Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

Note

Analysis of object-oriented programs: a survey

Note no DART/11/05
Author Bjarte M. Østvold
and Thor Kristoffersen

Date August 19, 2005

Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

The author

Use \aboutauthors to enter author information.

Norwegian Computing Center

Norsk Regnesentral (Norwegian Computing Center, NR) is a private, independent, non-profit foundation established in 1952. NR carries out contract research and development projects in the areas of information and communication technology and applied statistical modeling. The clients are a broad range of industrial, commercial and public service organizations in the national as well as the international market. Our scientific and technical capabilities are further developed in co-operation with The Research Council of Norway and key customers. The results of our projects may take the form of reports, software, prototypes, and short courses. A proof of the confidence and appreciation our clients have for us is given by the fact that most of our new contracts are signed with previous customers.

Norsk Regnesentral
Norwegian Computing Center
Postboks 114, Blindern
NO-0314 Oslo, Norway

Besøksadresse
Office address
Gaustadalléen 23
NO-0373 Oslo, Norway

Telefon · telephone
(+47) 22 85 25 00
Telefaks · telefax
(+47) 22 69 76 60

Internett · internet
www.nr.no
E-post · e-mail
nr@nr.no

Title	**Analysis of object-oriented programs: a survey**
Author	**Bjarte M. Østvold** <bjarte@nr.no> **and Thor Kristoffersen** <thor@nr.no>
Date	August 19, 2005
Year	2005
Publication number	DART/11/05

Abstract

We survey code analysis research, applications and infrastructure, focusing on object-oriented languages. In particular we cover static and dynamic program analysis, including non-conservative analysis, and also consider language extensions facilitating program analysis, and libraries and tools for doing analysis. The discussion outlines challenges and opportunities for program analysis when applied to object-oriented applications development.

Keywords	program analysis, static analysis, dynamic analysis
Target group	Project group, partners
Availability	Open
Project	Reducing Software Entropy
Project number	802000
Research field	program analysis, object-oriented programming
Number of pages	15

Contents

1 Introduction

This documents surveys, in a broad sense, the theory and practise of code analysis, with emphasis on class-based object-oriented languages such as Java and C#.

Roughly code analysis may be divided into two kinds: static analysis and dynamic analysis. Static analysis is concerned with analysing program code to derive properties about it, but without considering concrete inputs. Static analysis derives abstract properties that hold for all program executions. Dynamic analysis seeks to derive properties of a program based on running it on various inputs. Properties found by dynamic analyses can be about concrete program values, but they hold only for the analysed executions.

The appearance of modern techniques such as just-in-time compilation and classloaders means that there is no longer a clear distinction between, on the one hand compile-time and static analysis, and on the other hand, run-time and dynamic analysis. An example is JIT compilation, sometimes called jitting, is a byte-code to assembly compiler built into a virtual machine. Here there are two levels of interacting compilation, before and mixed into, execution of a program.

The traditional goal of code analysis, or program analysis as it then called, is to improve performance directly or to analyse a program's performance as input to the programmer. Typical examples of such program analyses are register allocation (static) and profiling (dynamic). There are, however, also analyses for determining other properties such as security properties, and

2 Static analysis

Classical static analysis determines precise properties of program variables and parts. A introduction to basic static analysis may be found in Appels compiler textbook [3, §10,§17]; we base part of our presentation on his. Static analysis is a well developed field and static analyses are widely deployed as part of optimising compilers.

2.1 Call graphs and data/control flow analysis

A call graph is a useful data structure for static analyses. Program statements are the nodes and there is an edge from statement s to statement t if execution may, after performing s, go to immediately to t. In a procedural language without pointers call graph construction is trivial: the target of a call depends solely on the code at the call site. In object-oriented languages, however, this is no the case. The actual method invoked depends on the class of the object on which the method call is performed. Grove and Chambers remark the following on the relation between call graphs and flow analyses for object-oriented (and functional) languages:

> In general, determining the flow of values needed to build a useful call graph requires an interprocedural data and control flow analysis of the program. But interprocedural analysis in turn requires that a call graph be built prior to the analysis being performed. This circular dependency be-tween interprocedural analysis and call graph construction is the key technical difference between interprocedural analysis of object-oriented and functional languages (collectively called higher-order languages) and interprocedural analysis of strictly first-order procedural languages. Effectively resolving this circularity is the primary challenge of the call graph construction problem for higher-order languages. [14]

Finding the liveness of variables is an example of dataflow analysis. Here we briefly consider interprocedural liveness analysis. The analysis seeks to determine what program variables are *live* when, that is, at what positions in a program variables hold values that are later required by the computation of the program. Given information about definitions and uses for each variable, liveness calculation proceeds in iterations to solve equations about variable liveness. The analysis results may be used to determine the number of temporary variables needed and allocate registers as a preparation for code generation. As many other flow analysis, liveness analysis is undecidable. Therefore analysis algorithms compute conservative estimates, and in addition makes trade-offs between precision of the analysis and the cost of performing it.

Examples of other dataflow analysis are common subexpression elimination and dead-code elimination. Dataflow analysis algorithms, and smart intermediate representations for such, is an active field of research.

2.2 Points-to analysis

Points-to analysis, also called reference analysis [30], is concerned with determining (properties of) the set of objects to which a reference variable or field may point.[1] Points-to analysis forms the basis for other analyses [22]:

- Side-effect analysis: determine which memory locations a statement execution changes.
- Def-use analysis: relate all use sites for a variable to all definition sites for that variable and vice versa.
- Alias analysis: find references that may point to the same memory location.
- Escape analysis: determine the set of objects which are reachable at method return, that is, the set of objects that 'escape' from the method.

Improving points-to analysis is a subject of current research [6, 20].

2.3 Type-based and constraint-based analysis

Type systems [7, 26] play an important role in capturing static information about programs. Program analysis for languages with static type systems can assume that programs type check and take advantage of this fact during analysis. Also, program analyses may be formulated as type analyses. Palsberg surveys work on type-based analysis [25].

Aiken [1] argues that constrains allows good separation of the specification of an analysis from its implementation. Furthermore he argues that constraints is a natural form of specification and that leveraging constraint theory is useful. Constraint-based analysis divides analysis into constraint generation and constraint resolution. Other analyses may be recast as constraint-based analysis.

3 Dynamic analysis

Dynamic analysis seeks to derive properties about a program from one or more executions of the program. As such one may consider traditional programmer activities such as program profiling, debugging as testing as rudimentary dynamic analyses. Likewise, compiler and VM technology such as just-in-time compilation [18] and run-time optimisation [27]

Ball [5] argues as follows that dynamic program analysis is useful. First, when instrumenting

1. The term pointer analysis is usually reserved for a analysis of languages with C-like pointers.

a program for recording of execution-time data the analyst can gather exactly the information he or she wants. Second, and in contrast to static analysis, dynamic analysis can relate inputs to program state and outputs. Ball [5] also presents two kind of analyses: Frequency spectrum analysis, which measures the occurrence frequencies of program elements in a program execution, and coverage concept analysis, which records what program elements where executed in an execution and group the recorded data based on so-called concept analysis.

Rountev*et al.* [28] combine static and dynamic analysis of Java call chains, that is, edges in the call graph (cf. Section 2.1).

Dufour *et al.* [12] define a family of dynamic metrics for a unambiguously characterising properties of program runs. The accompanying *J tool realises these metrics (Section 4.2).

Massalin's superoptimiser [21] that find the smallest instruction sequence that realises a given function, is an interesting combination of static and dynamic analysis.

4 Analysis tools and frameworks

All tools discussed here have available and liberal licences.

4.1 Non-conservative analysis

Classical static analyses techniques are *conservative* in the sense that they only draw sound conclusions, that is, conclusions that follow logically from the program. Since such analysis is the basis for optimising programs, it better be conservative: nearly-dead code elimination would not be a useful compiler optimisation phase. To increase the power of a conservative analysis one must either improve the analysis algorithm,[2] or make additional information about the program, other than the program source, available for analysis. The first is the subject of much current research in static program analysis. The second is less investigated in conservative static analysis since for the analysis to remain strictly conservative, the additional information must not conflict with the program. Ensuring this is a hard problem in itself.

Non-conservative analysis is directly relevant to program analysis, since analysis results must serve as inputs to developers, and not as input to compiler optimisation. Examples of systems performing non-conservative analysis appear in some of the following sections.

4.2 Dynamic program comprehension

This section lists tool that aids the programmer in understanding a running program. We leave out traditional tools, such as debuggers and profilers.

The Java PathExplorer [16] monitors an executing Java program and checks that the execution conforms to a property specification in temporal logic. It is again based on the Maude rewriting system.[3]. The Java PathExplorer can also indicate possible deadlocks and data races.

Caffeine [15] combines dynamic analysis using the Java Platform Debug Architecture and a Prolog-based query engine to check conjectures about the program.

The tool *J [13][4] is available for dynamic analysis of Java programs, based on metrics [12].

JNuke [4] is a custom Java Virtual Machine for dynamic analysis facilitating efficient analysis

2. The power of conservative analysis has theoretical limits imposed by computability theory.
3. `http://maude.cs.uiuc.edu/`
4. `http://www.sable.mcgill.ca/starj/`

with full access to program state and backtracking. JNUke does run-time verification and model checking of Java programs.

4.3 Bug detection

ESC/Java2 translates Java sources files, with JML [9] annotations, into a form that can be analysed by a theorem prover. The translation and prover together reason about the consistency between the Java code and the annotations, where the latter capture low-level design information. The analysis points to possible bugs in the code, but due to the non-conservative analysis the bugs indications must be verified by human programmers, and no bugs reported does not mean there are none.

The FindBugs tool[5] also applies non-conservative analysis to locate typical Java programming bugs and may also report false positives. FindBugs applies *bug-patterns*, description of undesirable Java programming practises, and standard static program analysis.

4.4 Style checking

PMD[6] checks Java programs for style-errors such as empty try-catch blocks, unused variables and parameters, and empty if-statements. In addition PMD can detects 'overcomplicated' parts of the code and identifies duplicated code (instances of 'cut-and-paste programming). The tools has a large number of pre-defined rules and allows the user define new ones. The released version considers only one source file at a time and does not currently do dataflow analysis. PMD uses a JavaCC-generated parser.

PMD has support for finding copied and pasted code. Another tool for doing this is Simian.[7] Both compare tokenised Java programs, that is, they do not take the full language grammar into account. The tools have options for ignoring constants and identifiers.

Checkstyle[8] is checks that a Java program follows stylistic conventions, for example, there is a pre-defined rule set defining the Sun Code Conventions.

Structural Analysis for Java[9] analyses structural dependencies in Java applications, including checking for design anti-patterns, and presents the results graphically.

4.5 Compilation-related

This sections lists compiler-related tools relevant when implementing program analyses.

The Polyglot extensible compiler framework [23][10] facilitates creation of variants of Java through an extensible parser, new compiler passes and code generation. Polyglot translates Java variants into pure Java. The framework is itself also written in Java.

Stratego/XT[11] is a language for specifying program transformations, based on rewriting strategies and XT is an add-on to Stratego that deals with parsing and pretty-printing of programs. Stratego has applications in program transformation and optimisation, and promotes languages-independence of transformations [24].

Jikes[12] is an open source, high-performance Java byte-code compiler that has two advantages

5. `http://findbugs.sourceforge.net/`
6. `http://pmd.sourceforge.net/`
7. `http://www.redhillconsulting.com.au/products/simian/`. Note that commercial use of Simian costs money.
8. `http://checkstyle.sourceforge.net/`
9. `http://www.alphaworks.ibm.com/tech/sa4j`
10. `http://www.cs.cornell.edu/Projects/polyglot/`
11. `http://catamaran.labs.cs.uu.nl/twiki/pt/bin/view/Stratego/WebHome`
12. `http://jikes.sourceforge.net/`

over other Java compilers. First, it performs dependency analysis, which makes it possible to do incremental builds and makefile generation. Second, it offers constructive assistance by providing clear error and warning text to assist the programmer in understanding problems.

Jikes RVM[13] is an advanced virtual machine for research and experiments with such machines. Jikes RMV is written almost entirely in Java and targets Linux on Intel and a few less common configurations. A dynamic analysis could be realised using an instrumented Jikes RVM.

Soot[14] [19] is an optimising Java bytecode compiler written in Java. It is available as an Eclipse plugin, and it can be used for analysis and transformation of Java bytecode, including decompilation. Whole-program analysis is also possible. Soot has several intermediate representations (IRs), of which the most important are Baf, Jimple, and Shimple. Baf is an IR that is very much like Java bytecode, but slightly more high-level. In particular, everything is typed. This makes it more convenient to manipulate the code. Jimple is a typed, stackless, 3-address code representation of Java bytecode (Java Simple) in a flow graph. Shimple is a Static Single Assignment (SSA) version of Jimple. The essential property of the SSA form is that every variable receives, in the static context, only one assignment during its lifetime. This is a very efficient form for doing optimisations based on dataflow. Other, less important IRs are Grimp and Dava. Grimp is like Jimple, but with aggregated expressions, and Dava is a structured representation suitable for Decompiling Java.

The Byte Code Engineering Library (BCEL)[15] of the Apache Jakarta project[16] is a library and API for creating and manipulating Java byte-code. It is used by the FindBugs project. ASM [8][17] is a similar software, but it trades completeness for speed and is optimised for dynamic code generation.

CUP[18] and JLex[19] are Java tools similar to Yacc and Lex for C. CUP produces LALR parsers. There is no special facility for generating parse trees.

BYACC/J[20] is an LALR parser generator that emphasises compatibility with Yacc for C. In particular, it can parse existing Yacc grammars.

SableCC[21] is an LALR parser generator. It does not have any actions, but instead generates a parse tree.

Beaver[22] is an LALR parser generator capable of generating very fast parsers.

JavaCC[23] is an open source parser generator for Java, similar to YACC for Unix and C. However, JavaCC generates LL(k) recursive descent parsers, unlike YACC which generates LALR parsers. This makes the grammar specification less powerful than in YACC, but this is compensated for by several inelegant workarounds, like multiple token lookahead, syntactic lookahead, and semantic lookahead. JavaCC does not in itself produce parse trees, but the authors of JavaCC have provided an add-on called JJTree that does. There is also another third-party add-on called Java Tree Builder[24] (JTB) that produces parse trees for JavaCC. JTB is similar to JJTree, but simpler and less flexible.

---

13. http://jikesrvm.sourceforge.net/
14. http://www.sable.mcgill.ca/soot/, http://freshmeat.net/projects/soot/
15. http://jakarta.apache.org/bcel/
16. http://jakarta.apache.org/
17. http://asm.objectweb.org/
18. http://www2.cs.tum.edu/projects/cup/
19. http://jflex.de/
20. http://byaccj.sourceforge.net/
21. http://sablecc.org/
22. http://beaver.sourceforge.net/
23. https://javacc.dev.java.net/
24. http://compilers.cs.ucla.edu/jtb/

ANTLR[25] is a public domain parser generator for Java that is also LL(k) and hence fraught with the same problems as JavaCC. However, ANTLR has built-in support for generating parse trees. There is a default format, but other custom formats, including XML, are possible.

5 Research problems

In this section we list various research problems in program analysis, with slight emphasis on analyses relevant for agile software development.

In general, analysis that leverage design or architecture information about the application being analysed seems an interesting subject, especially for analysis results intended only for human consumption. This requires, however, that such information be available. Ammons *et al.* [2] consider the related problem "specification mining"

Also, for analysis to be useful in large software projects incremental analysis is important to save time and/or computational power; an incremental analysis may be computed locally before check-in into a central code repository. This allows the developer to inspect the analysis results and possibly revise the code. The alternative is a centrally run analysis computed at regular intervals for the whole project. Compositionality of analyses is also an issue: to what degree can analyses of modules be composed to an analysis of a larger system.

5.1 Frameworks and partial analysis

Most deployed object-oriented applications no longer run on a minimal core of standard libraries. Instead applications are typically built using extensible frameworks and other third party libraries and APIs. Java 2 Enterprise Edition (J2EE) represents an important group of such frameworks. Thus, deployed applications consist of custom-written source code and pre-compiled framework and library code, intermingled by instantiation, inheritance and callbacks.

Analysing such applications cannot be done by reviewing the application source code in isolation—the framework (and third party APIs) must be taken into account. There are at least two possible approaches:

- Either the framework must also be analysed, assuming its source or byte-code is available. Analysing byte-code has the disadvantage that byte-code generation typically loses information that is potentially useful for analysis. Also, the code-base of a typical framework is so extensive that analysing it is prohibitively expensive. Realistic analysis needs some knowledge about the framework to guide analysis. Rountev *et al.* [29] present a general approach for adapting whole-program class analysis to analysis for program fragments.
- Alternatively, a specification delineating the relationship between the application and the framework is required, with the obvious disadvantage that this somebody must write this specification. Sweeney and Tip [31] discuss such a specification language.

5.2 Leveraging meta-information

Application development frequently includes writing non-source meta-information that may also be the subject of, or inform, analysis. Examples of such information are numerous: Enterprise Java Beans (EJB) deployment descriptors that specify, for example, name, persistence, state-fullness, and abstract query and schema information; build scripts; unit tests or other kinds of test code;

25. `http://www.antlr.org/`

configuration files. There appears to be little work on this with the exception of deployment-time checking of EJBs [10].

5.3 Catering to the interactive setting

Programmers have come to expect more of their code writing tools than what is provided by a text editor. Integrated development environments (IDEs) are language-sensitive, supports advanced code browsing, performs context-sensitive generation of code snippets and supports refactoring the code. Eclipse is a state-of-the art IDE for Java. It is easily extensible through a plug-in architecture and its Java development tooling API[26] (JDT). Based on the JDT developers can realise new ways of editing, compiling and viewing Java programs.

Presenting non-conservative analysis results (Section 4.1) to the programmer inside an IDE appears interesting for the following reasons:

- The programmer can browse the results, and the code, inside a familiar environment.
- A programmer can use the IDE to fine-tune the analysis or the presentation of results to his or her personal preference, for example, suppressing false positives.
- Incremental update of analysis results on editing and immediate presentation to the programmer. This gives an instant gratification effect.

In fact, Eclipse already supports simplistic edit-time analysis: if the programmer references an unknown method of a class Eclipse suggests to add a new definition for the method.

References

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, 1999.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[3] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[4] Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. JNuke: Efficient dynamic analysis for Java. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 462–465. Springer, 2004. ISBN 3-540-22342-8.

[5] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and M. Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 1999. ISBN 3-540-66538-2.

[6] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114. ACM, 2003. ISBN 1-58113-662-5.

[7] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.

26. See the topic JDT Plug-in Developer Guide, in the Eclipse IDEs interactive help.

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Systèmes à composants adaptables et extensibles*, Grenoble, 2002.

[9] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005. To appear.

[10] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In Crocker and Steele Jr. [11], pages 374–387.

[11] Ron Crocker and Guy L. Steele Jr., editors. *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, 2003. ACM.

[12] Bruno Dufour, Karel Driesen, Laurie J. Hendren, and Clark Verbrugge. Dynamic metrics for Java. In Crocker and Steele Jr. [11], pages 149–168.

[13] Bruno Dufour, Laurie J. Hendren, and Clark Verbrugge. *j: a tool for dynamic analysis of Java programs. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 306–307. ACM, 2003. ISBN 1-58113-751-6.

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

[15] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine: A tool for dynamic analysis of Java programs. In *ASE*, pages 117–. IEEE Computer Society, 2002. ISBN 0-7695-1736-6.

[16] Klaus Havelund and Grigore Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[17] Görel Hedin, editor. *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003*, volume 2622 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-00904-3.

[18] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 326–336, 1994.

[19] Jennifer Lhoták, Ondrej Lhoták, and Laurie J. Hendren. Integrating the Soot compiler infrastructure into an IDE. In Evelyn Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2004. ISBN 3-540-21297-3.

[20] Ondrej Lhoták and Laurie J. Hendren. Scaling Java points-to analysis using SPARK. In Hedin [17], pages 153–169. ISBN 3-540-00904-3.

[21] Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 122–126. IEEE Computer Society Press, 1987. ISBN 0-8186-0805-6. doi: http://doi.acm.org/10.1145/36206.36194.

[22] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1): 1–41, 2005.

[23] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Hedin [17], pages 138–152. ISBN 3-540-00904-3.

[24] Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In Rastislav Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer, 2005. ISBN 3-540-25411-0.

[25] Jens Palsberg. Type-based analysis and applications. In *PASTE*, pages 20–27. ACM, 2001. ISBN 1-58113-413-4.

[26] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[27] Feng Qian and Laurie J. Hendren. Towards dynamic interprocedural analysis in JVMs. In *Virtual Machine Research and Technology Symposium*, pages 139–150. USENIX, 2004.

[28] Atanas Rountev, Scott Kagan, and Michael Gibas. Static and dynamic analysis of call chains in Java. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 1–11. ACM, 2004. ISBN 1-58113-820-2.

[29] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, 2004.

[30] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Hedin [17], pages 126–137. ISBN 3-540-00904-3.

[31] Peter F. Sweeney and Frank Tip. Extracting library-based object-oriented applications. In *SIGSOFT FSE*, pages 98–107, 2000.