. The results from these experiments are shown in Table 4.3. The column labeled "Hierarchy" represents the coverage with respect to the *RC* and *TM* criteria that were computed by just examining the class hierarchy. Class analyses that are more precise result in higher reported coverage percentages. In the best case, the analyses introduce no imprecision (i.e., they do not report infeasible receiver classes), and the reported coverage is 100%.

[4]The only difference is that our analysis is more precise with respect to inherited fields. For example, if class `B` inherits a field `f` declared in class `A`, our analysis distinguishes between `A.f` and `B.f`, while 0-CFA does not make this distinction.

Task	0-CFA$_f$ (sec)	AND$_f$ (sec)
task1	4.7	8.6
task2	12.8	25.1
task3	2.9	5.3
task4	5.3	6.4
task5	3.6	4.3
task6	12.2	35.8
task7	13.8	18.1
task8	15.4	20.4

Table 4.4: Analysis running times.

There are two important conclusions from these results. First, using the class hierarchy and RTA$_f$ to determine the coverage requirements often results in infeasible receiver classes and target methods. Thus, even for test suites that in reality achieve high coverage, the tool may report low coverage statistics. This situation is clearly unacceptable, and there is a need to use more precise analyses. Second, 0-CFA$_f$ and AND$_f$ perform very well, and in fact in half of the cases they achieve prefect precision. This indicates that these analyses are good candidates for inclusion in realistic coverage tools for testing of polymorphism.

As part of our experiments, we also measured the cost of computing the coverage requirements. All measurements were performed on a 360MHz Sun Ultra-60 machine with 512MB memory. The reported times are the median values out of three runs. Using the class hierarchy or RTA$_f$ has linear worst-case complexity, and in reality had negligible cost (less than 5 seconds for each testing task). The cost of using 0-CFA$_f$ and AND$_f$ is shown in Table 4.4; this is the cost of analyzing all methods that are directly or transitively reachable from the interface methods, both in directly related classes and in their server classes. Despite the cubic worst-case complexity, the two more precise analyses clearly have practical cost, due to the implementation techniques described in [51].

Chapter 5

Related Work

To the best of our knowledge, the techniques from Chapter 2 are the first attempt to define a general framework for interprocedural dataflow analysis of software fragments. In the context of particular analyses, previous work proposes a variety of techniques to address specific limitations of the traditional model of whole-program analysis. These techniques can be broadly classified in two categories: (i) approaches for describing the possible effects of parts of a program in order to analyze the rest of the program, and (ii) approaches for analyzing parts of a program without any information about the rest of the program. In this chapter we only describe previous work that belongs in these two categories—for example, we do not discuss whole-program pointer analyses and class analyses which employ techniques that are not relevant to fragment analysis.

5.1 Describing the Effects of Software Fragments

Various approaches in previous work attempt to create descriptions of the behavior of a software fragment and to use these descriptions when analyzing software that interacts with that fragment.

5.1.1 Summary Information Dependent on the Rest of the Program

One of the first uses of whole-program interprocedural analysis is in the $\mathbf{R}^n$ programming environment for Fortran [14]. The need for such analysis is motivated by the limitations of the intraprocedural analyses and optimizations that a compiler can perform when separately compiling an individual procedure. For example, the lack of information about the effects of calls to other procedures inhibits a number of intraprocedural optimizations.

To address this problem, the environment uses interprocedural whole-program analysis to compute and store summary information that is subsequently used for intraprocedural compiler optimizations. For example, a whole-program interprocedural side-effect analysis determines the set of variables that may be modified by each call statement in the program. This information is subsequently used during the compilation of individual procedures to improve intraprocedural analyses and optimizations. The similarity between this work and our framework for fragment analysis is that in both a program fragment is analyzed using some available information about the rest of the program. However, there are two conceptual differences. First, in $\mathbf{R}^n$ the analyses that use the summary information are intraprocedural (employed by the compiler when compiling a single procedure), while we consider a more general problem in which these analyses are interprocedural. Furthermore, we are interested in situations in which whole-program analysis is not possible and cannot be used to compute summary information for use by a subsequent fragment analysis. For example, Chapter 3 presents an approach for constructing library summary information without having access to the rest of the program.

There are various interprocedural whole-program analyses that construct summary information about a procedure and use this information when analyzing the callers of that procedure. One common category of approaches is based on the idea of constructing the summary information by analyzing not only the procedure under consideration, but also all procedures that are directly or transitively called by that procedure. For example, several existing whole-program analyses [10, 13, 8, 67, 12, 52, 37] perform a bottom-up traversal of the program call graph and compute a summary function for each visited procedure. This summary function is then used when analyzing the callers of that procedure and when constructing their summary functions. Summary functions can also be created in top-down manner, by introducing all possible contexts at the entry of the analyzed procedure and by computing the effects of the procedure (including the effects of its callees) for each context [31].

One serious limitation of these approaches is the implicit assumption that a called procedure can be analyzed either before its callers are analyzed, or while its callers are being analyzed. For example, if there is no available source code or summary information for a called procedure, the analysis of its callers either cannot be performed, or has to be performed with conservative assumptions about the possible effects of the procedure. Furthermore, callee procedures may not even exist at the time when their callers are analyzed. A classical example of this situation is the use of the callback mechanism (e.g., calls through function pointers in C, or calls to unknown overridden methods in Java). When designing the approach from Chapter 3 for creating summary information for precompiled C libraries, one of our goals was to avoid the above limitation.

5.1.2 Summary Information Independent of the Rest of the Program

Several analysis approaches compute summary information for a software fragment independently of the callers and callees of that fragment. One particular approach is to compute partial analysis results for each fragment, to combine the results for all fragments in the program, and then to perform the rest of the analysis work. For example, the points-to analysis due to Das [17] processes each compilation unit separately, produces a partial points-to graph for that unit, combines all such graphs, and performs additional work to obtain a points-to graph for the entire program.

The above approach improves the scalability of the analysis and reduces the amount of necessary work after a program change. However, the computed summary information is specific to the particular analysis, and even the particular analysis implementation. Heintze [32] takes a different approach and creates summary information that only encodes the structure of the source code of the fragment, without performing any analysis work in advance. The advantage of this approach is that the resulting summary information can be used for many different analyses, and can be reused in the context of different compilers and tools. Our approach for creating summary information for precompiled C libraries (presented in Chapter 3) is based on the same idea: we create summaries that represent the structure of the source code of the library, and that can be used to perform a variety of points-to and side-effect analyses of library clients. In addition, we propose summary optimizations that filter out parts of the summary that are irrelevant with respect to the clients of the library. Thus, unlike [17, 32], we do not maintain a complete description of the library, and as a result we reduce the cost of the analysis that uses the summary information.

Flanagan and Felleisen [24] present a componential set-based analysis for Scheme. This analysis creates a system of constraints for each program component. Similarly to the summaries from Chapter 3, the system of constraints encodes the potential effects of the component on the rest of the program, without describing all interactions within the component. The systems for all program components are combined and information is propagated among them in order to create a global solution. This solutions is then propagated to the individual components. Conceptually, the component-level constraint systems in [24] are similar to the optimized summaries from Chapter 3, and the global phase of the analysis is similar to our summary-based fragment analysis of the client module. However, our approach targets a different language and analysis, and uses summaries and summary optimizations different from the ones in [24].

5.1.3 User-defined Summary Information

While the approaches described above construct the summary information automatically, there have been proposals for employing summary information provided by the analysis user (e.g., programmer, tester, etc.). Guyer and Lin [29] propose annotations for describing libraries in the domain of high-performance computing. The annotations encode high-level semantic information (e.g., points-to and side-effect properties) and are produced by a library expert. Similarly, Rugina and Rinard [53] propose the use of design information in the context of optimizing compilers. In particular, they present summary information that describes how a called procedure affects points-to relationships and how it accesses regions of arrays; this information is used to perform automatic program parallelization. In essence, the approaches from [29, 53] use summary functions provided by the analysis user. Dwyer [20] presents a modular dataflow analysis for verifying correctness properties of concurrent programs. Information about the surrounding environment of a module is represented using an environment automaton. This automaton describes the possible interactions between the module and the environment, and is provided by the user (based on design artifacts or existing code).

Approaches based on user-defined summary information are capable of achieving precision that is very hard (or even impossible) to achieve automatically. Their key advantage is the ability to employ high-level user knowledge. On the other hand, constructing user-defined summary information may be hard, time-consuming, and error-prone.

5.2 Using Conservative Assumptions about External Code

The second broad category of related work consists of approaches for analyzing a software fragment when there is no available information about the surrounding environment. In essence, these approaches perform fragment analysis using conservative assumptions about the possible effects of unknown external code. In previous work, specific techniques have been proposed in the context of particular kinds of analyses. To the best of our knowledge, the approach presented in Chapter 2 is the first one that provides a general framework for addressing this problem.

Harrold and Rothermel [31] describe an approach for applying Landi-Ryder's whole-program pointer analysis for C [35] to a software module. All possible contexts are introduced at the entry of the module, and then information is propagated in top-down manner inside the module. The approach assumes that there is available summary information for all other modules that are being called by the analyzed module. Our method for performing fragment points-to analysis of a library module (described in Chapter 3) addresses a problem similar to the one from [31]. While the analysis from [31] is based on Landi-Ryder's flow- and context-sensitive analysis, our approach can be applied to the entire category of flow- and context-insensitive points-to analyses (e.g., [4, 60, 69, 56, 17, 32]). Unlike [31], we present an approach that can be used when there is no available information about called modules. We also provide empirical results that confirm the practicality of our approach.

Chatterjee et al. [10] present a flow- and context-sensitive whole-program points-to analysis for object-oriented languages, which subsequently has been modified to perform analysis of library modules [9]. The library analysis processes the library and all of its callees in bottom-up manner, and then performs a top-down traversal that essentially propagates conservative assumptions about the clients of the library. This approach addresses the same problem as the fragment class analyses described in Chapter 4. Our approach for constructing fragment class analyses is more general and can be applied to a large number of existing whole-program analyses [2, 5, 28, 18, 48, 61, 62, 65, 38, 51, 27, 42]. Furthermore, we present empirical results that evaluate the absolute precision of our fragment analyses and confirm their effectiveness.

Previous work presents several analyses for fragments of Java programs, performed without having any information about the callees or callers of the analyzed fragment. Sreedhar et al. [59] define extant analysis that determines whether a reference variables may point to instances of unknown classes. This information is used to perform compiler optimizations in the presence of dynamic class loading. Sweeney and Tip [63] describe analyses and optimizations for the removal of unused functionality in Java modules. Conservative assumptions are used to approximate the effects of code located outside of the optimized modules. We believe that the fragment analyses from Chapter 4 can be easily generalized to handle unknown callees, which would make it possible to apply them to the problems addressed in [59] and [63]. Ghemawat et al. [26] present several analyses of the properties of fields in Java modules. Analysis results are used to perform various compiler optimizations such as removal of redundant run-time tests and compile-time resolution of virtual calls. A variety of whole-program escape analyses [67, 13, 8] can be modified to analyze a Java software fragment without having any information about the callers or callees of the fragment. For example, Vivien and Rinard [66] present an incrementalized escape analysis that is based on the whole-program analysis from [67]. The analysis dynamically grows the analyzed program region, and makes conservative assumptions about the rest of the program.

Harrold and Rothermel [30] present a method for performing def-use analysis of a given class for the purposes of dataflow-based unit testing in object-oriented languages. Their approach is based on a whole-program def-use analysis for C by Pande et al. [45].

The method in [30] constructs a placeholder driver that represents all possible sequences of method invocations initiated by client code; however, the driver does not take into account the effects of aliasing, polymorphism, and dynamic binding. The placeholder `main` method presented in Chapter 4 is essentially a placeholder driver that models these features, and therefore can be used to perform dataflow-based testing of individual classes and collections of classes.

Chapter 6

Summary and Future Work

The traditional model of whole-program dataflow analysis has several limitations that make it unsuitable for many real-world software systems. Whole-program analysis cannot be applied to incomplete programs, to programs containing unanalyzable modules, and to very large programs. The impact of dataflow analysis research can be broadened significantly if the limitations of whole-program analysis are addressed and resolved. To achieve this goal, in this thesis we propose the paradigm of fragment dataflow analysis.

6.1 Theoretical Foundations

Fragment dataflow analysis is an interprocedural dataflow analysis that analyzes software fragments rather than complete programs. The input to the analysis is a software fragment and some knowledge about the environment in which the fragment will operate. The goal of the analysis is to determines properties of the possible run-time behaviors of the fragment in that environment.

We have developed a theoretical framework for constructing fragment dataflow analyses. This framework allows analysis designers to construct fragment analyses and to reason about their properties. The key idea of our approach is to derive fragment analyses from existing whole-program analyses, in order to reuse existing algorithmic techniques and implementations of whole-program analyses. The essence of the approach is to construct fragment analyses that "simulate" the behavior of the underlying whole-program analyses with respect to all whole programs that could contain the analyzed fragment. Our framework provides a sound theoretical basis for adapting the large body of existing work on whole-program analysis to solve many problems that currently cannot be solved with whole-program analysis.

6.2 Points-to Analysis and Side-effect Analysis for C Programs Built with Precompiled Libraries

We present an approach for performing points-to analysis and side-effect analysis for C programs that are built with precompiled library modules. Points-to and side-effect analyses compute information that is of fundamental importance for many other analyses and optimizations. Our work targets flow- and context-insensitive points-to analyses, as well as side-effect analyses based on such points-to analyses. We first propose fragment analyses of a library module; these analyses can be used without any available information about the callers and callees of the library. The key idea of our approach is to create placeholders that simulate the possible effects of unknown external code. We also show how to perform fragment analyses of a client module without having the source code of the used library modules, based on a technique for constructing summary information that precisely describes the possible effects of library modules. The summary information is optimized to exclude details that are irrelevant with respect to client modules. Our empirical results confirm the practicality and effectiveness of this approach. As a result of this work, many existing (and future) techniques for points-to and side-effect analysis can be adapted to programs that are built with precompiled components.

6.3 Class Analysis for Testing of Polymorphism in Java Software

We have defined a general method for performing class analysis for the purposes of testing of polymorphism in Java. Class analysis is a fundamental dataflow analysis for object-oriented software that has a wide variety of uses in optimizing compilers and software engineering tools. We consider the use of fragment class analysis in a coverage tool for testing of polymorphism, in order to determine the test coverage requirements. Our method allows many existing and future flow-insensitive whole-program class analyses to be adapted for use in such coverage tools. Our empirical results clearly show that appropriately chosen analyses have high precision and low cost, and therefore are good candidates for inclusion in real-world testing tools. This work is the first one to show how to construct high-quality coverage tools for testing of polymorphism in Java software.

6.4 Future Work

One possible direction for future work is to generalize the techniques from Chapter 3 for flow- or context-sensitive points-to analyses. In particular, it is interesting to evaluate empirically how different levels of detail in the library summary affect the precision of analyses that use the summary. A related direction of future work is approaches for summarizing the effects of object-oriented libraries for the purposes of class analysis of library clients. This problem is of particular importance because object-oriented software systems are often built on top of already existing large and complex infrastructure (e.g., standard libraries, virtual machines, middleware), and re-analysis of this infrastructure for each analyzed program is infeasible.

Another interesting problem is to generalize the fragment class analyses from Chapter 4 for situations when some of the callees of the fragment are not available. It is particularly interesting to consider how such analyses can be useful for compiler optimizations (e.g., for optimization of reusable libraries) and for software engineering tools (e.g., for program understanding). Any such analysis has to be evaluated empirically with respect to its intended application. For example, for software engineering applications where high imprecision is unacceptable, experiments would have to determine absolute precision by comparing analysis results with the actual set of possible run-time values (similarly to the experiments from Chapter 4).

As a more general direction of future work, it is important to investigate different categories of summary information for software components, and the use of this information for analyzing component interactions in software systems that are built with these components. A variety of interesting questions remain open: what are important kinds of summary information, how to compute it precisely and efficiently, and how to make it reusable for different clients and different analyses. The importance of this research area is significant because of the widespread use of component-based software systems. In this context, it is crucial to be able to analyze systems that are built with components, in order to understand, modify, optimize, test, and verify such systems. We believe that the approaches presented in this thesis are a step toward solving this problem.

References

[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis Symposium*, LNCS 864, pages 78–100, 1994.

[2] O. Agesen. The cartesian product algorithm. In *European Conference on Object-oriented Programming*, LNCS 952, pages 2–26, 1995.

[3] R. Alexander and J. Offutt. Criteria for testing polymorphic relationships. In *International Symposium on Software Reliability Engineering*, pages 15–23, 2000.

[4] L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[5] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, 1996.

[6] R. Binder. Testing object-oriented software: a survey. *Journal of Software Testing, Verification and Reliability*, 6:125–252, December 1996.

[7] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[8] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999.

[9] R. Chatterjee and B. G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-433, Rutgers University, April 2001.

[10] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[11] M. H. Chen and M. H. Kao. Testing object-oriented programs—an integrated approach. In *International Symposium on Software Reliability Engineering*, pages 73–83, 1999.

[12] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[13] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, 1999.

[14] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8(5):491–523, October 1986.

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[16] B. Cox. The need for specification and testing languages. *Journal of Object-Oriented Programming*, 1(2):44–47, June 1988.

[17] M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[18] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[19] A. Diwan, J.Eliot B. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 292–305, 1996.

[20] M. Dwyer. Modular flow analysis of concurrent software. In *International Conference on Automated Software Engineering*, pages 264–273, 1997.

[21] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Conference on Programming Language Design and Implementation*, pages 242–257, 1994.

[22] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[23] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[24] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999.

[25] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, LNCS 1824, pages 175–198, 2000.

[26] S. Ghemawat, K. Randall, and D. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Conference on Programming Language Design and Implementation*, pages 334–344, 2000.

[27] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, November 2001.

[28] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, 1997.

[29] S. Guyer and C. Lin. Optimizing the use of high performance software libraries. In *Workshop on Languages and Compilers for Parallel Computing*, LNCS 2017, pages 227–243, 2000.

[30] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. *Symposium on the Foundations of Software Engineering*, pages 154–163, 1994.

[31] M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):442–460, July 1996.

[32] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA. In *Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[33] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, May 1999.

[34] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[35] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[36] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 199–215, 1999.

[37] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symposium*, LNCS 2126, pages 279–298, 2001.

[38] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, 2001.

[39] T. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[40] T. McCabe, L. Dreyer, A. Dunn, and A. Watson. Testing an object-oriented application. *Journal of the Quality Assurance Institute*, 8(4):21–27, October 1994.

[41] R. McDaniel and J. McGregor. Testing the polymorphic interactions between classes. Technical Report 94-103, Clemson University, March 1994.

[42] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, 2002.

[43] J. Overbeck. *Integration Testing for Object-Oriented Software*. PhD thesis, Vienna University of Technology, 1994.

[44] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 146–161, 1991.

[45] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations in C programs. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[46] D. Perry and G. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, January 1990.

[47] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–340, 1994.

[48] C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, December 1999.

[49] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[50] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

[51] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 43–55, 2001.

[52] E. Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[53] R. Rugina and M. Rinard. Design-driven compilation. In *International Conference on Compiler Construction*, LNCS 2027, pages 150–164, 2001.

[54] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, March 2001.

[55] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, LNCS 1302, pages 16–34, 1997.

[56] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[57] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[58] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[59] V. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Conference on Programming Language Design and Implementation*, pages 196–207, 2000.

[60] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[61] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, U. Passau, September 2000.

[62] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 264–280, 2000.

[63] P. Sweeney and F. Tip. Extracting library-based object-oriented applications. In *Symposium on the Foundations of Software Engineering*, pages 98–107, 2000.

[64] N. N. Thuy. Testability and unit tests in large object-oriented software. In *Proc. 5th International Software Quality Week.* Software Research Institute, 1992.

[65] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–293, 2000.

[66] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2001.

[67] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999.

[68] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[69] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Symposium on the Foundations of Software Engineering*, pages 81–92, 1996.

[70] H. Zhu, P. Hall, and J. May. Software unit testing coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Vita

Atanas Rountev

1995	B.S. in Computer Science and Engineering, Technical University, Sofia, Bulgaria.
1996–2002	Research Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.
1999	M.S. in Computer Science, Rutgers, The State University of New Jersey.
1999	Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1999.
2000	Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.
2001	Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of the 10th International Conference on Compiler Construction*, April 2001.
2001	Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001.
2002	Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2002.
2002	Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Constructing precise object relation diagrams. In *Proceedings of the IEEE International Conference on Software Maintenance*, October 2002.
2002	Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graph construction in the presence of function pointers. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, October 2002.
2002	Ph.D. in Computer Science, Rutgers, The State University of New Jersey.