Marcus Edvinsson

Towards a Framework for Static Analysis Based on Points-to Information

Licentiate Thesis
Marcus Edvinsson

Towards a Framework for Static Analysis Based on Points-to Information

Licentiate Thesis

Computer Science

2007

Växjö University
A thesis for the Degree of Licentiate of Philosophy in Computer Science.

Towards a Framework for Static Analysis Based on Points-to Information
Marcus Edvinsson

Växjö University
School of Mathematics and System Engineering
SE-351 95 Växjö, Sweden
http://www.vxu.se/msi/

Reports from MSI, no 07126/2007
ISSN 1650-2647
ISRN VXU/MSI/DV/R/-07126–SE
To Anna-Greta and Bengt-Eric
for always supporting and believing in me
Abstract

Static analysis on source code or binary code retrieves information about a software program. In object-oriented languages, static points-to analysis retrieves information about objects and how they refer to each other. The result of the points-to analysis is traditionally used to perform optimizations in compilers, such as static resolution of polymorphic calls, and dead-code elimination. More advanced optimizations have been suggested specifically for Java, such as synchronization removal and stack-allocation of objects. Recently, software engineering tools using points-to analysis have appeared aiming to help the developer to understand and to debug software. Altogether, there is a great variety of tools that use or could use points-to analysis, both from academia and from industry.

We aim to construct a framework that supports the development of new and the improvement of existing clients to points-to analysis result. We present two client analyses and investigate the similarities and differences they have. The client analyses are the escape analysis and the side-effects analysis. The similarities refer to data structures and basic algorithms that both depend on. The differences are found in the way the two analyses use the data structures and the basic algorithms. In order to reuse these in a framework, a specification language is needed to reflect the differences. The client analyses are implemented, with shared data-structures and basic algorithms, but do not use a separate specification language.

The framework is evaluated against three goal criteria, development speed, analysis precision, and analysis speed. The development speed is ranked as most important, and the two latter are considered equally important. Thereafter we present related work and discuss it with respect to the goal criteria. The evaluation of the framework is done in two separate experiments. The first experiment evaluates development speed and shows that the framework enables higher development speed compared to not using the framework. The second experiment evaluates the precision and the speed of the analyses and it shows that the different precisions in the points-to analysis are reflected in the precisions of the client analyses. It also shows that there is a trade-off between analysis precision and analysis speed to consider when choosing analysis precision.

Finally, we discuss four alternative ways to continue the research towards a doctoral thesis.

Key-words: Static analysis, Points-to analysis, Framework
Sammanfattning

Statisk analys av källkod eller binär kod hämtar information om ett mjukvaruprogram. I objektorienterade språk hämtar statisk points-to-analys information om objekt och hur de refererar till varandra. Resultatet av points-to analys används traditionellt för optimeringar i kompilatorer, såsom statisk reduktion av metodanrop med flera möjliga mål, samt eliminering av död kod. Mer avancerade optimeringar har föreslagits, speciellt för Java, såsom borttagning av synkroniseringspunkter samt allokering av objekt på stacken. På senare tid har det dykt upp mjukvaruutvecklingsverktyg som använder points-to-analys, vilka har som syfte att hjälpa utvecklaren att förstå och debugga mjukvara. Sannantaget så finns det en stor mängd verktyg som används traditionellt, och som skulle kunna använda points-to information, både i akademiska och industriella miljöer.

Vi ämnar konstruera ett ramverk som ska stödja utvecklingen av nya och förbättringen av existerande klienter till points-to analysens resultat. Vi presenterar två klientanalyser och undersöker de likheter och skillnader som finns mellan dem. Klientanalyserna är en escape-analys och en sidoeffektsanalys. Likheterna är bland annat gemensamma datastrukturer och grundläggande algoritmer som båda är beroende av. Skillnaderna finns i hur de båda analyserna använder datastrukturerna och algoritmerna. För att återanvända dessa i ramverket behövs det ett specifikationsspråk som kan representera dessa skillnader. Klientanalyserna har blivit implementerade, med de gemensamma datastrukturerna och de grundläggande algoritmerna, men de använder inte något gemensamt specifikationsspråk.


Slutligen diskuterar vi fyra alternativa sätt att fortsätta forskningen fram till en doktorsavhandling.

Nyckelord: Statisk analys, points-to-analys, ramverk
Acknowledgments

I would like to thank my adviser, Professor Welf Löwe, for his invaluable support and knowledge and for convincing me to pursue my PhD studies. I would not be where I am and this thesis would not exist without you.

An appreciation goes to my colleagues for supplying a supporting and friendly environment to work in, and for the valuable discussions about computer science, research, work and all non work related things.

Thank you Professor Joakim Nivre for proofreading the thesis and for giving me feedback, both regarding the academic language, English and research in general.

Last, but definitely not least, I want to thank my family, my girlfriend and my friends for all your love and support. You have given me many hours of leisure and many unforgettable moments away from computer science that helped me recharge and proceed with my studies in general and this thesis in particular.

Thank you all!
Contents

Abstract v
Sammanfattning vi
Acknowledgments vii

I Introduction 1

1 Introduction 2
  1.1 Research Questions 3
  1.2 Method 3
  1.3 Goal Criteria 4
  1.4 Structure of the Thesis 5

II State of the Art 7

2 Foundations 8
  2.1 General Definitions 8
  2.2 Program Representations 9
  2.3 Static Analysis 12
  2.4 Summary 20

3 Related Work 21
  3.1 Relating the Related Work 21
  3.2 Static Analysis Frameworks 24
  3.3 Client Analyses 26
  3.4 Client Analyses Summary 35
  3.5 Conclusions 37

III Initial Framework 39

4 Towards a Framework 40
  4.1 Points-to Analysis 40

viii
Part I

Introduction
Chapter 1

Introduction

Software should be developed efficiently and with a minimum of errors. Existing software is often in need of maintenance, i.e., correcting existing errors and adding new features, etc. For the developers it is necessary to understand how the software behaves and how it is constructed. This is a very complex task, even for small systems, even if the developer has been involved in the same development project for a long time. This is even harder for developers that are introduced to and supposed to work with new software, software they have never worked with before. One way of helping the developers understand software is to let them use tools specialized in analyzing different properties of the software, such as its structure and behavior. Such tools let the developer collect information about, for instance, the structure of the software and clues to what problems it currently may have.

In static analysis, the source code is analyzed using appropriate assumptions and restrictions enabling us to draw conclusions about how the analyzed program may behave. Dynamic analysis of programs, i.e., analyzing program execution, is often not appropriate since it only gives information about specific executions. In many situations, we want to be conservative in our statements about the analyzed program, i.e., the analysis result should include all results that may occur. It may not even be possible or feasible to run the analyzed program in order to analyze it dynamically. Static analysis is more appropriate in these cases, since it overcomes these shortcomings.

Points-to analysis is a static analysis that finds reference information in a program. In object-oriented languages, such as Java, points-to analysis answers the question: Where in the program may a certain abstract object be referenced? Direct usage of points-to information includes static resolution of polymorphic calls and dead-code elimination. The result of the points-to analysis also provides information that other analyses may use. A client analysis is an analysis using the points-to analysis results to calculate its results. Creating new static analyses based on the points-to result can be very useful, in both compiler optimization and in software comprehension settings. A new abstraction of the program can be shown by simply adjusting a client analysis or creating a completely new analysis. It would be highly useful to have a framework that supports the process of creating new client analyses.
1.1. Research Questions

Such a framework would separate the points-to analysis from the representation of the points-to analysis results. It would also provide a number of basic analyses that are useful when building new analyses based on points-to results.

The quality of an analysis can be measured in, for instance, accuracy and speed. These two entities are competing with each other; to have higher accuracy it is necessary to spend more time; and to reduce time it is often inevitable to sacrifice some accuracy. Different analyses in different settings may prefer different trade-offs between these two qualities. A framework that supports the development of client analyses and allows the accuracy of the points-to analysis to be selected would make the development of client analyses faster and more reliable. It would also make it possible to find suitable trade-offs between precision and speed in an easier and faster way.

We construct such a framework in this thesis and we call it the Client Analysis Framework.

1.1 Research Questions

Based on the previous discussion, this thesis answers the following research question:

1. What is needed to produce client analyses based on points-to results?
   (a) What are the commonalities and the differences of the analyses?
   (b) How could the analyses be specified?
   (c) How could the analyses be generated from such a specification?

1.2 Method

We use a constructive approach to answer the research question, i.e., we construct a framework that supports the development of static analyses based on points-to results. One answer to the research question is given by the way the framework is constructed. This answer is only one of many possible, and it is only a partial answer; a complete answer would find out what is sufficient and what is necessary. We only find what is sufficient with our limitations.

More specifically, we answer the research questions by creating two client analyses that make use of points-to results. We find a number of common basic analyses, besides points-to analysis, that are useful for these client analyses, and it can be argued that they are useful for other types of client analyses, as well. We also identify the differences between these two analyses and we discuss how the analyses could be specified and how analyses could be
Chapter 1. Introduction

generated from such a specification. The goal criteria, which are discussed in Section 1.3, are evaluated in two experiments, one evaluating the development speed and one evaluating the analysis precision and analysis speed.

1.3 Goal Criteria

The criteria we want to evaluate our work against are development speed, analysis precision and analysis speed. We use the term precision to express the accuracy of the analysis. The properties of the analysis, presented in Section 3.2, ensures that the recall is 100% at all times. This is why we only consider precision.

The criteria will also be used to relate our work to the work of others. The development speed is important since this is a measure of the direct benefit one can draw from a framework, as opposed to programming from scratch. We chose to prioritize the development speed over the analysis speed/precision. We do not prefer any of the latter, since it is necessary to find the trade-off between these two qualities that works best for a specific client. Sometimes it is more important to be fast and sometimes it is the precision that counts the highest. Therefore, we decided to have these two as two criteria on the same level of importance.

Development Speed

Developing new analyses and variants of existing analyses can be made easier if other analyses can be reused. Reuse will have two effects, stability of the analysis and decrease of the time spent on development. The development speed is measured as the effort needed for a developer to create a new client analysis. A more efficient support and reuse of algorithms and data structures enables a higher development speed. A framework supplies such support, and it requires the developer to give certain specifications of the developed client analysis. The effort spent by the developer to do so is considered development speed. However, this is not possible to measure without performing controlled experiments involving developers. To relate approaches to this criterion we instead look at how much reusable code an approach supplies, and this is an estimate of the support the developer gets and a measure of the effort the developer needs to invest to create a client analysis.

Analysis Speed and Analysis Precision

The time an analysis takes to produce its result is important for the usability of the analysis. A slow analysis is neither suitable for tasks involving human interaction, nor for use in compiler optimizations. However, it could still be
1.4. Structure of the Thesis

The precision of the analysis result is a measure of the quality of the analysis. A result will enable client analyses to perform better, i.e., produce a preciser result, as well. In some applications it is of great importance to have high precision. The need for precision could vary depending on the size of the analyzed code or other circumstances.

Traditionally, \textit{precision} is a measure of how many correct answers are included in the result set in relation to the total number of answers in the result set and \textit{recall} is a measure of how many correct answers are included in the result set in relation to the total number of correct answers. Measuring the analysis precision and speed can be done in controlled experiments by measuring the analysis result size for analysis precision and the time spent performing the analysis for the analysis speed. The analysis results size can be used as a metric of the analysis precision, since the recall of the analysis result is guaranteed to be 100\%, which is given from the analysis properties discussed in Section 3.2.

A more precise analysis produce a more precise result, i.e., a smaller number of objects. The reduced number of objects may increase the analysis speed in some aspects, even though the analysis is more time-consuming in general. Therefore, it may not be obvious that a more precise analysis is more time-consuming. A more detailed discussion about this can be found in Section 3.3 when we discuss how to relate the criteria.

1.4 Structure of the Thesis

The thesis is divided into four parts; Introduction (Chapter 1), State of the Art (Chapters 2 and 3), Initial Framework (Chapters 4 and 5), and The Present and the Future (Chapters 6 and 7).

Chapter 2 presents the foundations of this thesis. The methods, techniques and theories that this thesis is based on are presented.

Chapter 3 presents the work of others that relate to this thesis. It also evaluates the related work using our goal criteria.

Chapter 4 presents the commonalities and variation points we have identified in the client analyses. This knowledge is used to sketch a framework. This includes what the framework provides to the user and what the user should provide in form of specifications and programming code.

Chapter 5 presents the experiments that support our work. We argue how the framework presented in Chapter 4 will help the client analysis developer and support this work to get a fast client analysis development.
process. We also show that the framework allows the tradeoff between speed and precision, to be varied.

Chapter 6 presents the conclusions and main contributions of this thesis. The chapter also works as a summary of the content presented before.

Chapter 7 discusses four alternative ways towards a doctoral thesis. These ways are presented in detail and the main steps that need to be taken are discussed, as well.
Part II

State of the Art
Chapter 2

Foundations

This section presents notions and definitions that are used later in the thesis, either to present the related work or our framework. Some of the notions and definitions are presented in two different ways, one in general terms and one with details about how they are used in the framework.

The definitions and notions are mutually dependent on each other; unfortunately, it seems impossible to present them without forward-references. This problem is caused by five relations: The fact that the analysis uses data structures that represent the analyzed program, that the data structures are tailored for the specific use in the analysis, and that the analysis produces analysis results, which are also referred to by the data structures.

The chapter contains four sections. The first section presents some general definitions. The second section discusses four program representations that may be used in static analysis. The third section presents important concepts in static analysis and gives short motivations for their importance. The fourth section summarizes the concepts that are presented in this chapter, and works as a short chapter overview.

2.1 General Definitions

Points-to Analysis

Points-to analysis is a static program analysis that extracts reference information from a given input program. Points-to analysis targets object-oriented languages, while its predecessor analyses, e.g., alias analysis, are concerned with other programming paradigms, such as functional and imperative programming languages.

In an object-oriented language, objects are targets for, for example, method calls and field references. At run-time, a method call has a number of parameters that each holds exactly one object, the target object and the method arguments. However, for static methods the target object is statically known and this needs not be calculated by the analysis. When a field is referenced at run-time, an object is targeted and an object is referenced by the field. For each method call and field reference, points-to analysis finds a
set of abstract objects that may be targets or passed as arguments.

**Framework**

An object-oriented framework, or framework for short, is a coarse-grained component that, through parameterization, can be instantiated to solve specific problems within a problem domain [Akm03]. The framework provides functionality that is common to many of the problems in the problem domain. The parameterization tailors the framework for solving a certain subset of problems from the problem domain. The points that need to be specified when instantiating a framework are called variation points, or sometimes hotspots [Pre97].

There are two main approaches when it comes to parameterizing these hotspots, white-box and black-box instantiation [Pre97, Akm03]. The white-box instantiation requires that classes are extended or interfaces implemented in order to supply the functionality the framework needs to handle the specifics regarding the specific problems. In black-box instantiation, no new code needs to be created. It is rather a question of selecting and correctly composing a number of existing software components in such a way that they fill the hotspots.

### 2.2 Program Representations

There are a number of data structures that represent the structure of a program and that could be used in static analysis. Here, we present four of them, namely, Abstract Syntax Tree (AST), Basic Block Graph, Static Single Assignment (SSA) graph, and Sparse SSA graph, and we motivate why we use a type of Sparse SSA graph for program representation for our analyses.

**Abstract Syntax Trees**

The abstract syntax tree (AST) is a reduced form of the parse tree useful to represent program language constructs [ASU86]. It is a labeled, directed, ordered tree that represents a program by having nodes representing operators and children nodes representing operands [ALSU07], either being other operators or variables/constants. While the parse tree has nonterminals as internal nodes, the abstract syntax tree has programming constructs (operators). Many programming constructs in an AST have corresponding nonterminals in the parse tree, but for some nonterminals this mapping cannot be made. They may exist only to ensure, for instance, correct precedence. An example of an AST is given in Figure 2.1.

The AST does not represent the data dependencies in the program, which is an important aspect in points-to analysis. Therefore, we decided not to
Chapter 2. Foundations

Grammar

Assign ::= Variable '=' Expr
Variable ::= literal /* name */
Expr ::= Number Op Number
Number ::= integer | float
Op ::= '+' | '-' | '*' | '/'

Example statement

\[
a = 2 + 7
\]

Figure 2.1: Abstract syntax tree example.

use the AST data structure as the program representation in our points-to analysis.

**Basic Block Graphs**

A *basic block* is a sequence of statements in a programming language, such that the control flow will always start with the first statement, continue through the sequence, and end with the last statement [ALSU07]. There are no jumps to or from the block other than to the first statement and from the last statement in the sequence. A *basic block graph* is a graph where nodes are basic blocks and edges represent the flow between the basic blocks. An example of a basic block graph representation of a program is shown in Figure 2.2.

The basic block graph does not represent the data dependencies in the program, which is an important aspect in points-to analysis. Therefore, we decided not to use the basic block graph data structure as the program representation in our points-to analysis.

**Source code**

\[
\begin{align*}
\text{int } & \text{ i=} \ 0; \\
\text{int } & \text{ c=} \ 0; \\
\text{while } & \text{ (i<10) } \{ \\
& \text{ c=} \ c+\ i; \\
& \text{ i++; } \\
\} \\
\text{print("Result= " } & \text{ + c);}
\end{align*}
\]

**Basic Block graph**

\[
\begin{align*}
\text{int } & \text{ i=} \ 0; \\
\text{int } & \text{ c=} \ 0; \\
\text{while } & \text{ (i<10) } \{ \\
& \text{ c=} \ c+\ i; \\
& \text{ i++; } \\
\} \\
\text{print("Result= " } & \text{ + c);}
\end{align*}
\]

Figure 2.2: Basic block graph example.
SSA Graphs

In a Static Single Assignment (SSA) graph, the nodes are instructions and the edges are data flow (over local variables) connecting the instructions. In this way, the def-use relation is modelled explicitly, where a variable is statically defined and used only once, i.e., only one instruction defines a value/variable. Nodes may have a fixed number of incoming edges, one for each argument of the operation, e.g., a method call. The nodes have ports, one for each argument of the operation and one for each result. The value of a variable may be affected by different branches in the program. Since the def-use relation is modelled explicitly these values need to be merged when the branches merge. This is done by introducing a $\phi$-node, which has an arbitrary number of in-ports – one for each branch to merge – and it merges these values onto one out-port, which may have several uses and, hence, outgoing edges. This way, def-use relations are preserved.

SSA graphs, as presented in [CFR+91, Muc97], are primarily used as intermediate program representation in the analysis phase of compilers. Variants have been used for detecting program equivalence and inherent parallelism in imperative programs, according to [CFR+91].

![SSA graph example](image)

The example in Figure 2.3 shows seven lines of source code and its SSA graph representation. The two variables $c$ and $i$ are initialized to 0 and the same constant value is reused in the graph. The `while` loop in the source code causes two loops in the SSA graph. Both statements within the while loop may get its in-values from either the two lines preceding the loop or from the statements within the loop. The two $\phi$-nodes make sure that the values from the statements preceding the loop and the values from within the loop are merged, and that the def-use relations are preserved.
Sparse SSA Graphs

A sparse SSA graph is an SSA graph. It only contains the information (nodes and edge-types) that is necessary to perform a certain analysis/transformation task. Nodes that do not contribute with useful information are removed and so are edges that become unconnected. This reduces both the space and time complexity associated with the SSA graph structure, both regarding construction and access. There is less information to store and the analysis algorithms need not consider as many nodes as in a complete SSA graph, although the graph still contains enough information to complete the intended task.

The SSA graph used by the points-to algorithm in the client analysis framework is a sparse SSA graph called Points-to SSA [LL07]. There is one graph per method which can be seen as a semantic abstraction of that method. All operations not relevant to reference computations are removed, i.e., operations and edges related to primitive types. The Points-to SSA is used since it models the necessary features of the analyzed program, i.e., the necessary elements related to reference calculations, in a memory efficient way. At the same time, it is possible to construct efficient and precise algorithms using the data structure.

2.3 Static Analysis

Let us consider an analysis of a program where each method is represented by an SSA graph. The graph has statements as nodes and the edges represent the control flow in the program. There are a number of approaches to perform static analysis, such as data flow analysis and constraint-based analysis [NNH99].

The data flow analysis approach lets data flow through a program graph by propagating analysis values between the statements in the program. Each statement type has a certain effect on the analysis values, which is calculated for each statement in a data-driven way. The calculation stops when a fix-point is reached, i.e., when no new information is produced from calculating the effect of any of the statements in the analyzed program. A more exhaustive presentation of the data flow analysis approach is given in presentation of the monotone data flow framework that follows.

The constraint-based analysis approach constructs a set of constraints from the analyzed program and then calculates the smallest solution to this constraint system. The constraints are generated for each program from rules based on the syntactic and semantic structure of the programming language. The generated constraint system is solved by iteratively transforming these constraints applying rewrite rules until no new result is given by applying any of the rules.
2.3. Static Analysis

Even though these techniques are quite different, they have some things in common. In this section, we present the most important characteristics that are typical for static analysis in general, and specifically for points-to analysis.

Syntactic Creation Point

A syntactic creation point are program points that create objects based on the syntactic representation of the program. In Java, syntactic creation points are statements instantiating objects with the keyword `new`. For instance a statement like `new Vector();` creates an object of the type `Vector`, and two such statements create disjoint sets of objects.

Analysis Precision

The precision of the analysis result is a measure of how close the analysis is to the exact solution. Conservative analyses, discussed below, ensures that the recall of the analysis is 100% at all times, i.e., all correct answers are part of the analysis result. To compare the precisions of two conservative analysis results with each other, it is only necessary to compare the analysis results sizes. A smaller result set contains less elements that are included because of imprecision in the analysis.

It is expensive, both in time and space, for an analysis to be precise. In order to be efficient, an algorithm may have to sacrifice precision. The trade-off between analysis efficiency and precision of its result is influenced by how detailed its program model is and how it models program executions. Program models include object-field-sensitivity and models of program executions include flow-sensitivity and context-sensitivity.

An object oriented program uses objects with fields. The fields may refer to objects, and it is these references points-to analysis calculates. An analysis that models both objects and its fields is field-sensitive. On contrary, an analysis that does not model object-field is know as a field-insensitive analysis. A field-insensitive analysis merges fields and do not separate accesses to the different fields of the same object. The field-sensitive analysis is a more precise model than the field-insensitive, and it enables the analysis to be more precise.

A flow-sensitive analysis ensures that the data- and control-flow is preserved in the analysis. Preserving control-flow in a program analysis means that operations are never influenced by operations that occur later in all executions of the program. The data-flow is dual, but instead of control-flow it models data-flow.
Conservative Analysis

It is desirable that the results an analysis provides hold for all possible executions of the analyzed program. An analysis is called conservative if the analysis result contains analysis values that may be part of the exact result of a particular program execution for a particular input. The most conservative analysis would return all possible analysis values as the analysis result. Obviously, this is not very useful, since we have learnt nothing from this, even though it is correct. The challenge in static analysis is to produce a result that is precise but still conservative.

To be conservative when analyzing a program, we need to follow and analyze all possible execution paths. For example, we cannot select only one branch in an if-statement, we have to assume that any of the possible paths may happen. If a method call has several possible target methods, i.e., through method polymorphism, all these possible methods need to be analyzed in order for the analysis to stay conservative.

Using points-to analysis, the number of paths may be reduced by having better precision in the analysis. With higher precision less objects will appear in certain program points and methods need not be analyzed as often because of polymorphism if there are less objects.

Monotone Data Flow Framework

The Monotone Data Flow Framework is a framework that solves data flow problems [NNH99]. The analysis is performed on a representation of a program, where the nodes are statements and the edges represent the control-flow between the statements, e.g., a Basic Block graph or an SSA graph. This has to be a connected graph; otherwise there will be nodes that cannot be analyzed. The analysis uses a fix-point algorithm to find the least solution of the specific problem, given the current instantiation. An instantiation consists of five analysis elements, a merge operator, a control or data flow, a set of starting nodes, initial analysis information and a set of transfer functions. These will be explained individually in the remainder of this section.

The analysis starts in a start node with a specific initial analysis value as input. The start node is taken from the set of starting nodes and the analysis value comes from the initial analysis information, both elements of the framework instantiation. The contribution the start node makes to the analysis result is calculated using a transfer function associated with that node’s type. The output analysis result of the start node has now changed and hence also the start node’s successor nodes’ input values. Now, these successor nodes’ transfer functions are invoked to calculate the contribution these nodes have on the analysis result. Each node has an analysis result associated with it. The analysis result can be considered to be attached to the out-port of the node, i.e., the out-edges. The values on the out-ports are
2.3. Static Analysis

propagated to the in-ports of the successor nodes. When a node’s in-port value changes, its transfer function should be recalculated.

An instantiation of the framework needs a definition of the property space, i.e., which analysis results are possible, as well as a combination operator. The combination operator is used to merge analysis results where several edges act as in-edges to a \( \phi \)-node. Different analysis results may be propagated on each of the incoming edges and merged. The property space should, generally, be a complete lattice. It is also possible that the property space fulfills the ascending chain condition. Descriptions of these three concepts follow.

A Complete Lattice is a partial ordered set, with the restriction that each subset should have a least upper bound and a greatest lower bound.

The Least Upper Bound of two elements \( s, s' \) of the partial ordered set \( S \) is the lowest element in \( S \) such that it is greater than each of the elements \( s, s' \). The dual is called greatest lower bound.

The Ascending Chain Condition is true for a partially ordered set if every ascending chain of elements \( x_0 \leq x_1 \leq \ldots \) eventually stabilize. The ascending chain stabilizes if there is an \( m \) such that \( a_m = a_n \) for all \( m < n \). All partially ordered sets with finite size fulfill the ascending chain condition.

The analysis can also be performed as a backward analysis, instead of the standard forward analysis. The flow element of the instantiation defines the flow. In a backward analysis the calculations transfer results against the edges in the control-flow graph; the transfer functions use the values on the nodes’ out-ports as input and transfer the result to their in-ports. The transfer functions need to be monotone to ensure that the analysis algorithm finishes. This means that the analysis result may only get more information, become larger in the lattice, i.e., more imprecise after a call to a transfer function. It is not possible to remove a previously added analysis result.

The described approach to solve data flow problems only works on toy languages, since it is an intraprocedural analysis; it is not able to analyze programs with procedures and functions. It is possible to make additions and make the analysis interprocedural, i.e., allow the use of procedure and function calls in the analyzed language. When introducing procedure calls into the analysis new types of nodes are introduced, the call nodes. The procedures are each given a start node and an exit node. The call node is connected to the called procedure’s start node and the procedure’s exit node will be connected back to the call node. There are two transfer functions for the call node and two transfer functions for the procedure. The call node’s transfer functions correspond to calling the procedure, and returning from the
procedure, respectively. The procedure’s two transfer functions correspond to entering and exiting the procedure body, respectively. This structure does not ensure that analysis results from one call node are only returned to that particular call node. Since the return node has many call nodes as successor nodes, these will be updated as well. Even though they are unrealizable paths, they still preserve the conservative property of the analysis. Context information reduces the number of unrealizable paths that are analyzed. A context can be formulated in many ways, and the simplest is an encoding that enables only valid paths to be analyzed, e.g., by encoding from which call node the call came. Each context that is valid for an analyzed method is mapped to the analysis results for that method. This ensures that the analysis results are not mixed and that only valid paths are analyzed.

Points-to Analysis

Points-to analysis has existed for some time, with the origin in alias analysis for imperative languages, such as C. First we present two of the algorithms that are considered traditional and then we present an algorithm called Simulated Execution, which is the analysis used in our framework. The analysis values typical for points-to analysis are also discussed.

Traditional Algorithms There are two algorithms that are considered traditional, Andersen’s and Steensgaard’s; they are referred to as the baseline approaches in literature. Even though the two algorithms target programs written in the programming language C, their approaches can be adapted to other languages, as well.

Andersen’s algorithm is a points-to analysis targeting the C programming language [And94], which is inter-procedural, flow-insensitive, context-sensitive. It is constraint-based and includes two major steps, creating the constraint system and solving the constraint system. All statements in the language contribute to the constraint system. The analysis is a whole-program analysis on static call graphs. The analysis results consist of method summaries as well as program point results. The method summaries are used to speed up the analysis and save memory. When a method is called many times this approach reduces the quality of the results and the algorithm may separate different calling contexts instead of creating summaries. The analysis result is a points-to graph, where nodes model variables and objects, i.e., heap-allocated memory, and edges model the relation points-to. A node may point to many nodes and may be pointed to by many nodes.

Steensgaard’s algorithm is an almost linear solution to the points-to problem of the C programming language. It is a constraint-based algorithm that uses type inferences to solve the constraint system [Ste96]. Most of the approaches taken in the algorithm are identical or similar to Andersen’s ap-
2.3. Static Analysis

The analysis result is a points-to graph, where nodes model variables and objects and edges model the relation points-to. The difference to Andersen’s approach is that nodes may only have one out-edge and nodes may represent an arbitrary number of variables. This reduces the number of nodes and edges and enables the algorithm to perform in almost linear time and memory, while obviously sacrificing precision.

Simulated Execution Simulated execution is a variant of the traditional data flow approach for performing a points-to analysis, and was first introduced in [LL07]. This is the algorithm used for the points-to analysis in our framework. The difference from previous approaches lies mainly in the way the analysis values are propagated through the program graph. This method simulates an execution of the analyzed program, in the sense that it follows the method calling sequence as it would be when the program is executed. The analysis of a method $m$ is interrupted when a call to method $n$ is reached. The analysis of method $m$ continues when the analysis of method $n$ is completed. The results of the analysis of method $n$ are used when the analysis continues with method $m$. A key issue to solve for this approach to be successful is to find appropriate conditions for when to stop to process calls. If the analysis is not interrupted it will iterate endlessly when there are recursive calls in the analyzed program. If an analysis of a method using specific input values does not provide any new analysis results, the method will not be analyzed using the specific input values again. This ensures that the analysis terminates, even for recursive calls.

The analysis iterates over loops to stabilize them before continuing with following parts of the program. Inner loops are stabilized before outer loops. A loop is stabilized when the analysis reaches a fix-point over the analysis values in the loop.

Analysis Values The result of a points-to analysis is two-fold, a points-to graph and a points-to decorated program graph. The precision of the analysis result is influenced by the sensitivity of the analysis. A typical field-sensitive, object-sensitive points-to graph has objects and fields as nodes and the edges represent the relations has and refer-to. An object node has fields, and field nodes refer-to objects. The analysis values used in the points-to analysis, and that decorate the program graph, are sets of abstract objects that may occur in different parts of the analyzed program. When the analysis is finished there are analysis values on each ingoing edge and outgoing edge, or in the case of Points-to SSA, on each in-port and out-port.
Name Schemata
Reasoning about objects requires that run-time objects are abstracted to abstract objects, which may represent an arbitrary number of run-time objects. Without this abstraction the number of objects needed to be modeled would not be known statically and could theoretically be infinite and the analysis would never terminate. Different granularities, called name schemata, of this abstraction exist, such as the class schema and the allocation schema. Some of the existing name schemata are:

Class Schema is when each class is an abstract object. All objects of a certain class is mapped to the same abstract object.

Allocation Schema is when each syntactic creating point in the analyzed program models an abstract object.

Context-sensitive Allocation Schema is when the allocation is context-sensitive, i.e., the syntactic creation points are separated depending on the call context they occur in.

Exclude Schema is when we use the allocation schema for all objects, except for the objects instantiating specific classes. In Java, e.g., it makes sense to treat the following classes specifically:

- java.lang.String
- java.lang.StringBuffer
- java.util.SimpleTimeZone
- java.util.Locale
- java.lang.Integer
- java.lang.Double
- java.lang.ref.*
- subtypes of java.lang.Throwable
- subtypes of java.lang.Error

We use the class schema for objects of these special classes.

Exclude Strings Schema is when we use the allocation schema for all objects, except for objects instantiating the classes StringBuffer and String. We use the class schema for those objects.

Object Schema is a variant of the context-sensitive allocation name schema. The call context that is used to separate objects is a notation similar to a call string with depth $k$. Objects that are not replicated over contexts are treated in a context-insensitive fashion instead of separated by different calling contexts.
2.3. Static Analysis

Inter/Intraprocedural Analysis

In static analysis of imperative and object-oriented languages, there is a separation between intra- and interprocedural analysis. It is necessary to specify how each method/procedure is analyzed separately, i.e., intraprocedural analysis, as well as how the analysis handles calls between methods/procedures. Performing intraprocedural analysis with high precision is not as hard and resource consuming as adding the interprocedural analysis aspect. The complexity grows when methods/procedures are connected through calls. Two basic ways of connecting these method graphs to represent method calls are *inlining* and *graph connecting*; inlining creates new copies of the graphs and graph connecting creates a connection between call nodes in the methods and target method graphs. The inlining method results in an exponential explosion of method copies, and it will be infinite if recursive calls exist. The analysis is now reduced to an intraprocedural analysis, since the whole program can be seen as one huge method and is represented by a single graph. The graph connecting method also results in a single graph that can be analyzed using intraprocedural techniques but does not explode in space as the inlining method does. However, the analysis will be very imprecise, since it is not guaranteed that the analysis results from one method execution only comes back to the caller. The analysis results of many calls are mixed, and this degrades the analysis precision. Other techniques that used to get better precision, such as call contexts, are described below.

Call Context-sensitivities

Call contexts are used to overcome the shortcomings of the inlining and the method copy approaches. A call context is an abstraction of the call stack to separate calls to a specific method. It can be based on, for instance, the $k$ last call stack entries, or the arguments passed to the called method. In a context-sensitive interprocedural analysis, all method calls are clustered according to some scheme. The granularity of the clustering will influence the number of contexts and thus the space, time and precision properties of the analysis. A call context scheme that produces a large number of call contexts will theoretically have a higher precision, but take more time to perform and require more memory. Some of the different call context schemata that are defined in literature and that we use in our analysis are:

**CallString** A call context is defined using the call history, i.e., the return address entry of the call stack.

**Object** A call context is defined for each abstract object a certain method is called on, i.e., $c \rightarrow (o, m)$. 

19
Chapter 2. Foundations

**This** A call context is defined for each set of objects a certain method \( m \) is called on, i.e., \( c \rightarrow (\{o|o \in O_m\}, m) \).

**ThisArgs** Same usage of abstract object as in *This*, but now all positions in the argument list are used, not only the first.

**ObjectArgs** Same usage of object as in *Object*, but now all positions in the argument list are used, not only the first.

### 2.4 Summary

In this chapter we have presented some general definitions, four program representation and a large number of notions and concepts regarding static analysis. The general definitions presented definitions of points-to analysis and framework.

The four program representations that we present are Abstract Syntax Tree (AST), Basic Block graph, Static Single Assignment (SSA) graph and Sparse SSA graph. The Sparse SSA graph has two properties that are desirable, namely that it models the def-use relation and that it contains only information required by the analysis algorithm. This is why we chose the program representation to be Spare SSA graphs.

The section on static analysis discusses analysis precision, conservativeness, the Monotone Dataflow Framework, algorithms and analysis values related to points-to analysis, syntactic creation points, name schemata, intra/interprocedural analysis, and call context-sensitivities. These are all relevant concepts in the field of static analysis.
Chapter 3
Related Work

The previous work that relates to the client analysis framework is presented in this chapter. It is also compared to a baseline approach regarding the goal criteria presented in Chapter 1. The approach used in the client analysis framework presented in Chapter 4 is related to the baseline in the conclusions chapter, Chapter 6. This chapter is divided into five sections. The first section describes how we relate the related work to the goal criteria. The second section discusses three frameworks that are similar to the client analysis framework. The third section presents two client analyses and the previous research related to them. The client analyses are implemented in the client analysis framework. The fourth section contains a summary and a categorization of the work presented in the second section and a list of important concepts presented in the related work. The fifth section draws conclusions from the related work.

3.1 Relating the Related Work

In the presentations of related work in this chapter, we give short comments that relate to the goal criteria we have in this thesis. The comments regarding development speed look at the existence of a framework idea as signs of high development speed, i.e., the level of development support and possible reuse are high; and the analysis speed and precision.

When relating the development speed criteria, some of the presented related approaches partly fulfill the framework idea. This is because they clearly separate the different participating analyses and let their analysis be parameterized in some way, for instance with different points-to analyses. These are awarded an ‘S’, short for Simple framework, to distinguish them from related work that does not present any signs of a framework or reuse.

We establish a baseline approach and compare the related work to this approach, regarding the goal criteria. We use the symbol ‘+’ when the compared approach performs better than the baseline. The symbol ‘−’ is used when the baseline approach performs better than the compared approach, and the ‘0’ denotes that there is no difference between the two compared approaches. Sometimes it is not possible to make a distinction, usually because
Chapter 3. Related Work

of a too imprecise or too short presentation of the specifics of the related work’s approaches and methods. We consider these cases to be undecidable and denote them with ‘?’.

The criteria of the analysis precision and speed are divided into six characteristics that are all related to the tradeoff between these criteria. The six characteristics form a comparison vector with six positions, each given a mark out of ‘+’, ‘0’, ‘−’ and ‘?’.

The order of the characteristics are as listed above, with the ‘+’ as the highest and the ‘?’ as the lowest. For a vector \( v \) to be larger than a vector \( u \), all elements in \( v \) has to be larger than or equal to the corresponding elements in \( u \), with at least one element being larger (analogously for smaller). The comparison vector is written after each presented client analysis and used in a summary that can be found in Section 3.5.

The six characteristics are the following:

**Allocation schema** is the granularity regarding how abstract objects are modelled, whether it is objects or classes that are modelled. If the allocation is more precise it results in more objects for the analysis to consider and the analysis speed may suffer, while the precision may improve.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>uses a context-insensitive allocation schema based on syntactic creation points</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>when used allocation schema is more precise, e.g., context-sensitive</td>
</tr>
<tr>
<td>0</td>
<td>when same allocation schema is the same as for baseline</td>
</tr>
<tr>
<td>-</td>
<td>when used allocation schema is less precise, e.g., class-based</td>
</tr>
</tbody>
</table>

**Context-sensitivity** is how different calls to the same method are separated. A more precise context-specification results in more contexts to consider and the analysis may be slower, but more precise.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>uses object-sensitivity as its most precise context-sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>when used context-sensitivity is more precise, i.e., more contexts may be distinguished</td>
</tr>
<tr>
<td>0</td>
<td>when same context-sensitivity is used</td>
</tr>
<tr>
<td>-</td>
<td>when a less precise context-sensitivity is used</td>
</tr>
</tbody>
</table>

**Object-sensitivity** is whether objects are modelled as a collection of all its fields or if the fields are separated. Analyses that consider object fields are more precise, but the analysis speed will suffer.
3.1. Relating the Related Work

<table>
<thead>
<tr>
<th>Baseline</th>
<th>models each field of an object separately</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>when used object model is more precise than baseline, e.g.,</td>
</tr>
<tr>
<td>0</td>
<td>when same object model as baseline is used</td>
</tr>
<tr>
<td>−</td>
<td>when used object model is less precise than baseline, e.g., field-insensitive</td>
</tr>
</tbody>
</table>

**Program representation** is the efficiency in the representation of the analyzed program. A problem adapted representation enables more efficient processing and storage.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>uses a Sparse SSA graph, optimized for points-to analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>when used program representation is more optimized for the analysis purpose than baseline, e.g., more information not related to the specific client analysis is removed from the program representation</td>
</tr>
<tr>
<td>0</td>
<td>when used program representation does not provide any advantages compared to baseline</td>
</tr>
<tr>
<td>−</td>
<td>when used program representation is less optimized than the one baseline uses</td>
</tr>
</tbody>
</table>

**Heuristics in favor of speed** may be used, which usually results in loss of precision. In some cases the heuristics are used to ensure that the analysis terminates, especially in the case of recursive calls in the analyzed program.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>does not use any particular heuristics in favor of speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>is not used for this characteristic, since we are only looking for usages of heuristics that degrades precision in favor of speed and not using a heuristic is awarded ‘0’</td>
</tr>
<tr>
<td>0</td>
<td>when no heuristics are used in favor of speed having negative effect on precision is used</td>
</tr>
<tr>
<td>−</td>
<td>when heuristics are used in favor of speed, that has a negative effect on precision</td>
</tr>
</tbody>
</table>

**Flow-sensitivity** is whether the control and data flow of the analyzed program is considered in the analysis.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>uses control flow- and data flow-sensitive program representation and analysis approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>when used flow-sensitivity is more precise than the one used by baseline</td>
</tr>
<tr>
<td>0</td>
<td>when same flow-sensitivity is use as baseline</td>
</tr>
<tr>
<td>−</td>
<td>when less or no flow-sensitivity is used</td>
</tr>
</tbody>
</table>

The vectors are written as \(| + | 0 | 0 | - | ? | + |\), where the positions correspond to the characteristics listed above, using the same ordering, starting
Chapter 3. Related Work

A summary of the presented client analyses is given. Each client analysis is summarized with respect to how it meets the research criteria development speed and the trade-off between analysis precision and speed. The vectors are transformed into a grade for analysis precision and one for analysis speed. The grades are the same as in the vector itself, i.e., ‘+’, ‘0’, ‘−’ and ‘?’.

3.2 Static Analysis Frameworks

This section presents three static analysis frameworks, the Monotone Data Flow Framework, PAG and CoSy. It discusses how they compare to the baseline approach. They are related to the client analysis framework in Chapter 6.

Monotone Data Flow Framework

The monotone data flow framework, presented in Section 3.2, provides more of a design for the implementation of a data flow problem solver than help with the implementation directly. If the requirements on an instantiation are fulfilled, such as correctly defined monotone transfer functions, then it ensures termination of the analysis, etc. The specific details for how to implement data structures and algorithms to ensure efficiency in space and time are not given by the framework. An attempt to close the gap between this theoretical framework and a realistic solution is the Program Analyzer Generator (PAG) framework we present below.

The generality of the framework ensures that it can be instantiated to solve many data flow problems. However, the generality and the need for an instantiation degrades the development speed of the framework. There is no support for any algorithms or data structures. They have to be implemented by hand. In principle, it is possible to implement any sensitivity, and the speed is all up to the implementation of the algorithms. Therefore, it is not possible to say anything about the analysis speed.

PAG

Program Analyzer Generator (PAG) is a generator framework for program analysis. It can be seen as an implementation of the monotone data flow framework discussed before. It can generate program analyzers performing
3.2. Static Analysis Frameworks

Interprocedural analysis [Mar98]. The analyzers can be integrated into compilers through the use of an interface connected to generated analyzers. PAG is based on the theory of abstract interpretation, and aims to provide a framework that makes it easier and faster to implement program analyzers. The user is not bothered with details about the traversal of control-flow graphs or with details about the fix-point iteration algorithm. PAG is implemented in a modular fashion that makes it possible to change parts of it for other implementations. The fix-point algorithm may for instance be replaced with a user-defined version [Mar98].

The specification for an analyzer to be generated by PAG consists of three parts: one for data structures, one for the main structure of the analyzer and one for the transfer functions. A selection of iteration algorithm can be made as well. If we see PAG as an instantiation of the monotone data flow framework, PAG’s data structures represent the program, i.e., the flow and the start nodes, PAG’s transfer functions are mapped to the monotone data flow framework’s transfer functions, and PAG’s iteration algorithm implements the merge operator and provides with the initial analysis information.

The data structures are described in a language called DATLA, in which one may specify value sets and value lattices for the analyzer domain. The transfer functions are written in a programming language called FULA, a functional language that borrowed many constructs from ML [GHT84, NNH99]. Transfer functions are mapped to a node-edge-type pair. Some characteristics of the analyzer is specified separately. These characteristics include analysis direction (forward or backward), widening and narrowing. The control flow graph that is used in PAG is an extended basic block graph, where nodes are basic blocks instead of single instructions in order to save space and computation time. It is capable of performing inter-procedural analysis using the effect calculation and the static call graph approaches. These approaches may simulate three other approaches that are available, namely inlining, invalidating and call string.

The techniques that are used in PAG to speed up the computations are recalculation avoidance; widening and narrowing; and the use of basic block graphs [Mar99]. Only edges with changed values cause recalculation. Widening and narrowing are functions that may be used in the fix-point algorithm to speed up the computation. The basic block graph speeds up the computations, since nodes are combined into blocks and some operations need only be calculated in the block, rather than on each node.

CoSy

CoSy is a compiler development system which is highly flexible and has been used to create numerous compilers in the industry targeted at various hardware architectures. It has a modular design, an extensible intermediate rep-
presentation and makes use of generators in different steps.

CoSy provides support at different levels when developing a compiler system. The lowest two levels, i.e., framework and component, provide all facilities that would be used in points-to analysis and its client analyses. On the higher levels site, person and project it only provides supportive facilities. On the framework level, tools and interfaces are provided, and on the component level even more facilities are provided in the form of interfaces, an individual generator, a functor library, support tools, etc.

A compiler is divided into a number of components, either provided by the framework or user-defined. The user may define engines that are used as components in the system. The components share a memory bank in the form of a repository for storing and exchanging data. Each component has its own view of this repository, where its required data are accessible. The data types include program representation structures, intermediate representations, and more.

To construct a compiler, it is necessary to specify a number of components: a micromenu, a main program, and some specification files, including engines specified by the Engine Description Language (EDL), full-Structure Definition Language (fSDL) and Compiler Configuration Language (CCL) files. The micromenu specifies which components to include and various options and settings for these components. The main program generates some engine specifications and implements some features according to the user options given in an options file. The EDL file specifies the logical decomposition of the compiler into engines. It is a kind of architecture description language for the compiler, with engines as components and interactions schemes (pipelines, data-parallel operations, etc.) as connectors. The SDL file specifies the reference level domains, how it relates to the intermediate representation language, and the view domain per engine. The CCL file specifies a mapping between the compiler’s mapping to host facilities, i.e., how resources should be used on the target host in order to make best use of them. An analysis of the control-flow of the EDL file makes sure that data parallelism is sustained and to make the engines sequential if needed.

### 3.3 Client Analyses

The two client analyses we have studied and implemented are escape analysis and side-effects analysis. Both exist in many variants targeting different programming paradigms and used as a basis for several other analyses and optimizations separately.

An object that is created in a method and then accessed after the method’s execution is said to escape the method. However, an object that is created in a method and not made available to the execution environment after the
method exits has not escaped the method. Escape analysis separates the objects created during the execution of a method into these two categories, escaping and non-escaping object. This information has shown useful in a number of compiler optimizations, such as stack allocation of objects [Bla03, Bla99, GS00, CGS+03, CGS+99, WR99] and synchronization removal [Bla03, Bla99, CGS+03, CGS+99, WR99, BH99].

A method with no side-effects does not affect the observable state the system is in when invoked. A method with side-effects is potentially more difficult to understand and more error-prone. It has been shown that side-effects analysis can be used for compiler optimizations [Raz99], program comprehension and documentation [LBR99, DR02], and software model checking [TD03, VHBP00, CDH+00].

**Escape Analysis**

Here we present different escape analyses that have been proposed in literature. The first three presented analyses are not relevant to compare to our approach since they target either the C language ([RM88]) or functional languages ([PG92], [Deu97]), but they are relevant since they contain original work when it comes to escape analysis. They present concepts that are still used, even when constructing analyses, for example, for the Java language. The remaining works presented here target Java and have been related to our approach. A summary of this comparison is listed in Table 3.1 in Section 3.5.

**Classic**  Ruggieri and Murtagh present a lifetime analysis [RM88] which determines an upper bound for the lifetime of a dynamically allocated data structure. The analysis can be used in object oriented languages. The upper bound of an object’s lifetime is associated with a function and the object can be allocated in the stack frame of this function instead of dynamically on the heap, which would eliminate the overhead that garbage collection introduces. Their method is based on formal lattice theory and is, according to the authors, extendible and can be used in other analyses.

Park and Goldberg present an escape analysis for higher order functional languages [PG92]. In their previous work, they presented an analysis that only worked for non-list objects, such as closures. In [PG92], they extend this analysis to also work with lists. Their analysis answers one question: “Given a function application, does a parameter (or some part of a parameter) get returned in the result of the application?” The answer can be used to optimize the execution of the application through stack allocation of list segments, in-place reuse of list segments, and allocation and reclamation of continuous memory blocks. They define a non-standard semantics called the escape semantics, such that this holds the exact escape information about the arguments. This is used along with abstract interpretation techniques
to answer the question. The analysis is performed on each argument of a function separately. It is shown that the analysis is polymorphically invariant.

The work of Park and Goldberg is reworked by Deutsch and the time complexity is reduced to almost linear \( O(n \log^2 n) \) for first-order escape analysis, i.e., analysis of the first-order part of the functional language [Deu97]. The improvements include a two-stage approach where the first stage is to construct a compact system of semantic equations, which is then solved using a fix-point algorithm, both using well-established algorithms.

**Java** [BH99] presents an inter-procedural, flow- and context-insensitive data flow analysis to identify situations where it is possible to remove unnecessary synchronizations in Java. The analysis is constraint-based and is performed in two different stages with different purposes. First, it detects \( s \)-escaping objects and then it detects \( f \)-escaping objects. The former are objects that are references from other objects on the heap, and the latter are objects transitively reachable from an \( s \)-escaped object. All objects that are not \( s \)- or \( f \)-escaping are considered to be thread-local and all synchronization operations can be eliminated from them. The empirical results the authors present show that the removal of the found unnecessary synchronization operations speeds up the execution of the tested applications. There is no explicit framework idea. The analysis does not have as high precision as the baseline approach has, since it is flow- and context-insensitive. The result of the comparison is \( |0| - |0| ? |?| - \).

Gay and Steensgaard present a constraint based analysis for locating objects in Java that escape a method [GS00]. The constraints may be solved in time and space linear in the number of constraints which are directly derived from the analyzed program. The number of constraints generated is linear in the size of the program plus the number of actual/formal parameter pairs at all call sites in the program. The data and control flow of the program is represented by SSA graphs. The result of the analysis is used for stack allocation of objects that have a lifetime that does not exceed a method’s runtime stack frame. The data flow and control flow information comes from an SSA (Static Single Assignment) graph. Empirical results from a number of Java systems are presented that show that the number of objects that can be stack allocated increase due to the analysis. The authors do not present an explicit framework idea. Their approach does not use as precise context specification as the baseline does and they use method summaries, which degrades the precision while giving higher speed. The result of the comparison is \( |0| - |0| |0| - |0| \).

Whaley and Rinard [WR99] present an analysis for Java that produces a points-to escape graph which is a points-to graph with objects as nodes and object references as edges. Both nodes and edges can be of two types, inside or outside. Inside nodes model objects created inside the analyzed
3.3. Client Analyses

region, while the outside nodes model objects created outside the analyzed region. An inside edge model a reference created inside the region, and analogously for the outside edge. If an inside node is referenced with an inside edge from an outside edge, it has been made available to the system outside the analyzed region and has thus escaped. The analysis uses a compositional approach where each method is analyzed individually and the result of an analyzed method can be parameterized in order to get the right information for a specific calling context. This enables the analysis to be used without analyzing the whole program. The analysis result is used to remove synchronizations and to stack allocate objects in Java programs. In [VR01], this analysis is extended to an incremental analysis algorithm. Instead of analyzing the whole program, the analysis uses a policy to focus on parts of the program, which have the highest potential to contribute to the optimization results. The policy bases its decisions on empirical data from previous analyses, the current analysis and profiling data. There seems to be no framework idea, since the escape information is interleaved with the points-to information in the analysis results. They use method summaries, which degrades the precision while giving higher speed and their contexts are not as precise as the baseline is. The result of the comparison is 

Rountev et al. present a points-to analysis for Java based on annotated inclusion constraints [RMR01]. The annotations on the constraints make it possible to get higher precision in the analysis than it would be using unannotated constraints. Their approach is built on Andersen’s algorithm for points-to analysis of the C language. They provide appropriate extensions to this algorithm in order to handle the object-oriented features of Java. They list a number of client analyses of the points-to information, and describe how they use points-to information. Their points-to algorithm is evaluated using these client analyses. Escape analysis is included twice in the list, once to remove synchronizations and once for stack allocation of objects. Both techniques involve a traversal of the points-to graph in order to find the objects that are thread local and local to a method. Thread local objects need not be synchronized on and method local objects can be stack allocated. There seems to be no framework idea, since the escape analysis works intimately on the points-to result data structure. Their approach does not use as precise context specification as the baseline does. The result of the comparison is 

Bruno Blanchet discusses an escape analysis [Bla99, Bla03] able to analyze Java byte code. The analysis is based on an alias analysis which identifies the objects aliased by object fields. The result is a directed graph with abstract objects as nodes, and fields as edges, i.e., similar to a points-to graph. The analysis is context-sensitive and flow-sensitive, since an SSA graph is used as program representation. The escape analysis is an abstract interpretation analysis, using both forward and backward passes to generate
the correct set of equations. The equations are later solved by an iterative fixed point solver. The analysis result is used to stack allocate objects and to remove unnecessary synchronizations, which is the actual goal of the analysis. Objects that do not escape a method are stack allocated and synchronizations are removed from classes of objects that are not reachable from objects of classes extending the Thread class. There is no explicit framework idea, but the escape analysis is not integrated with the points-to analysis, which indicates that there may be an intention for reuse. Their approach does not use as precise context specification as the baseline does. The result of the comparison is $|0| - |0|$, $|0|$. 

Choi et al. present analyses finding objects that escape a method or a thread [CGS+99, CGS+03]. This is done by introducing a connection graph which contains the points-to edges between reference nodes and object nodes; deferred edges between two reference nodes; field edges from an object node to a reference node. Object nodes are associated with an escape state, being one of NoEscape, ArgEscape and GlobalEscape. Objects marked with NoEscape do not escape their creating method, while objects marked with ArgEscape are stored in fields of objects reachable from the arguments of the method call or from the objects the method returns. Objects marked GlobalEscape are reachable from static fields of classes or from Runnable objects, i.e., threads. NoEscape objects may be stack allocated and ArgEscape objects are subject to synchronization removal. When the connection graph is created, a reachability analysis is performed on the graph; objects reachable from GlobalEscape objects are marked as GlobalEscape; objects reachable from ArgEscape objects are marked ArgEscape if not already marked GlobalEscape; all other objects are still marked NoEscape. The authors also present a simple inter-procedural framework for flow-insensitive and flow-sensitive escape analysis. They exploit the escape analysis result, not only by stack allocation of non-escaping objects, but also in a synchronization removal analysis which removes unnecessary synchronizations which results in performance speedup of Java programs. To handle recursion, they use a maximum of iterations for the fix-point algorithm. When this limit is reached, the conservative analysis result is calculated, which degrades the analysis precision. There seems to be no framework, since the escape analysis works intimately on the connection graph. There is a parameterization regarding the flow-sensitivity, but this is not substantial enough to say that there is a more extensive framework. Their approach does not use as precise context specification as the baseline approach does and they use method summaries, which degrades the precision while giving higher speed. The limitation on iterations in the fix-points algorithm also degrades the analysis precision. The result of the comparison is $|0| - |0|$, $|0|$. 

Whaley and Lam [WL04] present a context-sensitive points-to analysis using cloning of contexts, the programming language Datalog, and a translation
engine that creates ordered\(^1\) binary decision diagrams (BDDs) [Bry86] implementations from Datalog code, called “bddbddb”. Their cloning technique creates a new instance of each method for each calling context. The analysis is then performed using an efficient context-insensitive approach. Contexts are defined using the call paths, i.e., all positions of the call stack trace, using certain reductions when it comes to cyclic stack traces. To handle the amount of contexts that reasonable large applications may have, the contexts are represented using BDDs and encoding schemes that make it possible to use commonalities between contexts. The authors also present applications to the points-to result in the form of, for instance, a thread escape analysis. This analysis finds objects that escape a thread, in order to perform synchronization optimizations, stack allocate objects and for understanding the synchronization behavior of programs. The analysis is done by cloning the \texttt{run()} method of the thread object, once for each thread context. Objects that are not accessed by its thread’s clones do not escape the thread and optimizations may apply to this object. There is no explicit framework idea, but the escape analysis is not integrated in the points-to analysis, which indicates that there may be a simple framework idea. They also allow the user to pose queries on the analysis results in order to create other client analyses. Information about how this is organized is not available in their presentation. We can only say that there is a possibility to produce client analyses, but not what kind of support is given, etc. They have a more precise context specification in theory, but in practice they reduce this number dramatically to save time and space. The result of the comparison is \( \text{[0] + [0] ? - [0]} \).

Hill and Spoto present two escape analyses for Java based on abstract interpretation [HS06]. They provide proofs that the associated transfer functions are optimal with respect to the abstractions of the two analyses. The first analysis uses a representation that is not field-sensitive, while the second analysis is field-sensitive. They analyze a number of applications and report both static and dynamic results. The authors claim that their solution does not aim for efficiency or precision and that the main contribution is a solution that is provably correct. They also claim that it scales well and is comparable in precision to some previous work. They make use of two different sensitivities in their analysis, which indicates that there may be a simple framework idea. Their approach does not use as precise context specification as the baseline does and the program representation used is not optimized for the problem. The result of the comparison is \( \text{[0] - [0] - [0] [0]} \).

**Side-Effects Analysis**

There has been quite some work done regarding side-effects analysis and related problems, e.g., what is known as the modification side-effects prob-

\(^1\)A variant on the BDDs introduced by Lee [Lee59]
Chapter 3. Related Work

lem [MLR+93]. These are typically directed towards procedural languages and are not directly relevant to our work [Bar78, CK88, Ban79, Bur90], since they are not concerned with object-oriented languages constructs such as pointers and interprocedural analysis techniques. The first approaches performing reference analysis and side-effects on languages with references were presented in [CBC93, MRL+93]. It is hard to evaluate this work, since neither their algorithm nor implementation results are presented. Later, other approaches were presented where the side-effects analysis uses reference information to produce a result [LRZ93, ZRL96, ZRL98, HP00, RLS+01]. Others use the side-effects analysis as a client analysis to motivate the results of their points-to analysis [SH97, GH98]. The rest of this section will present approaches to find side-effects in object oriented languages relevant to us, most of them directed towards Java. A summary of this comparison is listed in Table 3.1 in Section 3.5.

C A framework for modification side-effects (MODc) analysis is presented in [RLS+01]. The framework is capable of analyzing programs written in C, rather than Java. However, it is interesting for us to study it because of its framework properties. The authors present two variants of side-effects analysis, both making use of points-to information, FSAlias and FIAlias. The FSAlias uses a flow- and context-sensitive points-to algorithm from [LR92] and the FIAlias uses a flow and context-insensitive algorithm from [ZRL98]. The MODc schema is independent of the points-to algorithm, which can be exchanged, given a suitable interface. The flow- and context-sensitivity is propagated to the MODc method from the points-to algorithm, which can be exchanged, given a suitable interface. The flow- and context-sensitivity is propagated to the MODc method from the points-to algorithm, which can be exchanged, given a suitable interface. The MODc algorithm is decomposed into sub-analyses, in the same way as in [LRZ93]. The extensive empirical results presented results in a comparison of FSAlias and FIAlias and how their sensitivities influence the precision and efficiency of the analysis. The authors present a simple framework idea. It is not possible to say anything about precision or speed, since they target another programming paradigm than we do.

Java Clausen presents a side-effects analysis for Java which is interprocedural, where possible [Cla97]. Instructions are given, in relation to object fields and array elements, as one out of four different properties: pure, read-only, write-only and read/write. A pure instruction does not inspect or modify any fields or array element; a read-only instruction reads from a field or an array element; a write-only instruction writes to a field or an array element; and a read/write operation both reads and writes to a field or an array element. These four properties are partially ordered, with read/write as the top element and pure as the bottom element. The local purity property for a field is found by the analysis through calculation of the least upper bound in the property space. This analysis is very conservative, since
it neither separates on objects nor on fields. The author presents three more fine-grained analysis variants, namely, *subclassing purity*, *class-based purity* and *field-based purity*. The *subclassing purity* uses the least object that all affected objects are subclasses of. The *class-based purity* maps the property to the class of the affected object, and the *field-based purity* to the field that is affected. The analysis does not separate on objects in any case, not even in the field-based purity analysis. The analysis granularities suggest that there is a simple framework idea in their work. Their context-sensitivity is less precise than the baseline approach and the object representation as well. They are also using heuristics that gain speed but lose precision. The result of the comparison is | ? | − | − | − |

Razafimahefa presents a side-effects analysis that uses points-to results from a flow- and context-insensitive analysis under a closed world assumption, i.e., that all possible classes are known at compile time [Raz99]. Two granularities are presented for the points-to analysis, one considering objects opaque, and one considering fields within objects. The side-effects analysis inspects each statement and decides, which objects are read or written by this statement. This result is used to optimize Java code through *loop-invariant removal*. The author concludes, from the empirical results retrieved from performed experiments, that side-effects analysis has an impact and that the field-based approach renders results with higher quality than the approach that considered objects opaque. There is no explicit framework idea presented by the authors. The analysis is flow- and context-insensitive and the program representation is not optimized for the analysis. The result of the comparison is | 0 | − | 0 | − | 0 | − |

Tkachuk and Dwyer present an analysis framework [TD03] able to perform points-to and side-effects analysis adapted for modular program model checking. The approach aims to automatically extract abstract behavioral models, which model environment assumptions. This would limit the manual work needed to annotate these assumptions, which is the default approach in other methods. A unit is a set of components that should be tested, and the environment is the other components in the system. The approach uses a side-effects analysis based on a flow- and context-sensitive points-to analysis to calculate summaries for how the environment may modify the unit data. These summaries drive the generation of an abstract Java program, which is the units’ code combined with code models as a result of the analysis approximations. This generated program can be submitted to an existing model checking framework to test the properties of the unit. The side-effects analysis is based on the one presented in [LRZ93]. They introduce two new concepts, *return sensitive* and *must* side-effects. The *return sensitive* addition separates results from the different return sites of a method, instead of merging these results, as previous approaches do. The *must* addition identifies the side-effects that happen during all invocations of a method. These
two new concepts are used to improve the precision of the analysis. There is no explicit framework idea presented by the authors. The analysis uses method summaries, which degrades the context-sensitivity. Their program representation is not optimized for the analysis and it does not enable optimized calculations. The result of the comparison is $|0| - |0| - |0| - |0| - |0|$.

Whaley and Lam [WL04] present a context-sensitive points-to analysis using cloning of contexts. Their work is presented in more detail in the previous section about escape analysis. The analysis is also evaluated using a side-effects analysis, not only the escape analysis. The side-effects analysis uses the points-to results provided by the cloning-based points-to analysis. The side-effects analysis inspects the store instruction of the analyzed method $m$ and the instructions that may be reached when calling the $m$. If there is a store such that a field of object $o$ is the target then $m$ can modify $o$. The result of the analysis is sets of modifiable objects per analyzed method. There is no explicit framework idea, but the escape analysis is not integrated in the points-to analysis, which indicates that there may be a simple framework idea. They have a more precise context specification in theory, but in practice they reduce this number dramatically to save time and space. The result of the comparison is $|0| + |0| ? |0| - |0|$.

Rountev presents an approach that finds side-effect free methods using a technique that works on incomplete programs and can be parameterized with class analyses in [Rou04]. Two class analyses are used; a Rapid Type Analysis (RTA) from [BS96] and a points-to analysis from [MRR02]. The technique to analyze incomplete programs creates a virtual Java main method that calls the methods that should be analyzed. Now, the main method can be analyzed using a whole-program analysis engine. The order of statements in the main method is supposed to be of no interest since the technique uses a flow-insensitive analysis algorithm. The analysis is performed on a number of libraries and the results show that the difference in precision between the use of an RTA and the context-sensitive flow-insensitive points-to analysis is so small that it may be neglected. There is no explicit framework idea presented by the authors. The analysis has less precise context-definition and it is flow-insensitive. The result of the comparison is $|0| - |0| ? |0| - |0|$.

Nguyen and Xue present a side-effects analysis that is aware of dynamic class loading [NX05]. The analysis uses the results of a points-to analysis, which is based on the analysis presented by Lhoták and Hendren in [LH03]. An internal world is created which is composed of all classes that are known at compile-time. A whole-program analysis is performed using the classes in the internal world. An internal analysis (IA) is then performed to classify all references, objects, and call sites, as either complete on incomplete. Those classified as complete will not be affected by classes loaded dynamically, while the incomplete may be affected. There is no explicit framework idea presented by the authors. The analysis uses a less precise context-definition.
compared to the baseline approach and it is flow-insensitive. The results of the comparison is $0 \mid - \mid 0 \mid 0 \mid 0 \mid -$.

In [MRR02, MRR05], the authors present two analyses targeting Java, an object-sensitive points-to analysis and a side-effects analysis that uses its analysis results. The modification side-effect analysis (MOD), previously described in [RMR01], is shown to be significantly more precise using the object-sensitive approach compared to the context-insensitive version. They also provide a parameterization framework which enables analysis designers to choose the tradeoff between precision and efficiency to suit the specific needs. The parameterization framework lets the analysis developer tune the analysis in two dimensions: size of the object name schema and that objects should be replicated over contexts. Specifying the depth of the object name schema gives the analysis developer a chance to influence the precision that should be used. The authors have a simple framework idea, since they let the side-effects analysis be independent of the points-to analysis and parameterized the points-to analysis. The precision span they have is wider than the baseline approach has because of the object name scheme they are capable of. However, the analysis is flow-insensitive. The result of the comparison is $0 \mid + \mid 0 \mid 0 \mid ? \mid 0 \mid -$.

A so called purity analysis, which is an extended side-effects analysis, is discussed in [SR04, SR05]. The analysis is based on the combined escape and pointer analysis presented in [WR99]. The purity analysis makes use of the escape information this points-to analysis produces. It can determine which methods have no side-effects. It also retrieves more information on which side-effects a method may have. It allows a pure method to create complex object structures and even return them, as opposed to a side-effect free method. The authors also present the concepts of safe and read-only parameters, respectively, in a method, as a result of the analysis. A read-only parameter is only read in the method; a parameter is safe if it is read-only and the method does not write objects to the reachable object fields from the parameter. There is no explicit framework idea presented by the authors. The analysis has a less precise context-definition and they use heuristics to speed up the analysis. The result of the comparison is $0 \mid - \mid 0 \mid 0 \mid - \mid 0$.

3.4 Client Analyses Summary

Here, we categorize the client analyses presented in the literature according to analysis type, analysis design and what empirical results are presented and relevant variants/concepts. We also present a list of concepts and notation used by one or several authors, extracted from the client analysis articles.
Chapter 3. Related Work

**Escape Analysis**

There is a wide range of *analysis types* used for escape analysis. [GS00, RMR01] use constraint-based approaches, [Bla03, Bla99, HS06] use abstract interpretation, [BH99] uses a dataflow analysis, and [WL04] uses a cloning-based approach.

Most authors present an *analysis design* that integrates the escape analysis with the reference/alias/points-to analysis [WR99, BH99, GS00, Bla03, Bla99, CGS+03, CGS+99, HS06]. These are also analyses that separate the escape analysis from the reference/alias/points-to analysis [RMR01, WL04].

The *empirical results* presented by the authors are of four types: number of removed synchronizations [Bla03, Bla99, CGS+03, CGS+99], speedup caused by removed synchronizations [BH99], number of stack allocated objects [VR01, WR99, GS00, HS06, CGS+03, CGS+99], and number of escaped objects [WL04]. The three first are measures of the result of an optimization based on the escape analysis results. Only the last of the four is a direct measure of the escape analysis result.

Five concepts were identified in the literature presented earlier, that are interesting for our work, *NoEscape*, *ArgsEscape*, *GlobalEscape*, **s-escaping**, and **f-escaping**.

**NoEscape** is a label used for objects that do not escape a method [CGS+03, CGS+99]. These objects are candidates for stack allocation, if it is applicable.

**ArgsEscape** is a label used for objects that are stored in formal arguments to a method call [CGS+03, CGS+99]. They escape the method through the arguments.

**GlobalEscape** is a label used for objects that are stored in globally accessible variables, i.e., static fields [CGS+03, CGS+99]. They escape through the globally accessible variable.

**s-escaping** objects are objects stored on the heap that can be accessed by multiple threads [BH99]. An **s-local** object is only referenced by the thread’s stack frame, and not from any other object that is on the heap. s-local objects are only accessed by one thread and optimizations may be applied.

**f-escaping** objects are objects that are stored in fields of s-escaping objects or in a static field [BH99]. These objects escape the thread and optimizations may not be applied.
3.5 Conclusions

Side-Effects Analysis

Some approaches use a dataflow-based analysis type for the side-effects analysis [Cla97, Raz99, TD03, NX05], while [WL04] uses a cloning-based approach instead and [SR04, SR05, Rou04, MRR02, MRR05] use some sort of graph-traversing approach.

Some approaches use an analysis design that do not separate the points-to analysis from the side-effects analysis [NX05, SR04, SR05, Cla97], while others do [Raz99, WL04, MRR02, MRR05, Rou04, TD03].

The empirical results presented by the authors are of four types, a measure of side-effect free or pure methods (the count or in percent) [Rou04, SR04, SR05], the result of the loop invariant removal optimization [Raz99, Cla97], average object fields modified per statement modifying at least one object field [NX05], the number of modified objects per statement modifying at least one object [RMR01, MRR02, MRR05].

Two concepts were identified in the literature, return sensitive and must side-effects [TD03]. Both exist to increase the precision of the analysis.

Return sensitive analysis separates the analysis results for methods with multiple return points [TD03]. The analysis result is summaries per return point, and the analysis result for a method call is considered as the set of these summaries.

Must side-effects are side-effects that occur in every possible execution of the analyzed program [TD03]. For example, if a specific side-effect occurs on all branches of an if-statement it is a must side-effect for that if-statement.

3.5 Conclusions

There are a number of previous approaches presented in the client analysis section, Section 3.3, that are awarded the ‘+’ in one position in their comparison vector. However, they all have at least one – or ? in the vector as well. The result of this is that their comparison vectors are not comparable to the comparison vector of the baseline approach, since it is neither larger nor smaller and that the vectors form a partial ordered set. We can conclude that none of the approaches has an advantage over the baseline approach. The rest of the previous approaches have a comparison vector that is less than the comparison vector for the baseline approach, which indicates that they are less precise or computationally more demanding.

The conclusions regarding development speed is postponed until the conclusions chapter, Chapter 6.
Chapter 3. Related Work

### Table 3.1: Evaluation of the related work with respect to the goal criteria.

<table>
<thead>
<tr>
<th>Work</th>
<th>Criteria</th>
<th>Dev. Speed</th>
<th>Precision / Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDFF [NNH99]</td>
<td></td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>PAG[AMW95]</td>
<td></td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>CoSy</td>
<td></td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>[BH99]</td>
<td></td>
<td>−</td>
<td>0 − 0 ? 0 0</td>
</tr>
<tr>
<td>[GS00]</td>
<td></td>
<td>−</td>
<td>0 − 0 0 − 0</td>
</tr>
<tr>
<td>[WR99, VR01]</td>
<td></td>
<td>−</td>
<td>0 − 0 0 − 0</td>
</tr>
<tr>
<td>[RMR01]</td>
<td></td>
<td>−</td>
<td>0 − 0 ? 0 0</td>
</tr>
<tr>
<td>[Bla99, Bla03]</td>
<td>S</td>
<td></td>
<td>0 − 0 ? 0 0</td>
</tr>
<tr>
<td>[CGS+99, CGS+03]</td>
<td>−</td>
<td></td>
<td>0 − 0 ? − 0</td>
</tr>
<tr>
<td>[WL04]</td>
<td>S⁺</td>
<td>0 + 0 ? − 0</td>
<td></td>
</tr>
<tr>
<td>[HS06]</td>
<td>S</td>
<td>0 − 0 − 0</td>
<td></td>
</tr>
<tr>
<td>[Cla97]</td>
<td>S</td>
<td>? − − ? − 0</td>
<td></td>
</tr>
<tr>
<td>[Raz99]</td>
<td>−</td>
<td>0 − 0 − 0</td>
<td></td>
</tr>
<tr>
<td>[TD03]</td>
<td>−</td>
<td>0 − 0 − 0</td>
<td></td>
</tr>
<tr>
<td>[WL04]</td>
<td>S⁺</td>
<td>0 + 0 ? − 0</td>
<td></td>
</tr>
<tr>
<td>[Rou04]</td>
<td>−</td>
<td>0 − 0 ? 0 −</td>
<td></td>
</tr>
<tr>
<td>[NX05]</td>
<td>−</td>
<td>0 − 0 0 0</td>
<td></td>
</tr>
<tr>
<td>[MRR02, MRR05]</td>
<td>S</td>
<td>+ 0 0 ? 0 −</td>
<td></td>
</tr>
<tr>
<td>[SR04, SR05]</td>
<td>−</td>
<td>0 − 0 0 − 0</td>
<td></td>
</tr>
</tbody>
</table>

**MDFF** - Monotone Data Flow Framework

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Idea for a simple reuse of source components, not as advanced as baseline</td>
</tr>
<tr>
<td>S⁺</td>
<td>Idea for a simple reuse of source components, more advanced than S, but not as advanced as baseline</td>
</tr>
<tr>
<td>+</td>
<td>Potentially better than baseline</td>
</tr>
<tr>
<td>−</td>
<td>Not as good as baseline</td>
</tr>
<tr>
<td>0</td>
<td>Same like baseline</td>
</tr>
<tr>
<td>?</td>
<td>Not Decidable</td>
</tr>
</tbody>
</table>

Table 3.1: Evaluation of the related work with respect to the goal criteria.
Part III

Initial Framework
Chapter 4
Towards a Framework

We want to construct a framework that will make it easier and faster to construct analyses that work as clients to a points-to analysis. The framework should provide functionality and tools that support the development of such analyses. This includes data structures, algorithms, development support, etc. The work constructing the framework has started and parts of the functionality exists. It is possible to specify client analyses reusing common data structures and existing analysis results. The framework still needs improvements in the area of flexibility, efficiency both in time and space, reusability for more client analyses than the two presented in this chapter, i.e., aspects important to a framework.

In this chapter, we present two points-to client analyses, the escape analysis and the side-effects analysis, and show what commonalities and variation points we can identify. The commonalities include data structures and algorithms that will be part of the framework. The variation points will be left to the developer to specify using some kind of specification language. We also present conclusions we can draw from literature presented as related work in Section 3.3. This will help us in the task to identify important aspects of the client analysis framework.

First, we present the client analyses and the conclusions drawn from the related work. In the following sections, we present the commonalities and the variation points identified. Finally, a summary of what we have concluded is given.

4.1 Points-to Analysis

The points-to analysis used in the client analysis framework is based on a sparse SSA graph to represent the analyzed program, as described in Section 2.2. In this sparse SSA graph, all nodes not related to reference operations are removed, i.e., operations related to primitive data types. The SSA node types that are used in the points-to analysis are listed in Table 4.1. The Alloc node creates a new instance object of the class C. The nodes Load, Store, BagLoad and BagStore exist in two variants, one default and one static. The static variants are prefixed with an S, SLoad for example. The
4.2 Client Analyses

Bag nodes operate on array type objects. The BagLoad does not load from a specific field, rather from any of the cells of the array, since it is generally not possible to identify the different cells separately. The Entry and Exit nodes are the single entry and exit nodes for a method graph, respectively. The MCall and the PCall nodes reference the Entry nodes that are their call targets, and the Exit nodes refer back to the MCall and PCall nodes. The φ-node is a theoretical node that exists to preserve the properties of the SSA graph. It merges the values from its in-ports to its out-port.

The points-to analysis uses the simulated execution algorithm. The algorithm may be parameterized with both name schema and call context variants. The name schemata allowed are the allocation schema, the exclude schema, and the exclude strings schema. The class schema, the context-sensitive allocation schema and the object schema are not implemented, and thus not used. The class schema is partly used in the exclude schema and the exclude strings schema, but it is too imprecise to be used completely. The context-sensitive allocation schema and the object schema are two variants that may be implemented in the future. The call context variants that are implemented are the CallString, the Object, the This, the ObjectArgs and the ThisArgs sensitivities. However, the ObjectArgs and ThisArgs sensitivities are still not validated to work properly and are thus not used.

The results of the points-to analysis results that interest us in this thesis are a decorated SSA graph and a call context graph. The SSA graph nodes have the points-to analysis results on their out-ports. An in-port has the same values as the out-port it is connected to. The abstract objects that are possible targets for a method call are stored on the target in-port to the MCall or the PCall nodes, i.e., the port that contains the target objects. The call context graph consists of call contexts as nodes and the relation calls as edges; it is presented in more detail below.

4.2 Client Analyses

Escape Analysis

The escape analysis finds objects that are created within the scope of a method call under a certain context, and that are available to the application after the method execution is finished.

We classify the different ways objects may escape into three classes; direct escape, return escape and indirect escape. Direct escape occurs if an object is stored in a field of an object from the enter set, i.e., objects passed as arguments or in static object fields. Return escape is when objects escape when they are returned from the method. Indirect escape occurs if an object is stored in a field of an object that escapes.
Chapter 4. Towards a Framework

<table>
<thead>
<tr>
<th>SSA Nodes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>Method entry point</td>
</tr>
<tr>
<td>Exit</td>
<td>Method exit point</td>
</tr>
<tr>
<td>Alloc(C)</td>
<td>Creates new instance of class (C)</td>
</tr>
<tr>
<td>Load(f)</td>
<td>Loads field (a.f) into value (v)</td>
</tr>
<tr>
<td>Store(f)</td>
<td>Stores value (v) from field (a.f)</td>
</tr>
<tr>
<td>BagLoad</td>
<td>Loads objects from an array or (\text{java.util.Collection}) object</td>
</tr>
<tr>
<td>BagStore</td>
<td>Stores objects to an array or (\text{java.util.Collection}) object</td>
</tr>
<tr>
<td>(\phi)</td>
<td>Merges values on all in-ports onto one out-port</td>
</tr>
<tr>
<td>MCall(m)</td>
<td>Calls (r = a.m(v_1, \ldots, v_n)) with targets (m)</td>
</tr>
<tr>
<td>PCall(tgts)</td>
<td>Calls (r = a.m(v_1, \ldots, v_n)) with targets (tgts)</td>
</tr>
<tr>
<td>SLoad(f)</td>
<td>Loads static field (a.f) into value (v)</td>
</tr>
<tr>
<td>SStore(f)</td>
<td>Stores value (v) from static field (a.f)</td>
</tr>
<tr>
<td>SBagLoad</td>
<td>Loads objects from a static array or (\text{java.util.Collection}) object</td>
</tr>
<tr>
<td>SBagStore</td>
<td>Stores objects to a static array or (\text{java.util.Collection}) object</td>
</tr>
</tbody>
</table>

Table 4.1: SSA node types used by the points-to analysis algorithm.

The escape analysis is defined in Figure 4.1, as \(\text{Esc}(\text{ctx})\). The return escaping objects are identified through \(\text{Esc}_{\text{ret}}(\text{ctx})\); the direct escaping objects are identified through \(\text{Esc}_{\text{dir}}(\text{ctx})\); and the indirect escaping objects are identified through \(\text{Esc}_{\text{ind}}(\text{ctx})\). The definition of the indirect escaping objects contains a recursive definition, since it uses the values from \(\text{Esc}(\text{ctx})\). The indirect escaping set needs to be calculated in steps, one indirect escaping level at a time. The escape analysis algorithm would calculate \(\text{Esc}_{\text{ret}}(\text{ctx})\) and \(\text{Esc}_{\text{dir}}(\text{ctx})\) first. Now, the first level of indirect escaping objects can be found, i.e., the objects that are stored in fields of objects in either \(\text{Esc}_{\text{ret}}(\text{ctx})\) or \(\text{Esc}_{\text{dir}}(\text{ctx})\). The next level is the objects that are stored in the objects that escape indirectly at the first level. This is repeated until there are no new escaping objects found, i.e., until the minimum fix-point is found.

\(\text{StoredIn}(\text{ctx}, \text{objs})\) represents the objects that are stored in fields of the objects in \(\text{objs}\). \(\text{Source}(n)\) represents the objects that are stored by the SSA node \(n\), and \(\text{Targets}(n)\) represents the objects that are written to. \(\text{Stores}(\text{ctx})\) represents all SSA Store nodes that are within the scope of the context \(\text{ctx}\). Equation 4.5 iterates over all SSA nodes in the context \(\text{ctx} (n \in \text{Stores}(\text{ctx}))\) and takes the union from iterations. If any of the objects in \(\text{objs}\) are in \(\text{Targets}(n)\), the objects in \(\text{Source}(n)\) is used. If the intersection between \(\text{objs}\) and \(\text{Targets}(n)\) is empty, then no objects are added to the
\[ \text{Esc}(ctx) = \text{Esc}_{\text{ret}}(ctx) \cup \text{Esc}_{\text{ind}}(ctx) \cup \text{Esc}_{\text{dir}}(ctx) \]  
(4.1)

\[ \text{Esc}_{\text{ret}}(ctx) = \neg \text{Enter}(ctx) \cap \text{Returned}(ctx) \]  
(4.2)

\[ \text{Esc}_{\text{dir}}(ctx) = \neg \text{Enter}(ctx) \cap \text{StoredIn}(ctx, \text{Enter}(ctx)) \]  
(4.3)

\[ \text{Esc}_{\text{ind}}(ctx) = \neg \text{Enter}(ctx) \cap \text{StoredIn}(ctx, \text{Esc}(ctx)) \]  
(4.4)

\[
\text{StoredIn}(ctx, \text{objs}) = \bigcup_{n \in \text{Stores}(ctx)} \left\{ \begin{array}{ll}
\text{Source}(n) & \exists o \in \text{objs} \\
\emptyset & o \in \text{Targets}(n) \\
\emptyset & \text{otherwise}
\end{array} \right.
\]  
(4.5)

Figure 4.1: Context-Sensitive Escape Analysis.

result, i.e., the empty set \( \emptyset \) is added to the result set.

**Stores(ctx)** returns all SSA store nodes included in the scope reachable under the call context \( ctx \).

**Source(n)** returns all abstract nodes stored in an object field in the SSA store node \( n \).

**Targets(n)** returns all abstract objects written to in the SSA store node \( n \).

**Returned(ctx)** returns all objects returned when invoking the call context \( ctx \).

**Enter(ctx)** returns objects returned by the enter analysis in the call context \( ctx \), i.e., all objects visible from \( ctx \).

**StoredIn(ctx, objs)** returns all objects that are stored in field of the objects in \( \text{objs} \), in SSA Store nodes within the scope of the call context \( ctx \).

**Example** The example code in Figure 4.2 illustrates a number of different ways objects may escape their scope. The language in the example is highly influenced by Java; **extends** denotes inheritance. We can conclude that the object created in line 37 and stored in variable \( b \) escapes the method \( C.b() \) while the object created in line 38 and stored in variable \( b2 \) does not escape. The object referenced by \( b \) is returned from the method, while the object referenced \( b2 \) is not. The call \( b2.setB(b) \) at line 39 stores a reference from \( b \) in a field of \( b2 \), which renders no escape analysis result. However, in the method \( D.b() \) we can see that \( b \) escapes, since it is returned (direct escape). We can also see that \( b2 \) escapes, since the call \( b.setB(b2) \) stores a reference to \( b2 \) in a field of \( b \) (field escape).
Figure 4.2: Different escaping scenarios.

If we compare the method $g(B\ b)$ of the two classes $C$ and $D$, we can see that $C.g(B\ b)$ allocates a $List$ object $l$ in line 31 and stores it in a field of the formal method argument $b$ in line 33. In line 34, $b2$ is stored in the static field $A.field$ and thus escapes (direct escape) along with the list $l$ (indirect escape). The method $D.g(B\ b)$ stores a reference to the object $b2$ allocated in line 45 to the method argument $b$ via the method call $b.setB(b2)$. From $D.g(B\ b)\ b2$ escapes as do $b$ from $C.g(B\ b)$.

Let us consider the two calls to the method $call(Z\ z, B\ b)$ on lines 7 and 8 in the method $A.m1()$. The method $call(Z\ z, B\ b)$ calls the method $g(B\ b)$ on the argument $z$, which is either the object of type $C$ allocated in line 5 or the object of type $D$ allocated in line 6. We know that one object escapes $C.g(B\ b)$ and that another object escapes $D.g(B\ b)$. The escape analysis result depends on the sensitivity of the points-to result. When analyzing the method $call(Z\ z, B\ b)$, the insensitive analysis will not separate the points-to analysis results of the two calls. The escape analysis result will contain the object allocated by method $C.g(B\ b)$ in line 31 and the one allocated by $D.g(B\ b)$ in line 45. If we use a more sensitive analysis, such as $CallString$, the results between the two contexts associated with the method
call(Z z, B b) will have either of the two results depending on the call context, since the separate contexts have separate objects in their argument lists using this sensitivity.

**Summary** Regarding the information required from points-to analysis and the basic data structures: In the specification of the escape analysis in 4.1, we can see \( \text{Enter}(ctx) \), as well as \( \text{Returned}(ctx) \), \( \text{Stores}(ctx) \), \( \text{Targets}(n) \), and \( \text{Stores}(n) \). The analysis needs to know which objects are in the enter set \( \text{Enter}(ctx) \) and which objects are returned from the method called under the call context \( ctx \) (\( \text{Returned}(ctx) \)). It also needs to know which SSA Store nodes that can be found within the scope of the call to the method associated with the call context \( ctx \) (\( \text{Stores}(ctx) \)). Lastly, it needs information about which objects are stored and stored to, respectively, in these SSA Store nodes (\( \text{Targets}(n), \text{Stores}(n) \)).

**Side-Effects Analysis**

Side-effects analysis finds the abstract objects that are modified by a method call. In the literature there are two different concepts described, side-effects of a method, side-effect free methods, and pure methods. A pure method may return objects that are created inside the called method. The side-effects analysis considers the returned objects as side-effects as well, since the pre-call state is modified.

The analysis is defined in Figure 4.3. The definition of side-effects of a method call uses the definition of an impure method call. The impurity definition retrieves all objects that make a method call impure, i.e., if this set is empty, the method call is pure. This is the definition of a pure method call, as listed in figure 4.3.

The analysis algorithm can be extracted from the analysis definition in a straightforward fashion. The left sides in Figure 4.3 work as functions, returning either SSA nodes or abstract objects; \( \text{Stores}(ctx) \) returns nodes, while \( \text{Targets}(ctx), \text{WrittenTo}(ctx), \text{Returned}(ctx), \text{Enter}(ctx), \text{Impure}(ctx) \) and \( \text{Side-Effects}(ctx) \) return abstract objects. The first five are described above, while \( \text{Impure}(ctx) \) and \( \text{Side-Effects}(ctx) \) are described here:

\[ \text{WrittenTo}(ctx) \] returns all abstract objects written to in the scope of the call context \( ctx \).

\[ \text{Impure}(ctx) \] returns all objects that make a call to the call context \( ctx \) impure, according to the definition of pure, i.e., all objects in the enter set \( \text{Enter}(ctx) \) that are written to in the call context \( ctx \) (\( \text{WrittenTo}(ctx) \)).

\[ \text{Side-Effects}(ctx) \] returns all objects that make a call to the call context \( ctx \) having side-effects, i.e., all objects in \( \text{Impure}(ctx) \) plus the objects
Chapter 4. Towards a Framework

\[ Side-Effects(ctx) = Impure(ctx) \]
\[ \quad \quad \cup \]
\[ (Returned(ctx) \setminus Enter(ctx)) \]
\[ Impure(ctx) = Enter(ctx) \cap WrittenTo(ctx) \quad (4.7) \]
\[ WrittenTo(ctx) = \bigcup_{n \in Stores(ctx)} \text{Targets}(n) \quad (4.8) \]
\[ isPure(ctx) = \begin{cases} 
    \text{true} & \text{Impure(ctx) = } \emptyset \\
    \text{false} & \text{otherwise} 
\end{cases} \quad (4.9) \]
\[ isSideEffectFree(ctx) = \begin{cases} 
    \text{true} & \text{Side-Effects(ctx) = } \emptyset \\
    \text{false} & \text{otherwise} 
\end{cases} \quad (4.10) \]

Figure 4.3: Context-Sensitive Side-Effects and Purity Analysis.

returned from \( ctx \) \((Returned(ctx))\) minus the objects in the enter set of \( ctx \) \((Enter(ctx))\).

To find the objects that make a call context impure, the analysis algorithm visits all SSA store nodes in the scope of the call context and collects all abstract objects that are found on the target ports of the nodes. The abstract objects that are both in this set and the enter set constitute the impure object set. The side-effects analysis results are retrieved by adding the objects locally created and returned by the method, i.e., returned but not entered objects.

In the literature presented in Section 3.3, some additional concepts are discussed. The concepts of read-only and safe parameters are introduced by Salcianu and Rinard in [SR04, SR05]. A read-only parameter is only read from, and a safe parameter is read-only with the additional restriction that

\[ isPure(m) = \begin{cases} 
    \text{true} & \forall c \in C_m \land isPure(c) \\
    \text{false} & \text{otherwise} 
\end{cases} \quad (4.11) \]
\[ isSideEffectFree(m) = \begin{cases} 
    \text{true} & \forall c \in C_m \land isSideEffectFree(c) \\
    \text{false} & \text{otherwise} 
\end{cases} \quad (4.12) \]
\[ C_m = \text{The set of contexts associated with the method} \, m. \]

Figure 4.4: Context-Insensitive Side-Effects and Purity Analysis.
no new objects are stored in fields of objects transitively reachable from the parameter.

**Example**  The example code in Figure 4.5 illustrates a number of different ways a method can have side-effects, and how a method can be pure, but still have side-effects.

The method `B.store()` is not side-effect free, since each call to this method, independent of the calling context, will write to the static field `A.b_field` (`WrittenTo(ctx)`). All static fields are part of the visible state before the call to the method, the *enter set* (`Enter(ctx)`), and a write to such a field will result in a side-effect. If it causes a side-effect, it is also included in the *Impure(ctx)* set, since *Side-Effects(ctx)* is defined as the objects in *Impure(ctx)* plus the returned objects (`Returned(ctx)`) minus the objects in the *enter set* (`Enter(ctx)`).

To illustrate the difference between a pure and a side-effect free method, we will study the method `C.printList(List list)` that makes use of the class `ListIterator`. The class `ListIterator` iterates over a list of objects and returns them in the order they are stored in the list. When creating a new instance of this `ListIterator` class, it is necessary to supply a `List`, which should be iterated over. The method `ListIterator.hasNext()` returns true if there are any more objects in the list that has not been seen so far. The method `ListIterator.next()` returns the next object in the list. The method `C.printList(List list)` creates a new `ListIterator` object. Even though the constructor of the class `ListIterator` writes to its object field `list`, the method `C.printList(List list)` is still pure, since the `ListIterator` object `i` is created in `C.printList(List list)`. However, the method still has side-effects, not because of the mentioned write operation, but rather since it returns an object. This is a side-effect, according to the specification of side-effects in Figure 4.3.

**Summary**  Regarding the information required from points-to analysis and the basic data structures: In the specification of the side-effect and purity analysis in Figure 4.3, we can see that `Enter(ctx)` occurs as well as `Returned(ctx)`, `Stores(ctx)`, and `Targets(n)`. The analysis needs to know which objects are in the enter set (`Enter(ctx)`) and which objects are returned from the method called under the call context `ctx` (`Returned(ctx)`). It also needs to know which SSA Store nodes can be found within the scope of the call to the method associated with the call context `ctx` (`Stores(ctx)`). Lastly, it needs information about which objects are stored to in these SSA Store nodes (`Targets(n)`).
Chapter 4. Towards a Framework

```java
class A {
    static B b_field;
    ...
}
class ListIterator {
    List list;
    Integer index = 0;
    ListIterator(List l) {
        list = l;
    }
    boolean hasNext() {
        Integer l = list.length - 1;
        return l > index;
    }
    ListItem next() {
        index = index + 1;
        return list.get(index - 1);
    }
}
class B {
    store() {
        A.b_field = new B();
    }
}
class C {
    Integer printList(List list) {
        ListIterator i = new ListIterator(list);
        Integer sum = 0;
        while (i.hasNext()) {
            sum += (Integer)i.next();
        }
        return sum;
    }
}
```

Figure 4.5: Different side-effects and purity scenarios.

### 4.3 Commonalities

The conclusions that can be drawn from the two summary paragraphs in the two client analyses sections in Section 4.2 are that both analysis are dependent on the information represented by $\text{Enter}(ctx)$, $\text{Returned}(ctx)$, $\text{Stores}(ctx)$, and $\text{Targets}(n)$. The escape analysis also depends on the information from $\text{Source}(n)$. All of these are very basic and would very likely be a part of most analyses based on points-to information.

Here, we present the basic analyses that are needed by both the escape and the side-effects analyses, i.e., read/write analysis ($\text{Stores}(ctx)$, $\text{Source}(n)$, and $\text{Targets}(ctx)$), return analysis ($\text{Returned}(ctx)$), and the enter analysis ($\text{Enter}(ctx)$). Some of these basic analyses rely in their turn on the knowledge about the scope of the analyzed call context $ctx$, i.e., which methods that may be reached from the call to $ctx$. This information is calculated by the scope analysis, which is also presented below. The presentation of these analyses starts with a presentation of the common data structures that are used by the basic analyses.

### Data Structures

**BitVector** To represent sets we use a bit vector implementation. A universe set of objects is associated with instances of the BitVector class. Different types of BitVector sets are defined by the universe set they are associated with.
Each object of this universe is assigned an index in the bit vector. An array of `long` is created with a size such that the number of bits in this vector exceeds the number of objects in the universe. If a bit is set to 1 in this vector, the object with the corresponding index is said to be included in the set.

This data structure enables us to perform logical bit operations on the sets, which can be performed by hardware instead of in software and, thus, are very efficient. The `add` and `remove` operations are performed in constant time, since they are performed by flipping a bit in the bit vector. More advanced operations such as union, intersection, and the complement operations are performed in linear time against the number of `longs` the bit vector consists of, when logical bit operations are used. Union is implemented using the logical `or` operation, intersection using the logical `and` operation, and complement using the logical `xor` operation. The union and the intersection operations are binary operations, and so are `and` and `or`. However, the complement operation is a unary operation while `xor` is a binary operation. The first argument is the bit vector and the second argument is the universe. The result is that all ‘1’-entries in the bit vector is transformed to ‘0’-entries in the result, and vice versa.

There are some drawbacks with this way of implementing a set. Each set object needs a bit vector with a size large enough to contain all objects in the associated universe. This is space inefficient when the cardinality of the set is small, compared to the universe. A hybrid version could be used to reduce this problem, where the set’s cardinality decides if the bit vector version or a simple reference array should be used. At first, the objects are stored in a reference array, and when the set’s cardinality exceeds a predefined threshold, this representation is changed to the bit vector implementation. This hybrid solution needs two versions of operations, one for the bit vector version and one for the reference array version. It should be transparent to perform operations on set objects using either of the two versions.

**Call Context Graph** The call context graph is a directed graph $G = (C, E)$, where $C$ is the set of call contexts and $E$ is the set of edges connecting call context nodes $(c_i, c_j)$ such that $c_i$ contains a call to $c_j$. There may be cycles in this graph. A cycle occurs if there is a recursive call in the program.

The context call graph has a starting node, which is the context that is called by the environment that executes the program. In Java this is the call to the `main()` method done by the Java Virtual Machine (JVM). The call context graph is constructed during the points-to analysis. When a new call context is reached during the points-to analysis, a node with this call context associated is added to the call context graph. For each call context that is reached during points-to analysis, an edge between this and the call context that calls it is added to the call context graph. In the case of a polymorphic
call, this results in an edge for each of the possible target call contexts.

Figure 4.6 shows an example of a call context graph. The source code that is shown in the figure is an extension of the source code in the escaping scenarios example shown in Figure 4.2. The nodes in the call context graph shows the result after an object-sensitive points-to analysis, i.e., contexts are defined by an object and the method called on this object. The objects are referred to by the source code line that created them. The main method is not called on any object, since it is static. In Java, the constructors of a class is called when a new object is instantiated. The constructor method is named init in this example.

Analyses

Scope analysis was not explicitly required by the two client analyses we have looked at so far. However, the basic analyses that they require do rely on which nodes are contained in a scope of a call. This is information that scope analysis provides. Therefore, we chose to present this analysis first, while the rest of the basic analyses follow below.

Scope Analysis The scope analysis calculates the scope of each node (call context) in the call context graph. The result is a mapping $c_i \rightarrow \{c | c \in C \land c \in \text{Scope}(c_i)\}$, where $C$ is the set of call contexts in the analyzed program.
4.3. Commonalities

and $\text{Scope}(c_i)$ denotes the scope of the call context $c_i$. The associated set is the set of call contexts that may be called transitively when the call context $c_i$ is called.

Scope analysis is solved as a data flow problem. Each call context node is associated with a set of call context nodes (called scope set), which are reachable from this node. Each scope set is initialized with the node it is associated with, i.e., the mapping $c_i \rightarrow \{c_i\}$. The algorithm is a backwards dataflow problem, i.e., the data propagation goes against the direction of the edges in the call context graph. The propagation starts from the call context leaf nodes and continues until a fix-point is reached. When two sets are merged, the set union operation is used. The fix-point iteration is needed if there are cycles in the graph, i.e., if there are any recursive calls in the program.

An optimization to this algorithm that may save time and space is to merge the nodes that are on a cycle into a virtual node. The nodes that are on a cycle will have the same scope; from one node it is possible to reach any of the other nodes in the cycle. We can calculate the cycles as the Strongest Connected Components (SCC) for the graph. Each SCC is a virtual node that represents each cycle in the graph. This enables the algorithm to compute the fix-point algorithm over fewer nodes and the context information needs only be associated with the virtual nodes, instead of separate representations for each of the nodes. There is a known linear solution that calculates the SCC for a graph [Tar72]. The effect this optimization has on time and space is related to the number of nodes included in a cycle in the call context graph of the analyzed program.

**Scope analysis results**

\[
\begin{align*}
(56, \text{call}) & \rightarrow \{(56, \text{call}), (5, g) (4, \text{setL}) (31, \text{List}) \\
& (32, \text{init}) (6, g) (4, \text{setB}) (45, \text{init})\} \\
(5, g) & \rightarrow \{(5, g) (4, \text{setL}) (31, \text{Init}) (32, \text{init})\} \\
(6, g) & \rightarrow \{(6, g) (4, \text{setB}) (45, \text{init})\} \\
\end{align*}
\]

...  

Figure 4.7: Scope analysis example.

An example of scope analysis results is shown in Figure 4.7. The scope analysis has been performed on the call context graph illustrated in Figure 4.6. The scope analysis results is a mapping from a context ctx to a set of contexts that are reachable in the call context graph, starting in call.
context node $ctx$. The mapping node is included in the set as well. We can see in Figure 4.6 that from node $(6, g)$ it is possible to reach the nodes $(4, setB), (45, init)$.

**Return Analysis** The return analysis provides results to $Returned(ctx)$. The return analysis is rather straight-forward; it finds the objects that are returned from a call to a method $m$ under a certain context $ctx$. This analysis is very basic, but also very useful in many possible client analyses.

The result of the return analysis is a mapping between a call context $ctx$ to a set of abstract objects $objs$ that are returned by $ctx$, i.e., $c \rightarrow \{o|returnedBy(c)\}$. The SSA method graph has one exit node ($Exit$), which represents all return statements in the method. This node stores the abstract objects that are returned from the method under the given call context, and they are stored on the SSA $Exit$ node’s out-port. The analysis visits all $Exit$ nodes and creates the previous mentioned mapping between call context and abstract objects.

**Read/Write Analysis** The read/write analysis provides information to $Stores(ctx)$, $Source(n)$, and $Targets(n)$. The result of the read/write analysis is represented in a read set and a write set. The read set contains all load nodes, i.e., $Load$, $SLoad$, $BagLoad$ and $SBagLoad$ nodes, while the write set contains all store nodes, i.e., $Store$, $SStore$, $BagStore$ and $SBagStore$ nodes. The two sets are represented using BitVector objects. The advantage of creating these sets is that the calculation which read or write nodes are within the scope of a call to a method $m$ under a certain call context $ctx$ is reduced to calculating the intersection between the scope node set and the read or write sets. Therefore, it is not necessary to traverse the method graphs to find the read or write nodes.

**Enter Analysis** The enter analysis provides information to $Enter(ctx)$ by collecting the objects that are visible to a method call in a given context and that are used in the scope of the method call. The analysis is needed to work around a problem that arises from our object allocation schema. An abstract object is identified with the syntactic program point where it is allocated, thus it is not context-sensitive. If we want to separate versions of an object created in different call contexts, we should perform the enter analysis. If the method $m$ creates and returns the abstract object $o$, all methods that call $m$ will be considered to have allocated the object $o$. If the methods $n_1$ and $n_2$ both call $m$ and $n_1$ calls $n_2$ with $o$ as an argument to $n_2$ it is not conservative to say that all occurrences of $o$ are the object that was created in the call to $m$ from $n_2$; it could also be the version created when $n_1$ called $m$. An illustration of this scenario is shown in Figure 4.8. The object created in line 3 ends up in the variable $a_1$ and $a$ in the method $n_2$. The statement in
4.4 Variation points

The previous section presented the commonalities we identified between the escape analysis and the side-effects analysis. Here, we present the variation points that we identified, as well. The variation is on two different levels, on the level of the points-to analysis and in the client analyses.

The speed and precision properties of the points-to analysis may be selected by choosing different context-sensitivities and name schemas. However, how this is specified or performed is outside the topic of this thesis. For an in-depth description of the used points-to analysis in [LL07].

The specifications of the two client analyses are different, i.e., the way they make use of the results of the basic analyses. This is what decides what the analysis results will be. This specification should be implemented using some kind of specification language. It could be a script language or a constraint-based specification, like the client analyses specifications that are given in Figures 4.1 and 4.3. The focus should be on objects from the

Figure 4.8: Enter analysis example.
points-to analysis result, since these are an integral part of the client analyses calculation that produces its analysis result.

The client analysis framework should provide means to transform the client analysis specification into a running client analysis. This could be done in different ways, for instance by generating a compiled Java implementation of the analysis or providing an interpreting virtual machine for the specification language. Either of these options will make sure that the client analysis specification is realized in executable code.

4.5 Summary

We have presented two client analyses, the escape analysis and the side-effects analysis. We have identified the commonalities and the variation points between these two client analyses. The identified commonalities were both data structures and basic analyses. The data structures include a BitVector class representing sets and a call context graph. The basic analyses were the read/write analysis, the scope analysis, return analysis and the enter analysis. The identified variation points can be summarized in the need for a specification language that is able to express how the results of the basic analyses should be combined in order to produce the intended analysis result.
Chapter 5

Experiments

This chapter presents a number of experiments to verify that we have met the goal criteria posed in Section 1.3. The experiment chapter presents two experiments, one evaluating the development speed and one evaluating the analysis precision and speed. The first experiment shows that the client analysis framework provides a lot of functionality and support for client analysis developers and that the effort a developer has to invest when creating a client analysis is small. This supports the hypothesis that the client analysis framework enables high development speed, compared to develop without it. The second experiment shows that it is possible to select the tradeoff between speed and precision for the analyses implemented in the client analysis framework so far. It also shows that the precision in the points-to analysis propagates to the client analyses. This motivates the different precisions in the points-to analysis, since the measurable results from the client analyses depend on them.

5.1 Development Speed Experiment

The client analysis framework provides the client analysis developer with a points-to analysis, basic client analyses, appropriate data structures and a way to specify how to combine the basic analyses results into a client analysis. The client analysis developer has to write the client analysis specification in order to produce a new client analysis. The client analysis framework ensures that the results of the basic analyses are computed as needed and that an implementation of the client analysis is generated from the client analysis specification. The knowledge the client analysis developer needs to have is restricted to a general knowledge about the domain and the client analysis specification language, rather than an in-depth knowledge about how to construct a points-to analysis, or how to manually produce the analysis results that the basic analyses produce. The client analysis framework will help to minimize the time that is spent on activities such as debugging, testing and bug-fixing of the parts of the analysis that is not part of the main focus of the client analysis that is developed.

A deeper investigation of the client analysis framework, involving devel-
Chapter 5. Experiments

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
<th>LOS</th>
<th>Classes</th>
<th>Abstract</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escape</td>
<td>143</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Side-Effects</td>
<td>83</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Basic</td>
<td>2602</td>
<td>-</td>
<td>21</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Metric</td>
<td>975</td>
<td>-</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>General</td>
<td>1362</td>
<td>-</td>
<td>10</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Points-to</td>
<td>39180</td>
<td>-</td>
<td>204</td>
<td>25</td>
<td>11</td>
</tr>
</tbody>
</table>

**Sum (excl. Points-to)** 4939

**Sum (incl. Points-to)** 44119

LOS = Lines Of Specification

LOC = Lines Of Code

Table 5.1: Source code measures of the framework.

opers, is not possible, since it is not mature enough for this. However, we may compare the number of lines of source code between the framework and the two client analyses. Table 5.1 shows the number of lines of code (LOC) for four components in the client analysis framework, basic analyses, metrics, general and points-to analysis. The LOC measurement includes source code, comments, and white space, excluding import statements. It also shows the number of lines of specification (LOS) for the two client analyses. The LOS measurement is taken from the definitions of the client analyses in Section 4.2. The table also shows how many concrete classes, abstract classes and interfaces the components consist of. The basic component includes the basic analyses and its data structures, and the metric component contains functionality for storing and calculating measures on the results of client analyses. The general component consists of general purpose classes for the client analysis framework and the points-to component contains all functionality required to perform the points-to analysis. Third-party libraries, such as a graph library and a binary code reader library, are not included in these measurements. Some of the classes in the four components provide functionality that can be used via an Application Programming Interface (API), and some may be generated from the client analysis specification.

The side-effects analysis consists of only 83 lines of code, which is 1.7% of the lines of code for the client analysis framework, excluding the points-to analysis, and the escape analysis consists of 143 lines of code, which is 2.9%. This is what is needed for a client analysis developer to specify, in the client analysis framework. For developers not having a points-to analysis available, the benefit from the client analysis framework is even larger. With a client analysis specification language similar to the client analysis definitions presented in Section 4.2, it would be possible for a developer to specify the escape analysis and the side-effects in 6 and 3 lines, respectively.
5.2. Precision and Speed Experiment Setup

Benchmarks

We have used a benchmark containing 10 different programs ranging in size from 262 to 857 classes. All programs are presented in Table 5.2. It shows the number of used classes (Classes), the number of reachable methods (Methods), and the number of abstract objects (Objects). The programs in the upper half of the table are taken from well-known test suites [Ash, Spe, Dac06]. They are a bit “older” and are always analyzed using version 1.3.1 of the Java standard library. In the lower half, we have our own set of “newer” test programs. They are all publicly available and are always analyzed using version 1.4.2 of the Java standard library. The program obfusc0.73 is a source code obfuscator that comes with the Java source code transformation library Recoder v0.73.

Execution Environment

All experimental data presented in this thesis is the median value of three runs on the same computer (Dell PowerEdge 1850, 6GB RAM, Dual Intel Xeon 3.2GHz under Linux x86-64, kernel 2.6.12). The application is single

<table>
<thead>
<tr>
<th>Program</th>
<th>Classes</th>
<th>Methods</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>262</td>
<td>1335</td>
<td>787</td>
</tr>
<tr>
<td>javac1.3</td>
<td>340</td>
<td>2580</td>
<td>1017</td>
</tr>
<tr>
<td>javadoc</td>
<td>451</td>
<td>2359</td>
<td>1210</td>
</tr>
<tr>
<td>sablecc-j</td>
<td>692</td>
<td>3825</td>
<td>1715</td>
</tr>
<tr>
<td>soot-c</td>
<td>838</td>
<td>3604</td>
<td>1871</td>
</tr>
<tr>
<td>emma2.0</td>
<td>857</td>
<td>4936</td>
<td>2787</td>
</tr>
<tr>
<td>javacc3.2</td>
<td>311</td>
<td>2146</td>
<td>1047</td>
</tr>
<tr>
<td>jess4.5</td>
<td>364</td>
<td>1835</td>
<td>1108</td>
</tr>
<tr>
<td>obfusc0.73</td>
<td>728</td>
<td>4517</td>
<td>1472</td>
</tr>
<tr>
<td>pmd3.2</td>
<td>556</td>
<td>3338</td>
<td>1418</td>
</tr>
</tbody>
</table>

Table 5.2: Benchmark information

With the line of arguments presented above, we claim that the client analysis framework provides means to get fast development for client analyses, compared to not using the framework, or reusing of components providing equivalent functionality.
threaded and uses only one of the four available virtual processors. The Java Virtual Machine version 1.5.0_04-b05 was used.

Analysis Metrics

The analysis metrics used in the experiments are analysis time measurements and precision measurements for the client analyses. The time measurements are separated into three measurements, one for the points-to analysis, one for the basic analyses and one for the client analysis. The client measurements are different for the two client analyses, escape analysis and side-effects analysis. The presentation and definition of these measurements follow below.

The context-sensitive analyses ensure better precision in the analyses results. They can also be used when the interest is focused on a certain call to a method $m$ in a program. However, the context-sensitive result can also be used when analyzing a certain method, not only for a specific context of a method. The analysis results of all contexts associated with a method are merged into a method result. This merged analysis result corresponds to the measures $Escaping_U$ in escape analysis and $SideAffected_U$ in side-effects analysis, both defined later in this chapter.

In general to be able to compare the results using sensitive points-to analysis results to the insensitive version, we need to transform results per context to results per method. The insensitive analysis do not separate calls to the same method. We calculate the average result over the contexts having the same target method, and report the result per method.

Points-to Parameters The trade-off between precision and speed of the points-to analysis may be modified by selecting the used name schema and the context-sensitivity. The more precise name schemas and context-sensitivities may affect the analysis time negatively and the analysis could take longer. The name schema decides if all allocation points should result in an abstract object, e.g., allocation schema, or if objects of certain classes should be mapped to the same abstract object, e.g., exclude schema.

The schema used in the experiments is the allocation schema and the used context-sensitivities are insensitivity, This-sensitivity, Object-sensitivity and CallString-sensitivity. The implementations of ThisArgs-sensitivity and ObjectArgs-sensitivity have not been validated. Therefore, we chose not to include them in the experiments in this thesis.

Analysis Times The analysis time is measured to compare how the parameterizations of the points-to analysis affect the points-to analysis time, the basic analysis time and the client analysis time. The total analysis time, which includes the time for the points-to analysis, the basic analyses and the client analysis to be completed, gives a measure of how long time the user
of the client analysis would have to wait for the whole process to be completed. If the points-to analysis and the basic analyses are performed and their results are stored, different client analyses could reuse the results. In this case it is interesting to know both the time for the basic analyses and the individual client analysis times.

The time measurements for the analysis does not include preparing tasks such as parsing programming code or creating method graphs or loading stored method graphs from secondary storage. It does not include times for preparing or storing analysis results to memory or secondary storage either. It is only the actual analysis time.

The times are influenced by the name schema and the sensitivity that the points-to analysis is parameterized to use. The time measurements are thus separated on these two parameters. However, the experiment in this thesis is only concerned with different context-sensitivities and does not variate the name schema. The three time measurements that are used are described below.

\[
\text{Time}_{\text{points-to}}(ns, sens) \quad \text{is the time it takes to complete the points-to analysis using the name schema } ns \text{ and the sensitivity } sens.
\]

\[
\text{Time}_{\text{basic}}(ns, sens) \quad \text{is the time it takes to complete the basic analyses using the name schema } ns \text{ and the sensitivity } sens.
\]

\[
\text{Time}_{\text{escape}}(ns, sens) \quad \text{is the time it takes to complete the escape analysis using the name schema } ns \text{ and the sensitivity } sens.
\]

\[
\text{Time}_{\text{side-effects}}(ns, sens) \quad \text{is the time it takes to complete the side-effects analysis using the name schema } ns \text{ and the sensitivity } sens.
\]

**Points-to Analysis** The precision of the points-to analysis results are measured using an abstract metric on a data structure that is an auxiliary result of the analysis called Object Member Graph (OMG). The OMG is a graph with object methods \((o, m)\) and object fields \((o, f)\) as nodes, and three edge types: object method call, object field write and object field read. The edge count from this graph is a measure of how precise the analysis result is.

**Escape Analysis** The measures of the escape analysis results presented in the literature presented in Chapter 3 are all except one indirect measures. The only direct measure counts escaped objects \([WL04]\). Stack allocated objects, removed synchronization points and measured speedup enabled by these optimizations require additional analyses to be constructed and optimizations implemented. This work is out of the scope of this thesis and is instead considered future work. The measure we apply is the count of escaped objects. This measure helps to compare different context-sensitivities. It is
not a measure of an actual enabled improvement or optimization. However, since the framework is not directed towards a specific analysis, it makes more sense to use these abstract measures instead of measuring as in the literature.

We have chosen three measurements to present the escape analysis results; *escaping a method* ($Escaping_M$), *escaping a scope* ($Escaping_S$), and *unique objects escaping* ($Escaping_U$). The metrics are formally defined in Figure 5.1.

$Escaping_S$ counts all objects that escape a certain context scope. The objects that may escape the context scope are all objects that are allocated within the scope. The method average is used. A lower number of escaped objects is retrieved by a more precise analysis.

$Escaping_U$ counts the unique objects that escape from a method. All sets of objects escaping a context are put in the same set and the cardinality of this set is reported as the result for each method. A lower number of escaped objects is retrieved by a more precise analysis.

$Escaping_M$ counts objects that escape from a certain method. An object may only escape the method where it is allocated. The method average is used. A lower number of escaped objects is retrieved by a more precise analysis.

The metrics $Escaping_U$ and $Escaping_S$ are two variants of the same metric; $Escaping_U$ counts measure the size of the union of the escaping objects while $Escaping_S$ counts objects. $Escaping_M$ counts objects in the same way as $Escaping_S$, except only objects created in the method are counted. A metric that measure the size of the union of escaped objects from the method would be possible as well, but we do not present that metric in this thesis, since the difference compared to $Escaping_M$ has been shown to be very small in all analyzed programs. The maximum difference is 1.2\%, while 27 of the 40 measure points ($Programs \times Sensitivities$) show less difference than 0.05\%.

**Side-effects Analysis**  The identification of side-effect free and pure methods, respectively, is essential when it comes to for instance program comprehension, program documentation, compiler optimizations, and program model checking. If a method is identified as side-effect free/pure, regardless of target object and other arguments, the method can be documented as side-effect free/pure. The method is then known not to inflict any changes to the visible state, i.e., objects created and stored in locals, formals and static fields, and can be treated differently from methods that do not have this property.

Some of the side-effects analysis measures presented in the literature discussed in Chapter 3 are direct measures and some are indirect measures. The
5.2. Precision and Speed Experiment Setup

\begin{align*}
\text{Escaping}_S &= \sum_{m \in M} \left( \sum_{c \in C_m} \frac{|\text{Escaping}(c)|}{|C_m|} \right) / |M| \\
\text{Escaping}_U &= \sum_{m \in M} \left( \bigcup_{c \in C_m} \text{Escaping}(c) \right) / |M| \\
\text{Escaping}_M &= \sum_{m \in M} \left( \sum_{c \in C_m} \frac{|\text{Escaping}(c) \cap \text{Allocated}(m)|}{|C_m|} \right) / |M|
\end{align*}

\[ C_m = \text{The set of contexts associated with the method } m \]
\[ M = \text{The set of analyzed methods.} \]

Figure 5.1: Definitions of metrics related to escape analysis.

measure of the results of the loop invariant optimizations is an indirect measure, which requires another analysis and optimization to be implemented. Counting the number of objects that statements modify and how many methods are side-effect free, is a more direct measure of the side-effects analysis results. Since performing optimizations based on the side-effects analysis results is out of scope for this thesis, we only consider the direct measures.

We have chosen to present four measurements; side-effect free methods (\text{SideEffectFree}_U), pure methods (\text{Pure}_U), affected objects cause by a specific method call (\text{SideAffected}_S) and affected objects caused by a method call (\text{SideAffected}_U). The metrics are described below and formally defined in Figure 5.2.

\text{SideEffectFree}_U \text{ counts all methods that are side-effect free. All call contexts associated with the method need to be side-effect free for the method to be side-effect free. The number of side-effect free methods is reported. A higher number of identified side-effect free methods is retrieved by a more precise analysis.}

\text{Pure}_U \text{ counts all methods that are pure. All call contexts associated with the method need to be pure for the method to be pure. The number of pure methods is reported. A higher number of identified side-effect}
Chapter 5. Experiments

\[
\text{SideEffectFree}_U = |\{m | m \in M \land \text{isSideEffectFree}(m)\}|
\]
\[
\text{Pure}_U = |\{m | m \in M \land \text{isPure}(m)\}|
\]
\[
\text{SideAffected}_S = \frac{\sum_{m \in M} \left( \sum_{c \in C_m} |\text{Side-Effects}(c)| \right)}{|M|}
\]
\[
\text{SideAffected}_U = \frac{\sum_{m \in M} \left| \bigcup_{c \in C_m} \text{Side-Effects}(c) \right|}{|M|}
\]

\(C_m = \) The set of contexts associated with the method \(m\)
\(M = \) The set of analyzed methods.

Figure 5.2: Definitions of metrics related to side-effects and purity analysis.

free methods is retrieved by a more precise analysis.

\(\text{SideAffected}_S\) counts all objects that are affected in a certain context scope. The objects that may be affected within the context scope are all objects that are included in the enter set of the call context. The method average is used. A lower number of affected objects is retrieved by a more precise analysis.

\(\text{SideAffected}_U\) counts the unique objects that are affected by a method call. All sets of objects that are affected in a context are put in the same set and the cardinality of this set is reported as the result for each method. A lower number of affected objects is retrieved by a more precise analysis.

5.3 Precision and Speed Experiment Results

The results of the analysis precision and speed experiment is presented in five tables. Each table shows a number of metrics defined above. Each metric shows four values, one for each points-to analysis sensitivity in the experiment. The insensitive measure is a value, while the other sensitivities show its relation to Insensitive. This relation is expressed in how many percent the value for the other sensitivity is in relation to the insensitive value. This relation is defined as \(\text{sens/insens}\), and written in percent. The value for
a metric using the CallString sensitivity is retrieved through multiplication between the insensitive value and the fraction represented by the percent value. It is not always possible to retrieve the exact number, since the values do not have enough precision, but the relation between the sensitivities is more important than the exact numbers in this experiment.

At the bottom of each table the mean and the median are given for the different context-sensitivities, except for the insensitivity. These values do not make sense for the insensitive case, since we do not want to compare the analyzed programs with each other on this level. However, we are interested in the differences between the different context-sensitivities, relative the insensitive results. Therefore, it is interesting to be able to compare the mean and median values for the three context-sensitivities. There are a number of outliers in the tables, which motives the use of both the mean and the median value. The median value is usually more representative for the values in the specific column, than the mean.

It is worth reflecting on what the different numbers mean, if high values are good or if low numbers are good. This is of course dependent on which metric is considered. The lower the number, the better the result is generally the case, except for the two metrics $\text{SideEffectFree}_U$ and $\text{Pure}_U$. With these two metrics, it is better to have high values, i.e., it is better to identify more methods that are side-effect free and pure, respectively.

## 5.4 Discussion

The experiment results consists of points-to analysis results, escape analysis results, side-effects analysis results and analysis times for the points-to analysis, the basic analyses, and the two client analyses. These results are discussed in this section.

### Points-to Analysis

The precision of the points-to analysis results is shown in Table 5.3. The patterns found here are the patterns that are compared to in the client analyses discussion. The Insensitive is less precise than the others, while is it not possible to theoretically order the other three after how precise they are. However, it is evident from Table 5.3 that ObjectSensitive is more precise than both ThisSensitive and CallString and that ThisSensitive is more precise than CallString in our experiment.

### Escape Analysis

The result of the escape analysis is shown in Table 5.4. The results of the This-sensitive are quite close to the Object-sensitive. There are only very
small differences between these two, even though Object-sensitive is still
the most precise. However, the This-sensitivity is faster than the Object-
sensitivity in all experiments, as can be seen in Table 5.7 and 5.8.

The CallString-sensitivity is closer to the Insensitive analysis in its results.
It gives 9.6% and 8.5% improvement on average for \( Escaping_S \) and \( Escaping_U \),
respectively. However, it is substantially slower than the insensitive analysis,
as can be seen in Table 5.7 and 5.8.

The two metrics \( Escaping_S \) and \( Escaping_U \) are very similar, with \( Escaping_S \)
giving slightly better results. This comes from the definition of the metrics.
\( Escaping_U \) counts all unique objects that escape from a method, over all its
contexts, while \( Escaping_S \) takes the average number of objects that escape
from a method, over all its contexts. Some of the results for these metrics
using This- and Object-sensitivity are very low, such as javacc, sablecc-j and
obfuscator. These programs probably contain a few methods that are called
by many other methods. If these frequently called methods let objects they
create escape to static object-fields, these objects will escape even if other
methods call them. However, this is not the case for the \( Escaping_M \) metric,
since it only counts objects the method itself created.

The metric \( Escaping_M \) gives a more moderate improvement than the two
other escape analysis metrics. In the callstring-sensitivity it only gives a 2.0% improvement on average and 15.4% and 14.3% for the this-sensitivity and the object-sensitivity, respectively. The improvement means that less objects escape from the method that created them. This could be used in a successor step to compiler optimizations such as stack allocation and synchronization removal that may lead to a higher percentage of speedup.

<table>
<thead>
<tr>
<th>Program</th>
<th>Insensitive (#)</th>
<th>CallString (%)</th>
<th>This (%)</th>
<th>Object (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>223,499</td>
<td>21.2</td>
<td>19.9</td>
<td>7.7</td>
</tr>
<tr>
<td>emma</td>
<td>1,376,194</td>
<td>33.2</td>
<td>27.3</td>
<td>8.2</td>
</tr>
<tr>
<td>javac1.3</td>
<td>401,005</td>
<td>62.1</td>
<td>48.1</td>
<td>33.2</td>
</tr>
<tr>
<td>javacc</td>
<td>524,274</td>
<td>7.9</td>
<td>7.7</td>
<td>4.5</td>
</tr>
<tr>
<td>javadoc</td>
<td>347,212</td>
<td>47.8</td>
<td>37.6</td>
<td>33.4</td>
</tr>
<tr>
<td>jess</td>
<td>396,432</td>
<td>46.5</td>
<td>45.7</td>
<td>17.2</td>
</tr>
<tr>
<td>obfuscator</td>
<td>236,116</td>
<td>50.4</td>
<td>46.4</td>
<td>29.5</td>
</tr>
<tr>
<td>pmd</td>
<td>272,133</td>
<td>49.8</td>
<td>48.7</td>
<td>37.8</td>
</tr>
<tr>
<td>sablecc-j</td>
<td>255,443</td>
<td>14.1</td>
<td>13.5</td>
<td>10.9</td>
</tr>
<tr>
<td>soot-c</td>
<td>1,110,540</td>
<td>35.8</td>
<td>29.1</td>
<td>22.9</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td>36.9</td>
<td>32.4</td>
<td>20.6</td>
</tr>
<tr>
<td>Median</td>
<td></td>
<td>41.2</td>
<td>33.4</td>
<td>20.1</td>
</tr>
</tbody>
</table>

Table 5.3: Experiment results for the points-to analysis.
5.4. Discussion

Escaping

Program | $\text{Insens} (\# \cdot 10^3)$ | $\text{This} (%) $ | $\text{Object} (%) $ | $\text{CallString} (%) $ | $\text{Insens} (\# \cdot 10^3)$ | $\text{This} (%) $ | $\text{Object} (%) $ | $\text{CallString} (%) $ | $\text{Insens} (\# \cdot 10^3)$ | $\text{This} (%) $ | $\text{Object} (%) $ | $\text{CallString} (%) $
---|---|---|---|---|---|---|---|---|---|---|---|---|---
antlr | 25 | 99.5 | 52.2 | 52.1 | 25 | 99.5 | 52.2 | 52.2 | 0.58 | 99.5 | 89.7 | 89.7 |
emma | 2,024 | 97.0 | 32.0 | 31.8 | 2,024 | 97.2 | 32.0 | 31.9 | 1.75 | 98.6 | 74.6 | 74.6 |
javac1.3 | 1,516 | 90.6 | 91.3 | 90.5 | 1,516 | 92.6 | 91.6 | 91.5 | 0.69 | 88.0 | 84.8 | 84.8 |
javacc | 100 | 97.4 | 9.3 | 9.3 | 100 | 97.4 | 9.3 | 9.3 | 0.80 | 99.9 | 88.0 | 88.0 |
javadoc | 484 | 98.6 | 89.5 | 88.9 | 484 | 98.7 | 89.7 | 89.6 | 1.03 | 96.9 | 76.8 | 76.8 |
jess | 120 | 93.5 | 75.9 | 75.2 | 120 | 93.6 | 75.9 | 75.9 | 0.72 | 99.6 | 76.4 | 76.4 |
obfusc. | 564 | 96.7 | 21.4 | 21.4 | 564 | 96.8 | 21.4 | 21.4 | 1.49 | 99.7 | 78.9 | 78.9 |
pmd | 155 | 98.0 | 31.3 | 31.3 | 155 | 98.0 | 31.3 | 31.3 | 1.11 | 98.9 | 87.0 | 87.0 |
sablecc-j | 121 | 98.2 | 10.2 | 10.1 | 121 | 98.8 | 10.2 | 10.2 | 0.72 | 99.7 | 93.4 | 93.4 |
soot-c | 215 | 34.0 | 89.8 | 89.8 | 215 | 42.0 | 89.9 | 89.8 | 1.85 | 99.7 | 96.7 | 96.7 |
**Mean** | **90.4** | **50.3** | **50.0** | **91.5** | **50.4** | **50.3** | **99.8** | **84.6** | **84.6** |
**Median** | **97.2** | **42.1** | **42.0** | **97.3** | **42.1** | **42.0** | **99.5** | **85.9** | **85.9** |

Table 5.4: Experiment results for the escape analysis.

Side-Effect Analysis

The result of the side-effects analysis is shown in Tables 5.5 and 5.6. The measures are split into two tables only for better presentation. The results for the This-sensitive analysis are very close to the Object-sensitive results, as it was in the escape analysis. The Object-sensitive is the most precise, for these metrics as well. The This-sensitive side-effects analysis is much faster than the Object-sensitive variant, with larger differences than in the escape analysis, as can be seen in Table 5.8. The analysis of the program *emma* spends much more time performing the side-effects analysis under the Object-sensitivity, compared to both the Insensitive and the this-sensitive analyses.

The CallString-sensitivity does not provide much more precision compared to the side-effects metrics. In the $\text{SideEffectFree}_U$ metric the precision is 1.7% on average and for the $\text{Pure}_U$ metric it is 1.6% on average. The largest improvement is for the program *javac1.3*, with 8.1% for $\text{SideEffectFree}_U$ and 8.0% for $\text{Pure}_U$.

The metrics $\text{SideEffectFree}_U$ and $\text{Pure}_U$ show that we identify 42.1% more side-effect free methods, on average, with the This-sensitive analysis and 41.9% with the Object-sensitive analysis. The $\text{SideAffected}_S$ and the $\text{SideAffected}_U$ metrics gives another perspective on the differences between the sensitivities. The improvement these show are 90.5% for the This-
Chapter 5. Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>SideEffectFree&lt;sub&gt;U&lt;/sub&gt;</th>
<th>Pure&lt;sub&gt;U&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insens (#)</td>
<td>CallString (%)</td>
</tr>
<tr>
<td>antlr</td>
<td>132</td>
<td>100.0</td>
</tr>
<tr>
<td>emma</td>
<td>351</td>
<td>101.4</td>
</tr>
<tr>
<td>javac1.3</td>
<td>248</td>
<td>108.1</td>
</tr>
<tr>
<td>javacc</td>
<td>95</td>
<td>100.0</td>
</tr>
<tr>
<td>javadoc</td>
<td>225</td>
<td>101.3</td>
</tr>
<tr>
<td>jess</td>
<td>193</td>
<td>101.0</td>
</tr>
<tr>
<td>obfuscator</td>
<td>643</td>
<td>101.6</td>
</tr>
<tr>
<td>pmd</td>
<td>245</td>
<td>100.4</td>
</tr>
<tr>
<td>sablecc-j</td>
<td>664</td>
<td>101.5</td>
</tr>
<tr>
<td>soot-c</td>
<td>835</td>
<td>101.8</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td>101.7</td>
</tr>
<tr>
<td>Median</td>
<td></td>
<td>101.4</td>
</tr>
</tbody>
</table>

Table 5.5: Experiment results for the side-effects analysis I.

Sensitive analysis and 90.4% for the Object-sensitive, on average. This large difference in the results has two reasons; the higher precision reduces the object set size in all program points, and the result of a context $ctx$ may influence the result of other contexts that call $ctx$, directly or transitively. If the context $ctx$ affects many objects stored in static variables, this result will influence contexts calling $ctx$.

**Analysis Times**

Some of the analysis times, shown in Table 5.8, have already been discussed, but it is worth discussing them a bit further. The points-to analysis time can be ordered, with Insensitive as the fastest, This-sensitive, CallString, and the slowest is Object-sensitive. This is also true for the basic analyses.

The most noticeable values are the times for emma using the Object-sensitive analysis. The Insensitive points-to analysis of emma is by far the one that takes the most time of the analyzed programs. The points-to analysis time is almost a magnitude slower than the second slowest, the javac1.3, and the basic analysis time is 30 times slower than the second slowest, the javacc.
5.5 Summary

We have presented two experiments that show how the client analysis framework meets our goal criteria. The first experiment evaluates the development speed of the framework. The second experiment shows how different parameterizations of the points-to analysis affect the precision and speed of two different client analyses.

The conclusion of the development speed experiment shows that the framework reduces the effort a developer has to put in the process of creating client analyses. The analysis precision and speed experiment shows that different precisions in the points-to analysis affect the analysis results of the client analysis, and that there is a tradeoff between time and precision for the different analysis sensitivities. It also shows that the precision of the points-to analysis is preserved in the client analysis. The relation between the sensitivities are the same, even though the difference between Object-sensitive and This-sensitive is smaller than it was in the points-to analysis results. It is also evident that Insensitive and This-sensitive provide the best trad-offs between precision and time. Both CallString and Object-sensitive provide less precision or takes considerably more time to complete, then Insensitive and This-sensitive.

<table>
<thead>
<tr>
<th>Program</th>
<th>$\text{SideAffected}_S$</th>
<th>$\text{SideAffected}_U$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insens (#)</td>
<td>CallString (%)</td>
</tr>
<tr>
<td>antlr</td>
<td>90,785</td>
<td>97.0</td>
</tr>
<tr>
<td>emma</td>
<td>449,934</td>
<td>78.6</td>
</tr>
<tr>
<td>javac1.3</td>
<td>142,219</td>
<td>87.3</td>
</tr>
<tr>
<td>javacc</td>
<td>304,358</td>
<td>96.8</td>
</tr>
<tr>
<td>javadoc</td>
<td>132,042</td>
<td>80.0</td>
</tr>
<tr>
<td>jess</td>
<td>71,644</td>
<td>93.2</td>
</tr>
<tr>
<td>obfuscator</td>
<td>755,432</td>
<td>93.8</td>
</tr>
<tr>
<td>pmd</td>
<td>454,189</td>
<td>90.6</td>
</tr>
<tr>
<td>sablecc-j</td>
<td>180,366</td>
<td>65.5</td>
</tr>
<tr>
<td>soot-c</td>
<td>398,433</td>
<td>77.5</td>
</tr>
<tr>
<td>Mean</td>
<td>86.0</td>
<td>9.5</td>
</tr>
<tr>
<td>Median</td>
<td>89.0</td>
<td>8.4</td>
</tr>
</tbody>
</table>

Table 5.6: Experiment results for the side-effects analysis II.
These conclusions motivate a framework for creating client analyses to points-to analysis, i.e., a framework would save effort for a developer of client analyses of points-to analysis.
Table 5.8: Analysis times for the escape and the side-effects analysis. The insensitive columns show time in seconds. The other columns show the factor relative the insensitive analysis time.
Part IV

The Present and the Future
Chapter 6

Conclusions

The contributions of this thesis are presented in this chapter. The conclusions are related to the research question and the goal criteria presented in section 1.3.

6.1 Research Questions

Here, we revisit the research questions and discuss the answers given by the previous chapters. The research questions were formulated like this:

1. What is needed to produce client analyses based on points-to results?
   (a) What are the commonalities and the differences of the analyses?
   (b) How could the analyses be specified?
   (c) How could the analyses be generated from such a specification?

One answer to the research question 1a is given in Chapter 4. The commonalities and differences between two client analyses, escape analysis and side-effects analysis are presented. This is not the exhaustive answer to the question, since we have only looked at these two client analyses. However, it is very likely that the found commonalities are used in other client analyses as well, since they are very basic and elementary. The differences between client analyses are reflected in a specification language. This specification language is currently not defined, which leaves space for improvement regarding research questions 1b and 1c. So far, the two client analyses are implemented in Java using an abstract class provided by the client analysis framework. So, the current answer to these two questions is that the a client analysis may be specified in Java using an Application Programming Interface (API) to access the functionality provided by the framework. The client analysis generation is performed by the Java compiler.

To get a more exhaustive answer to the research questions, we need to investigate more client analyses to find their requirements and decide what the framework should supply in form of reuse and decide the features and capabilities of the client analysis specification language.
6.2 State of the Art Revisited

The state of the art part of the thesis presents the foundation and discusses the related work, and the conclusions drawn from these are presented separately, below.

Foundations

Different program representations are presented and the use of the Sparse SSA graph in our points-to analysis is motivated. The general algorithm idea behind the points-to analysis is given along with a description of the simulated execution algorithm, which is applied in our points-to analysis. Concepts that are important in static analysis are presented such as name schema, conservative analysis, inter/intraprocedural analysis and call context-sensitivity.

Related Work

In the related works chapter, Chapter 3, we were not ready to compare the related work to the client analysis framework, since this had not been discussed so far. The work of others was related to a baseline approach, instead of to the client analysis framework. Here, we relate two of the three static analysis frameworks to the client analysis framework. The comments regarding the comparison of the monotone data flow framework and the baseline approach as presented in Section 3.2 are relevant when comparing it to our approach, as well. These comments are repeated here, for completion. Then, we relate our approach to the baseline approach and to the related work. We repeat the table with the comparison vectors from Table 3.1 in Section 3.5, and add the comparison vector for the client analysis framework.

Monotone Data Flow Framework  The monotone data flow framework, presented in Section 3.2, provides more of a design for the implementation of a data flow problem solver than help with the implementation directly. If the requirements on an instantiation are fulfilled, such as correctly defined monotone transfer functions, then it ensures termination of the analysis, etc. The specific details for how to implement data structures and algorithms to ensure efficiency in space and time are not given by the framework. An attempt to close the gap between this theoretical framework and a realistic solution is the Program Analyzer Generator (PAG) framework we present below.

The generality of the framework ensures that it can be instantiated to solve many data flow problems. However, the generality and the need for an instantiation degrades the development speed of the framework. There is no support for any algorithms or data structures. They have to be implemented by hand. In principle, it is possible to implement any sensitivity, and the
speed is all up to the implementation of the algorithms. Therefore, it is not possible to say anything about the analysis speed.

**PAG**  
PAG is concerned only with solving data flow problems. Many problems may be formulated as data flow problems, even though there are simpler solutions. The analyses that our framework aims to support could probably be defined along with the specification of the points-to analysis in PAG. This would, however, be less efficient compared to our approach, where intermediate analysis is provided for and reused in other client analyses.

The analysis precisions offered by PAG are also implemented in our analysis engine. Inlining is not feasible to use when analyzing larger programs, or programs with recursion. Invalidating does not give the precision needed in the general case. The approaches (e.g., call string) that may be simulated using the static call graph approach, have corresponding implementations in our engine [LL07]. When it comes to a trade-off between precision and speed, we have one approach that works well in practice; the *this-sensitive* approach. However, anything corresponding to this is not provided by PAG.

The PAG system lacks a number of features that we are able to provide, such as specific context-sensitivities and solving not only data-flow problems. These could perhaps be implemented in PAG, but then arguments for higher development speed in our proposed framework would be strengthened. It is not possible to argue for the analysis speed, since too many factors that influence the speed are unavailable.

**CoSy**  
The CoSy system provides a very general solution that can be extended to areas outside compiler generation and static program analysis. This comes with a penalty compared to our solution, which is more focused on our specific domain and thus more appropriate. It would be possible to provide the same functionality as our solution using CoSy, but it would require much work in the form of creating engines and specifications. The functionality and support it provides is too coarse-grained and require too much work to be instantiated. This would give us arguments in favor of our solution when it comes to development speed. It is not possible to say anything about the analysis precision or the analysis speed. It would be possible to implement the same precisions using CoSy, and probably get the same analysis speed as well, depending on how the instantiation is performed. We may define a composition specification that enables us to perform fix-point iterations over engine results, and thus create a more flexible analysis specification than is possible with CoSy. However, our framework could be seen as a CoSy component, solving a subproblem in a larger system. They are not comparable at that level, since they solve different problems.
The Client Analysis Framework  Here, we compare the client analysis framework to the baseline approach and create a comparison vector, as done for the existing client analyses in Section 3.3.

The client analysis framework provides support for client analysis developers in form of reuse of existing analyses, data structures and a client specification language that ease the implementation of client analyses. Therefore, its development speed is considered to be high. In Table 6.1 this is denoted with a ‘+’.

The client analysis framework uses an allocation schema that is context-insensitive and based on syntactic creation points. Its most precise context-sensitivity is the object-sensitivity and it models individual fields of each abstract object. The program representation is a Sparse SSA graph optimized for points-to analysis, also known as the Points-to SSA data structure. The client analysis framework does not use any particular heuristics in favor of speed that degrades the precision of the analysis. Its program representation ensures that it is both control flow- and data flow-sensitive. The resulting comparison vector is \([0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]\).

Conclusions  None of the static analysis frameworks or any of the related work provides more support for the client analysis developer than the client analysis framework does. In Table 6.1 in column “Dev. Speed”, the client analysis framework is given a ‘+’, which is the highest mark in the column. Both ‘−’, ‘S’ and ‘S+’ denotes a lower reuse support level.

A number of previous approaches presented in the client analysis section, Section 3.3, are awarded the ‘+’ in one position in their comparison vector. However, they all have at least one ‘−’ or ‘?’ in the vector as well. The result of this is that their comparison vectors are not comparable to the comparison vector of the client analysis framework, since it is neither larger nor smaller and that the vectors form a partial ordered set. We can conclude that none of the approaches has an advantage over the client analysis framework. The rest of the previous approaches have a comparison vector that indicates a lower precision than for the client analysis framework, which indicates that they are less precise or computationally more demanding.

6.3 Framework

Two client analyses are presented and specified, the escape analysis and the side-effects analysis. The specifications define the input data the client analyses require and how the data is used by the analysis. The two analyses share required input data types. The data structures fill the gap between the points-to analysis results and the information these two client analyses require. The two client analyses have points where they differ from each
### Criteria

<table>
<thead>
<tr>
<th>Frameworks</th>
<th>Work</th>
<th>Dev. Speed</th>
<th>Precision / Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MDFF [NNH99]</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>PAG[AMW95]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>CoSy</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

| Escape     | [BH99]             | –          | 0 | − | 0 | ? | 0 | 0 |
|            | [GS00]             | –          | 0 | − | 0 | 0 | 0 | − |
|            | [WR99, VR01]       | –          | 0 | − | 0 | 0 | − | 0 |
|            | [RMR01]            | –          | 0 | − | 0 | ? | 0 | 0 |
|            | [Bla99, Bla03]     | S          | 0 | − | 0 | 0 | ? | 0 |
|            | [CGS+99, CGS+03]   | –          | 0 | − | 0 | ? | − | 0 |
|            | [WL04]             | S+         | 0 | + | 0 | ? | − | 0 |
|            | [HS06]             | S          | 0 | − | 0 | − | 0 | 0 |

| Side-Effects | [Cla97]          | S          | ? | − | − | ? | − | 0 |
|              | [Raz99]          | −          | 0 | − | 0 | − | 0 | − |
|              | [TD03]           | −          | 0 | − | 0 | − | 0 | 0 |
|              | [WL04]           | S+         | 0 | + | 0 | ? | − | 0 |
|              | [Rou04]          | −          | 0 | − | 0 | ? | 0 | − |
|              | [NX05]           | −          | 0 | − | 0 | 0 | 0 | − |
|              | [MRR02, MRR05]   | S          | + | 0 | 0 | ? | 0 | − |
|              | [SR04, SR05]     | −          | 0 | − | 0 | 0 | − | 0 |

| The Client Analysis Framework | + | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Legend**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Idea for a simple reuse of source components,</td>
</tr>
<tr>
<td></td>
<td>not as advanced as baseline</td>
</tr>
<tr>
<td>S+</td>
<td>Idea for a simple reuse of source components,</td>
</tr>
<tr>
<td></td>
<td>more advanced than S, but not as advanced as baseline</td>
</tr>
<tr>
<td>+</td>
<td>Potentially better than baseline</td>
</tr>
<tr>
<td>−</td>
<td>Not as good as baseline</td>
</tr>
<tr>
<td>0</td>
<td>Same like baseline</td>
</tr>
<tr>
<td>?</td>
<td>Not Decidable</td>
</tr>
</tbody>
</table>

Table 6.1: Evaluation of the related work with respect to the goal criteria.
other, and these are presented as the variation points of the client analysis framework. The variation points can be summarized as the need for a specification of how to combine the basic analyses in order to get the intended client analysis and difference in analysis precision. A specification language is needed to fulfill this need.

The conclusion is that the client analysis framework should supply data structures and analysis algorithms providing results the client analyses are based on. How the client analyses use this data is specified in a specification language, that is used to generate executable client analyses.

### 6.4 Experiments

The experiments chapter presents two experiments, one evaluating the development speed goal criteria, and one evaluating the analysis precision and analysis speed. The development speed experiment argues in favor of high development speed for client analysis framework. The analysis precision and speed experiment shows that the precision of the points-to analysis shows in the client analysis results. High precision for the points-to analysis results gives high precision for the client analysis results. This motivates the use of different precisions for the points-to analysis. It also shows that there are possibilities for trade-off between analysis precision and analysis speed.

The results of the two experiments motivate the existence of a framework for creating client analyses to points-to analysis. The framework would save effort for a developer of client analyses of points-to analysis, and make it easier to find an appropriate trade-off between analysis speed and precision, related to the external expectations on the analysis, such as precision, time, and space constraints.
Chapter 7

Future Work

This chapter discusses a number of the possible research paths that can be followed from the results and questions presented in this thesis. A selection of these may be pursued in order to complete the doctoral thesis.

The first path is to implement the framework we specified in this thesis. Two possibilities use the client analyses that we have looked at and their analysis results to create a program comprehension tool and a component conformance-checking tool, respectively. The fourth option is to fine-tune one or both client analyses in order to make them perform better than the versions presented in literature so far. These four paths are presented in individual sections below.

7.1 Framework

This thesis aims to take the first steps towards a framework for creating client analyses based on points-to analysis results. There are some areas that are sketched in this thesis that need more work in order to complete the framework and ensure that it is efficient and provides the necessary functionality and support. A client analysis specification language and a way to interpret it, or generate code from it, are needed as we concluded in Section 4.4. The specification language should be able to specify how the client analysis should make use of the points-to analysis results and the basic analyses in order to calculate its result. This specification should be either interpreted or translated into runnable code of some sort. This requires that an interpreter or generator is constructed. The specifications of escape analysis and side-effects analysis we present in Chapter 4 are not solved with a dataflow approach, like the points-to analysis. It is rather a constraint-based approach using set of objects and SSA nodes from the program representation. It is possible that other client analyses are defined easily using the same approach, which could be investigated by defining in the same way. The success of such an investigation would enable us to motivate a constraint-based specification language.

The basic analyses are implemented but there are still opportunities to improve the performance and space properties of some of them. This would
be necessary if the framework were to be used in development tools and other environments having high demands on efficiency both considering speed and memory consumption.

Finally, we need to verify the framework, related to the research criteria posed in this thesis and its completeness when it comes to providing the necessary support for developing new client analyses. The evaluation of the framework related to the research criteria should especially focus on the development speed. One way of doing this is to perform a controlled experiment where developers use the framework to implement a set of client analyses and a control group implements the client analyses without the support from the framework. To establish that the functionality and support the framework supplies is appropriate it is necessary to implement other client analyses based on points-to analysis results. Specifying and implementing these analyses will show if the framework needs to provide more basic analyses or if the specification language should be extended or changed in some way. Getting input from other developers using the framework is an important step in this process.

7.2 Program Comprehension

The complexity of software causes problems when developers are facing the problem of understanding software they are not familiar with from before. Having tool support in this situation is necessary to understand the software. The client analyses we have studied in this thesis, i.e., escape and side-effects analysis, could be helpful when creating such a tool, and so would the read/write analysis. Presenting the escape and side-effects analysis result in an informative way to developers may help the developer understand the software.

The approach would contain three steps. First, we need a comprehension paradigm for the escape and side-effects analysis information, i.e., what does the information provide in relation to program comprehension, which parts of the analysis results are useful, how should the information be presented, etc. Second, we need to create a tool that implements the comprehension paradigm in a static analysis tool available to software developers. The tool should be able to analyze an arbitrary program, under reasonable constraints. The user should be able to interact with the tool and visualize, filter and focus the information to make it useful. Third, this tool needs to be evaluated against criteria, to show that it is useful and actually gives a gain to the developers in understanding software. The criteria should include speed and correctness of the information the tools provide. This could be done in a controlled experiment where a group of developers may use the tool to understand new software and then make changes or answer questions about it.
A control group should produce artifacts providing the same functionality, but without the use of the tool.

### 7.3 Component Conformance

Reuse of software is very popular and important when it comes to lowering cost of software development, shortening the time-to-market for a software product, raising the dependability and reducing the bugs and faults introduced into the developed software. Certain functionality is packaged into components, which are then reused. Exchanging a component for another component performing the same tasks and providing the same functionality implies that the second component conforms to the first. For a component to be appropriate for a specific task in a software system it must conform to the requirements the system poses on the component. These requirements may be both behavioral and technical, i.e., it should provide the needed functionality and deliver it in the required way, and it should comply to the system’s hardware and software requirements.

The analyses that are presented in this thesis, i.e., points-to, client and basic analyses, may be helpful in creating a tool to determine if a component \(c_y\) conforms to another component \(c_x\). When the component \(c_x\) is used in a software system, it should already conform to the requirements of the system. The hypothesis here is that if the component \(c_y\) is shown to behave, i.e., write to and read from the same objects, in the same way as \(c_x\) when it comes to side-effects and escaping objects, it could be said to conform to \(c_x\) and be used in the system instead of \(c_x\). Additionally, how the system reads from and writes to the objects escaping from the component should also be part of the conformance theory. To realize this idea it is necessary to create a theory of when components conform and when they do not. This theory should specify what properties of the analysis results gives conformance and not. The correctness and usefulness of this theory then needs to be evaluated. The evaluation could be performed in an experiment where different systems are analyzed by the tool. The measurements could be of precision and recall of conformance between components at different positions in the systems. The conformance at these positions should be established manually in advance. The tool should mark the positions where there is a conformance issue. The percentage of positions marked correctly is the precision and ration between the number of correctly identified positions and the number of total positions that have conformance issues is considered to be the recall.
7.4 Improve Client Analyses

The client analyses that are presented in the literature and discussed in Chapter 3 use various methods and approaches. They use different sensitivities in their analyses and this could leave opportunities for improving their analysis result and in that way creating a more suitable side-effects analysis or escape analysis. It is possible that the use of the client analysis results that are presented leaves chances of improvements as well. For instance, an investigation of the restrictions on when objects are stack allocatable could lead to loosened restrictions so that more objects may be stack allocated.

To realize this track it is necessary to identify the potential of improvement in analyses presented in literature and improve at least one analysis. The improvement should be significant and at least applicable under the same constraints as the targeted analysis. The client analyses previously presented all target the Java programming language but they have individual restrictions for their results to be applicable. A more generally applicable analysis, that works in a number of these contexts, while still performing as well as or better than the existing analyses could be a research result as well.

It is important to establish there is a need and a general value of a better analysis before pursuing this track. The evaluation should focus on the criteria precision and performance with consideration to the client’s application context. The precision of the analysis results should come through to its applications, and the result of these applications should be evaluated. The performance is concerned with both the speed and the memory consumption. The evaluation could be performed by experiments, where the analysis is used on the benchmark programs used in literature, in order to compare to the existing analyses.
Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography
