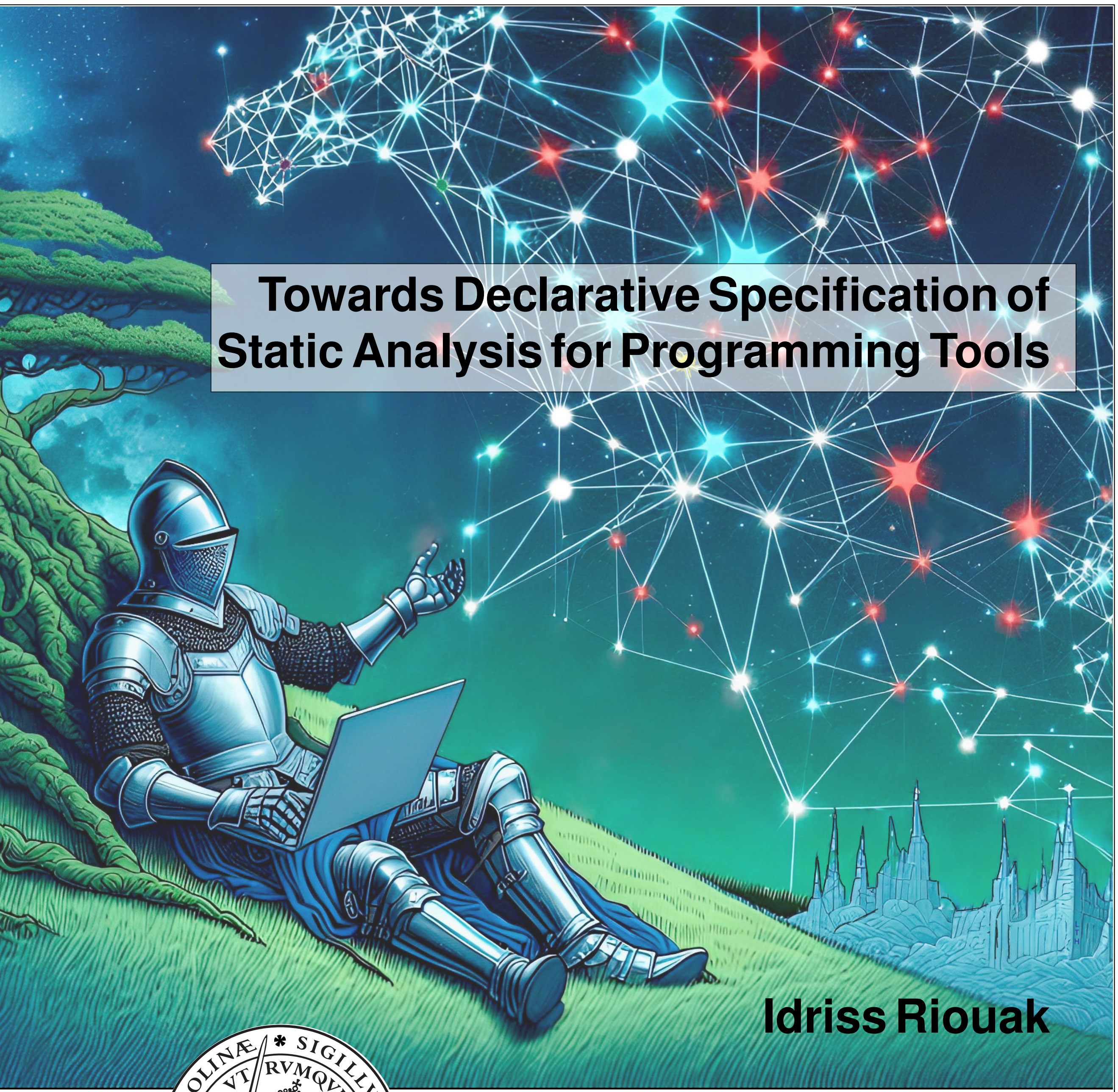


Towards Declarative Specification of Static Analysis for Programming Tools

Idriss Riouak

Doctoral Dissertation, 2024

Department of Computer Science
Lund University

LUNDS UNIVERSITET

ISBN 978-91-8104-242-9 (electronic version)
ISBN 978-91-8104-241-2 (print version)
ISSN 1404-1219
Doctoral Dissertation 78, 2024
LU-CS-DISS: 2024-05

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: `idriss.riouak@cs.lth.se`

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2024

The cover of this thesis draws inspiration from the iconic *Dragon Book* [Aho+07] by Aho, Sethi, and Ullman. Throughout my PhD journey, I often found myself feeling like the knight on its cover—facing difficult challenges, dealing with complex problems, and looking for answers in the literature. Like the knight, I tried to advance, equipped with little more than curiosity and determination. This image shows those moments of struggle and reflection, a reminder of the mix of uncertainty, hope, and moments of happiness that come with exploring a complex field like compiler technology and program analysis.

Abstract

Static program analysis plays a crucial role in ensuring the quality and security of software applications by detecting bugs and potential vulnerabilities in the code. Traditionally, these analyses are performed offline, either as part of the continuous integration / continuous deployment pipeline or overnight on the entire repository. However, this delayed feedback disrupts developer productivity, requiring context switches and adding overhead to the development process. Integrating these analysis results directly into the integrated development environment (IDE), similar to how type errors or code smells are reported, would enhance the development process. As developers increasingly rely on IDEs for real-time feedback, the efficiency and responsiveness of these tools have become critical. In such settings, developers expect immediate and precise results as they write and modify code, making it particularly challenging to achieve response times sufficiently low to not interrupt the thought process.

This thesis starts addressing these challenges by investigating the design and implementation of control-flow and dataflow analyses using the declarative Reference Attribute Grammars formalism. This formalism provides a high-level programming approach that enhances expressivity and modularity, making it easier to develop and maintain analyses.

Central to this thesis is the development of IntraCFG, a language-agnostic framework designed to perform control-flow and dataflow analyses directly on source code rather than relying on intermediate representations. By superimposing control-flow graphs onto the abstract syntax tree, IntraCFG removes the need for intermediate representations that are often lossy and expensive to generate. This approach allows for the construction of efficient but still precise dataflow analysis.

We demonstrate the effectiveness of IntraCFG through two case studies: IntraJ and IntraTeal. These case studies showcase the potential and flexibility of IntraCFG in diverse contexts, such as bug detection and education. IntraJ supports the Java programming language, while IntraTeal is a tool designed for teaching program analysis for the educational language Teal. IntraJ has proven to be faster than, and as precise as, well-known industrial tools.

Additionally, this thesis introduces a new algorithm for the demand-driven evaluation of fixed-point (i.e., circular) attributes, which has proven essential for the performance of dataflow analyses in IntraJ. This improvement allows IntraJ to achieve response times below 0.1 seconds, making it suitable for use in interactive development environments.

Contribution Statement

This thesis is a compilation consisting of an introduction and four papers. Three of this thesis' papers are published, and one is under review. The included papers are the following:

Paper I Idriss Riouak, Christoph Reichenbach, Görel Hedin and Niklas Fors.
"A Precise Framework for Source-Level Control-Flow Analysis".
In *21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 1–11.* Virtual, 2021, IEEE.
Paper DOI: 10.1109/SCAM52516.2021.00009.

Paper II Idriss Riouak, Niklas Fors, Görel Hedin, and Christoph Reichenbach.
"IntraJ: An On-Demand Framework for Intraprocedural Java Code Analysis".
Submitted for publication.

Paper III Idriss Riouak, Görel Hedin, Christoph Reichenbach and Niklas Fors.
"JFeature: Know Your Corpus!".
In *22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 236–241.* Limassol, Cyprus, 2022, IEEE.
Paper DOI: 10.1109/SCAM55253.2022.00033.

Paper IV Idriss Riouak, Niklas Fors, Jesper Öqvist, Görel Hedin and Christoph Reichenbach.
"Efficient Demand Evaluation of Fixed-Point Attributes using Static Analysis".
In *17th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, Pasadena, California, USA, 2024.
Paper DOI: 10.1145/3687997.3695644.

The table below indicates the responsibilities Idriss Riouak had in writing each paper:

Paper	*Writing*	*Concepts*	*Implementation*	*Evaluation*	*Artifact*
I	◐	◐	●	●	●
II	◕	◐	●	●	●
III	◕	◐	●	●	●
IV	◐	◕	●	●	●

The dark portion of the circle represents the amount of work and responsibilities assigned to Idriss Riouak for each individual step:

◐ Idriss Riouak was a contributor to the work

◕ Idriss Riouak led and did a majority of the work

● Idriss Riouak led and did almost all of the work

Artifacts and Awards Related to the Thesis

Artifact I: **Paper 1** introduces the IntraCFG and IntraJ static analyzer. The artifact includes the source code for both frameworks, along with the scripts used for the evaluation section presented in the paper. It was submitted to the ROSE (**R**ecognizing and Rewarding **O**pen Science in **SE**) festival and awarded the "*Open Research Objects*" and "*Research Objects Reviewed*" badges.
DOI: 10.5281/zenodo.5296618.

Artifact II: IntraTeal is an instance of IntraCFG used in teaching, briefly described in Section 4 of this thesis. The artifact contains the source code of IntraTeal.
DOI: 10.5281/zenodo.7649171.

Artifact III: **Paper 3** introduces the JFeature framework. The artifact includes the source code for JFeature, along with the scripts used for the evaluation section presented in the paper. It was submitted to the ROSE (**R**ecognizing and Rewarding **O**pen Science in **SE**) festival and awarded the "*Open Research Objects*" badge.
DOI: 10.5281/zenodo.7053768.

Artifact IV: **Paper 4** introduces a new algorithm for the evaluation of circular RAG attributes. The artifact contains the source code and evaluation results used in the paper. It was submitted for the SLE Artifact Evaluation

and awarded the badges "*Artifacts Available*" and "*Artifacts Evaluated - Reusable*".
DOI: 10.5281/zenodo.13365896.

Artifact V: The source code for the Callgraph Analysis Tool (CAT), used in **Paper 4**, is available on GitHub at: https://github.com/IdrissRio/cat.

Award: A short version of **Paper 4** was presented at the ACM Student Research Competition at <*Programming*> 24, Lund, Sweden, where it received first place in the graduate category. The paper, titled *"Using Static Analysis to Improve the Efficiency of Program Analysis,"* is available at ACM: https://src.acm.org/binaries/content/assets/src/2024/idriss-riouak.pdf.
For a complete list of participants and winners, see ACM Student Research Competition 2024.

Open Research Objects: *"Placed on a publicly accessible archival repository. A DOI or link to this persistent repository along with a unique identifier for the object is provided. Artifacts have not been formally evaluated."*

Research Object Reviewed: *"Artifacts documented, consistent, complete, exercisable, and include appropriate evidence of verification and validation,"* and *"very carefully documented and well-structured to the extent that reuse and repurposing is facilitated. Norms and standards of the research community are strictly adhered to."*

Artifacts Available: *"Author-created artifacts relevant to this paper have been placed on a publically accessible archival repository. A DOI or link to this repository along with a unique identifier for the object is provided."*

Artifacts Evaluated - Reusable: *"The artifacts associated with the paper are of a quality that significantly exceeds minimal functionality. That is, they have all the qualities of the Artifacts Evaluated - Functional level, but, in addition, they are very carefully documented and well-structured to the extent that reuse and repurposing is facilitated. In particular, norms and standards of the research community for artifacts of this type are strictly adhered to."*

Badges awarded to the artifacts. *Sources:* `https://icsme2021.github.io/cfp/AEandROSETrack.html`, `https://cyprusconferences.org/icsme2022/call-for-joint-artifact-evaluation-track-and-rose-festival-track/`, and `https://www.acm.org/publications/policies/artifact-review-and-badging-current`.

Acknowledgements

This journey has been filled with challenges, moments of joy, and invaluable support from many people. As I reach the end of this chapter, I would like to take a moment to express my deepest gratitude to those who have made this achievement possible.

First and foremost, I extend my deepest thanks to my main supervisor, *Görel Hedin*. Your guidance, patience, and belief in me have shaped my research in ways I could never have imagined. Every step of this journey has been made lighter by your understanding and constant encouragement. The support and suggestions you offered have been truly invaluable. I couldn't have asked for a better mentor.

I also want to express my deep appreciation to *Christoph Reichenbach*. Your constant support and belief in me made all the difference. You stood by me through the ups and downs, always ready with advice, guidance, and a bit of humor. I'm truly grateful for everything. *Niklas Fors*, thank you for always being available, ready to listen and offer advice, no matter how many times I dropped by your office. Your time and advice were invaluable to my progress.

I am also deeply grateful to Jesper Öqvist, Alfred Åkesson, Emma Söderberg, and every member of the SDE group, whose insights and support guided me throughout this journey. Your contributions, large and small, have left an indelible mark on this work.

Special thanks go to Anton Risberg Alaküla for our wonderfully strange conversations, especially about how vegetables should wear ties—discussions that were, oddly enough, more refreshing than anything else. Momina Rizwan, thank you for the delightful talks about food that somehow made the challenges of academia seem a little more digestible. To Noric and Flavius, for introducing me to the joys (and occasional pain) of bouldering, thank you.

I owe a big thank you to the "kids lunchroom table": Sergio, Matthias, Rikard, Gustaf, Gareth, Mike^2, Esra, Ayesha, Faseeh, and everyone else I haven't mentioned. You were the bright spots during the long days, and I am grateful for the laughter, the debates, and the very interesting conversations.

To Lars Bendix, thank you for the endless conversations about teaching, academia, Sweden, and Denmark. Your insights and humor were always a breath of fresh air. Anders Bruce, I am especially grateful for your patience and help with all my unusual requests—like the time I asked for a bigger whiteboard. Your kindness never went unnoticed.

My heartfelt thanks also go to Carina, Heidi, Ulrika, Peter, and Birger for your constant help with all the practicalities. Your support made everything run smoothly, and I am truly grateful for everything you did.

On a more personal note, I want to thank my family—Mammina e Papino, Mamon, la Nonenga, my brothers Amine and Yass, and my sisters-in-law Salva and Dani. Your love, support, and constant encouragement have been my anchor. A special thanks to Billy for our countless hours of phone calls discussing about "chicken milk".

Finally, and most importantly, to *Marta Rotari*—my rock, my unwavering support, and the one who believed in me even when I struggled to believe in myself. Your love, patience, and encouragement have been the foundation of my strength throughout this journey. Thank you for always pushing me to grow, for lifting me up when I needed it most, and for being by my side every step of the way. I am endlessly grateful for your presence in my life, and this achievement is as much yours as it is mine.

Oh, and of course, I cannot forget my cat, *Brioche*, who, in her infinite wisdom, inspired the Callgraph Analysis Tool (CAT). I dedicate this tool to you, dear *Brioche*, for your indispensable contributions of napping and silent companionship.

I would also like to thank the *Alice and Knut Wallenberg Foundation* for funding my research through the Wallenberg AI, Autonomous Systems and Software Program (WASP).

To all of you, from the bottom of my heart, thank you.

Thank you very much!

Tack så mycket!

Tusinde tak!

Grazie mille!

Contents

Introduction

1 Introduction

Over the past few decades, software has become increasingly important in all systems. As our dependence on software grows, so do the risks associated with bugs, which can lead to significant financial losses and even loss of life. Well-known examples of software bugs include the *Therac-25* radiation therapy machine, which caused six patient deaths due to radiation overdoses [LT93], the *Mars Climate Orbiter* crash [Saw99]—resulting in a loss of 327 million dollars—and the *Toyota unintended acceleration* which was linked to at least 49 deaths [Kan+10].

While it may seem intuitive to run the program directly to verify its functionality, this approach overlooks the deeper issues at play in large-scale, safety-critical systems. Testing alone is often insufficient to uncover subtle or rare bugs that might only manifest under specific circumstances, such as unusual inputs or environmental conditions. For instance, in the case of the *Therac-25*, multiple software failures occurred due to complex interactions between different system components, which were not anticipated through routine testing [LT93]. Similarly, with the *Toyota unintended acceleration* incidents, the flaw was not immediately apparent during standard testing protocols [Kan+10]. In both cases, static analysis could have identified the issues before they resulted in catastrophic consequences.

Static (program) analysis is a branch of computer science that aims to study the behavior and properties of computer programs without executing them. Static analysis is a key technique for ensuring software quality and reliability and is widely used in various applications such as safety [Cou+05; Bla+02], security [PKB21; Arz+14; Aye+08b; Say+22; FD12], and performance optimization [Aho+07; App04].

Static analyzers, the tools implementing these analyses, automatically examine source code to identify potential issues such as bugs, code smells, or security vulnerabilities. Due to the complexity and time required by these analyses, they are often executed offline as part of the continuous integration pipeline or run overnight on the entire repository. While this approach has proven to be effective, it has practical limitations. In particular, developers must wait for the analysis results before receiving feedback on their code changes, and if the static analyzer identifies issues, the developer must then return to the code editor to address them, potentially disrupting their workflow.

This process can significantly decrease developer productivity, especially if the developer has already shifted focus to another task while awaiting the analysis results. In such situations, the developer might lose the context of the code changes, making it more challenging to address the identified issues. Additionally, the need to repeat the analysis multiple times to ensure all issues are fully addressed can further reduce productivity.

The current capabilities of modern Integrated Development Environments (IDEs) have led to a growing interest in developing static analyses that operate concurrently with the developer's interactions, providing instant feedback on the code that is immediately visible to developers [Pis+22]. These analyses are generally limited to basic tasks such as syntax highlighting, code completion, type checking, and the detection of simple code smells, such as unused variables or unreachable code. These limitations are primarily due to the requirement that the analysis must be highly responsive, with response times in the IDE needing to be less than 0.1 seconds for users to perceive the analysis as responsive [Nie94]. More advanced analyses, such as dataflow analysis, however, are still primarily conducted offline. The complexity of dataflow analysis arises from its focus on the flow of data and information through a program. This technique is essential for identifying more challenging sources of errors, such as API protocol violations, which ensure that software components interact in the correct sequence, race conditions, and taint analysis, which tracks the flow of sensitive information through a program. In addition to improving reliability, such analyses can also enhance program performance by uncovering opportunities for parallelization and other forms of optimization [Aho+07].

Traditionally, dataflow analysis has been implemented using imperative paradigms, which are based on the idea of explicitly specifying *how* the analysis should be performed. Recently, there has been a growing interest in using declarative paradigms for dataflow analysis [SEV16; DR+11; DR24], which are based on specifying *what* the analysis should compute rather than *how*. The declarative approach leads to a higher-level specification, resulting in improved modularity as the emphasis is on the desired outcome, rather than the specific steps required to achieve it.

In this work, we use *Reference Attribute Grammars* [Hed00] (RAGs) as a declarative approach for implementing dataflow analysis. RAGs are a powerful and flexible formalism for specifying the abstract syntax and semantics of languages, and as such, they are widely used in the development of compilers and static analysis tools. Our implementation is primarily based on the JAStAdd [HM01] RAG system. JastAdd supports RAGs and implements recursive demand-driven evaluation algorithms, ensuring that properties are evaluated only when necessary, reducing the overall evaluation time by avoiding redundant computations. Additionally, JastAdd supports circular attributes, which are crucial for static analysis problems that involve recursive dependencies. These attributes rely on fixed-point computations, which are challenging [Far86; JS86] to implement efficiently in a demand-driven evaluation framework.

The primary goal of this thesis is to explore the implementation of static analysis frameworks using Reference Attribute Grammars. We aim to use the declarative nature of RAGs and their demand-driven evaluation to develop static analyses that can be executed directly within IDEs. One of our main objective is to achieve highly responsive analyses, with execution times under 0.1 seconds. Our attention is mainly directed towards intraprocedural dataflow analysis, which involves examining the behavior of a method or function in isolation (i.e., without considering the interactions with other methods or functions).

In this thesis, we present four complementary contributions that collectively advance the implementation of static analysis frameworks using RAGs. These contributions aim to enhance the performance, precision, and applicability of static analysis techniques, particularly in the context of intraprocedural dataflow analysis.

In Paper 1, we present IntraCFG, a language-agnostic framework that significantly improves the efficiency and precision of control-flow graph construction in static analysis. Our evaluations show that IntraJ, an instance of IntraCFG specifically designed for the Java programming language and built upon the ExtendJ [EH07b] compiler, outperforms the industrial tool SonarQube in both efficiency and precision, achieving a response time of less than 0.1 seconds for intraprocedural analyses.

In Paper 2, we provide a detailed description of IntraJ, including its architecture and implementation, as well as its practical applications as a standalone tool for full program analysis and for performing on-demand analysis of files currently visible in the user's editor.

In Paper 3, we present JFeature, a static analysis tool for automatically extracting features from a Java codebase. JFeature enables researchers and developers to explore various characteristics of a codebase, including the usage of different Java features e.g., lambda expressions, across various Java versions, simplifying the identification of suitable corpora for evaluating their tools and methodologies.

Finally, in Paper 4, we introduce the RelaxedStacked algorithm, a novel demand-driven evaluation strategy for circular attributes in RAGs. This algorithm addresses inefficiencies in attribute computation within the RAG framework, reducing redundant evaluations of circular attributes and leading to significant performance improvements.

The remainder of this thesis is organized as follows. In Section 2, we provide background information on program analysis and attribute grammars to ensure that the discussion is as self-contained as possible, providing the necessary context to understand the contributions of this work. In Section 3, we present IntraCFG along with its Java implementation, IntraJ, as detailed in Paper 1. Section 4 introduces IntraTeal, an implementation of IntraCFG used to teach program analysis concepts. Section 5 discusses practical uses of IntraJ and IntraTeal in integrated development environments. More details about the integration of IntraJ into IDEs are covered in Paper 2. In Section 6, we present JFeature, a tool for extracting features from Java codebases (Paper 3). Section 7 focuses on the RelaxedStacked algorithm, a novel demand-driven evaluation strategy for circular attributes in RAGs, which is presented in Paper 4. Throughout these sections, we discuss the key challenges in the field and the corresponding solutions proposed by this work. Finally, Section 8 concludes the thesis by summarizing the contributions and outlining potential directions for future research.

2 Background

This section provides an overview of the concepts that underlie the main results of this thesis, with a focus on *program analysis*. Specifically, we briefly discuss the existing techniques behind control-flow graph construction [Aho+07; NNH10] and dataflow analysis [Kil73], as these are the analyses we focused on in this thesis. We will also give a general overview of the declarative approach used to implement these analyses, namely *(Reference) attribute grammars* [Knu68; Hed00], and their implementation through the JastAdd metacompiler. The dependency graph in Figure 1 shows the relationship between the concepts and the contributions of this thesis.

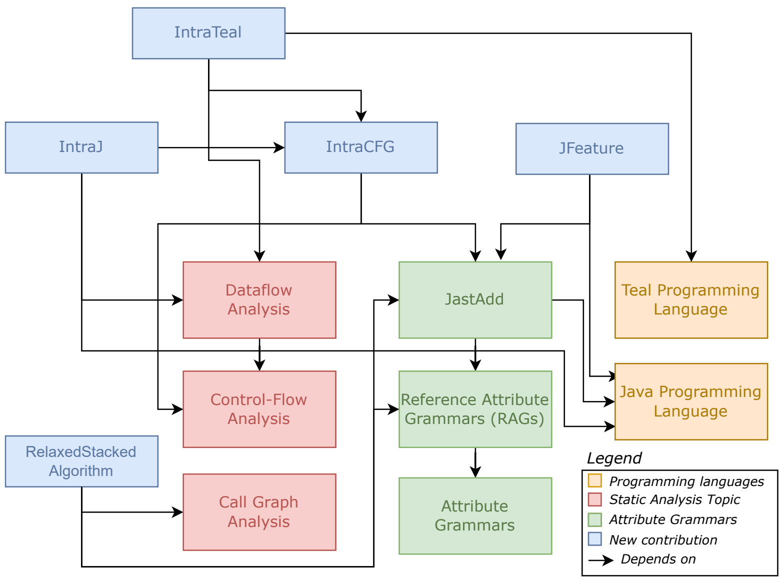


Figure 1: Dependency graph of the background concepts and the contributions of this thesis.

2.1 Automatic Program Analysis

Automatic program analysis is a key area in computer science, aiming to automatically examine and assess the properties of programs, such as *correctness*, *liveness*, and *safety*. Program analysis can be classified into two main approaches: *static* analysis and *dynamic* analysis.

Dynamic analysis involves evaluating a program's behavior by executing it. This approach gathers precise information from a specific execution of the program, which can then be used to infer properties about its behavior. Dynamic analysis is particularly effective in identifying runtime errors, such as memory leaks [Lab03], performance bottlenecks [Int], and security vulnerabilities [LZZ18]. However, this method has limitations, primarily due to its dependence on complete and accurate input data for each execution.

In contrast, static analysis examines programs without executing them, relying solely on their source code. This thesis focuses on two fundamental static analysis techniques: intraprocedural control-flow graph (CFG) construction and dataflow analysis.

Intraprocedural control-flow graph construction determines the order in which statements and expressions within a single method are executed. It provides a finite representation of all the possible paths within a method without taking into account interactions with other method or function calls. Building on this, intraprocedural dataflow analysis uses the control-flow information to deduce how data flows within the same method, enabling the identification of potential bugs or vulnerabilities.

Traditionally, static analyzers have been implemented using imperative approaches [LA04; VR+10], where control flow and data manipulation are explicitly handled through sequences of instructions, offering fine-grained control but often resulting in complex and less maintainable implementations. In contrast, alternative strategies, such as the use of Datalog [DRS21], functional programming approaches [MYL16b], or ad-hoc implementations designed for specific problems, provide different trade-offs between expressiveness and efficiency. In this work, we adopt *Reference Attribute Grammars* (RAGs) [Hed00], a declarative and modular approach. By using RAGs, we benefit from high-level abstractions, modularity, and on-demand evaluation, which contribute to a more efficient and maintainable implementation.

2.2 Precision in Static Analysis

Precision is a critical factor in static analysis, reflecting the accuracy and granularity of information about a program's properties. Traditionally, it has been assumed that achieving higher precision comes with trade-offs, particularly in terms of performance and scalability. As a general principle, greater precision has often been associated with increased computational resource demands, leading to slower execution times. On the other hand, lower precision can improve performance but at the cost of producing less accurate or overly general results.

However, recent insights challenge this traditional view. In particular, Erik Bodden's work in *The Secret Sauce* [Bod18] highlights that imprecision caused by overapproximation can, in many cases, slow down the analysis rather than improve it. While overapproximating may initially seem to reduce computational complexity, it often leads to excessive false positives and the need to handle a larger set of spurious information, ultimately making the analysis less efficient. As Bodden suggests, the belief that precision always comes at the cost of performance is no longer as straightforward as once thought. In fact, in the long run, more precise analyses may outperform less precise ones by reducing the unnecessary work caused by imprecision. Balancing precision and performance in static analysis requires careful consideration of how overapproximation affects long-term efficiency. In simpler terms, the *precision* of a static analyzer refers to its ability to minimize false positives, i.e., incorrectly identifying non-existent bugs or vulnerabilities. In contrast, *recall* measures the ability to detect all relevant issues, minimizing false negatives. Achieving a balance between precision and recall often involves trade-offs: increasing precision can reduce recall, potentially missing some bugs, while improving recall may introduce more false positives, thereby lowering precision.

Despite efforts to balance precision and recall, no static analysis can guarantee both soundness and completeness [Ric53]. In fact, our approach does not aim to be either sound or complete but rather *practical*, focusing on usability and effectiveness in real-world scenarios. This inherent limitation means that our analyses, while effective, may still fail to identify all bugs, leading to potential false positives or negatives [Liv+15].

Nevertheless, we chose to improve the precision of our analysis by focusing on two specific aspects of the Java language: control flow and exceptions. While control-flow analysis has been extensively studied [MS18], exception handling has not received the same attention in this context. By implementing exception-sensitive analyses, we aimed to address this gap. Control-sensitivity allows the analysis to differentiate between true and false branches in conditional statements, such as `if`-statements, thereby improving precision by considering the side effects of conditional expressions. Similarly, exception-sensitivity enables the analysis to track both checked and unchecked exceptions thrown and caught within the program, enhancing precision by identifying potential bugs related to exception handling.

2.3 Control-flow Graph Construction

Control-flow graph construction refers to the computation of the execution and evaluation order of the program's statements and expressions. Each possible execution order of a program is called a control-flow path. The result of the control-flow analysis is a control-flow graph (CFG) $G = (V,E)$. Each vertex $v \in V$ represents a unit of execution, e.g., a single statement or expression, or a basic block (a sequence of statements without labels and jumps). Each edge $(v_1,v_2) \in E$ represents a control-flow edge, indicating that the execution of v_1 may be directly followed by the execution of v_2.

We can distinguish two main approaches to constructing the CFG for a program: at the source level and the intermediate representation (IR) level. The

```
void foo(boolean b){
  Integer x = 0;
  if (b) {
    x = 1;
  } else {
    x = null;
  }
}
```

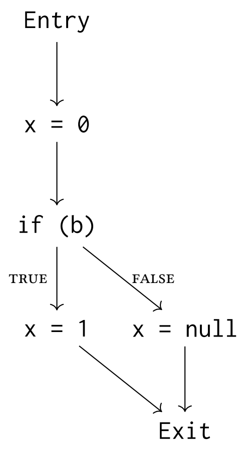


Figure 2: Source level control-flow graph of the `foo` method, showing the branching behavior of the `if`-statement.

source-level approach involves analyzing the source code directly and constructing the CFG based on the abstract syntax tree (AST). The IR approach, on the other hand, first converts the source code into an intermediate representation, such as bytecode, and then constructs the CFG from the IR.

Both approaches have their advantages and drawbacks. Constructing the CFG at the source level allows analysis results to be mapped directly back to the source code, making it easier to present findings in the context of the original program. In contrast, constructing the CFG at the IR level requires a translation step to relate the analysis results to the source code, which may not always be feasible, as seen with Java's `source-file retention` policy for annotations where information is lost during the process. For this reason, constructing the CFG at the source level is particularly valuable for tasks such as debugging and program understanding, as it provides a clear and direct representation of the program. Additionally, this approach can enable faster and more efficient analysis by eliminating the overhead of IR generation, while also being capable of handling semantically and syntactically invalid code, making it suitable for analyzing incomplete programs. Most importantly, constructing the CFG at the source level is particularly advantageous in interactive scenarios, such as analysis within an IDE, where the overhead of IR generation may introduce unacceptable latency.

However, there are also disadvantages to constructing the CFG at the source level. One significant limitation is the difficulty in accurately capturing the control flow of a program due to unsugared constructs, such as macros and preprocessor directives, which can complicate the specification of the analysis. Additionally, the source-level approach requires significantly more engineering effort compared to the IR approach. IRs are typically more compact than source lan-

```
1 : iconst_0
2 : invokestatic #7
3 : astore_2
4 : iload_1
5 : ifeq 17
6 : iconst_1
7 : invokestatic #7
8 : astore_2
9 : goto 19
10: aconst_null
11: astore_2
12: return
```

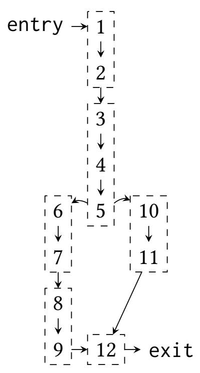


Figure 3: Bytecode control-flow graph of the `foo` method. Each dashed box represents a basic block.

guages, which reduces the number of language constructs that need to be handled, thus simplifying CFG construction. Moreover, since an IR can be targeted by multiple languages, it helps solve the NxM problem by providing a common representation. In general, the diversity of programming languages, each with unique syntax and semantics, makes it challenging to design a single analysis that can be applied universally across different languages.

To illustrate the differences between the source-level and IR-based approaches discussed, consider the examples in Figures 2 and 3, which show the control-flow graphs for a simple method `foo` at both the source level and bytecode level. As can be seen, the source-level CFG provides a clear and direct representation of the program's control flow, making it easier to understand and interpret the program's behavior. In contrast, the bytecode CFG is more abstract and difficult to interpret, as it lacks the high-level constructs present in the source code. We can also notice that both graphs have been augmented with *Entry* and *Exit* nodes, which represent the unique entry and exit points of the method, respectively. These nodes are crucial for simplifying the structure of dataflow analyses. For example, the *Entry* node provides a clear starting point for the analysis, ensuring proper initialization of parameters and variables at the beginning of the method. Similarly, the *Exit* node becomes important in backward dataflow analyses, which we will discuss in the next section.

2.4 Dataflow Analysis

Dataflow analysis is a technique used to analyse the flow of data through a program. It has its roots in the field of program optimisation [Kil73], where it was initially used to identify opportunities for improving the performance of programs by tracking variable definitions and uses. This information can be used to optimise the program by eliminating unnecessary computations (e.g., Very Busy Expression or Available Expression analyses [Aho+07]) and improving the use of available resources (e.g., registers optimisation).

In the context of bug detection, dataflow analysis can be used to identify potential sources of errors in a program by tracking the flow of data through the program and identifying points where data may be used in unexpected or incorrect ways. This can be particularly useful in identifying bugs that may not be immediately apparent, such as those that only occur under certain conditions or when certain combinations of input data are used (e.g., `IndexOutOfBound` exception). Many static analysis tools for Java programs, e.g., FindBugs [Aye+08a], employ intraprocedural dataflow analysis to identify potential bugs in Java code. Dataflow analysis, particularly *interprocedural* dataflow analysis, is widely used to identify potential security vulnerabilities in software [Arz+14]. For example, an interprocedural control-flow graph enables tracking of the flow across multiple methods or functions, thereby allowing identification of points where the data may be exposed to unauthorized access or manipulation.

We will demonstrate an application of intraprocedural dataflow analysis by presenting the following practical, but incomplete, example. Let us reconsider the `foo` method introduced in Figure 2. Our goal is to determine at each stage of the program whether the variable `x` has a `null` value or not[1].

At the entry point of the method, i.e., Entry node, it is indeterminate whether `x` is `null` or not, as it has not been initialized yet. However, at the declaration of the variable `x`, we can determine that it is not `null` because it is initialised to a non-`null` value. Then, if the condition `if(b)` is `true`, the variable `x` is assigned a new value, which is not `null`. If the condition is `false`, the variable `x` is assigned `null`. Therefore, at the end of the method, the variable `x` may be either `null` or not `null`. Consider the scenario where `x` is used and dereferenced immediately after the if-else statement, for example, calling a method on `x`. The program will then crash, with a `NullPointerException`, if `x` is `null`. Dataflow information can be used to identify potential bugs like this in a program.

We keep track of the value of `x` by mapping it to a finite set of possible values: `null`, `notnull`, `maybenull`, or `unknown`. As we traverse the control-flow graph, we propagate this information from node n to node n' if $(n,n') \in E$, until it reaches the Exit node.

The information is updated at each node n according to the following rules:

- If n is an assignment node, the information is updated according to the assignment operation. For example, if the assignment is `x = null`, then it is recorded that `x` is updated to `null`. If the assignment is `x = y`, then `x` is mapped to the value `y` maps to.

[1]For simplicity, just for this example, we assume that the language allows only assignments of the form `x = y` where `y` can be either a variable, a numeric constant or the `null` literal. We also assume that CFG nodes are individual assignments.

- If n is not an assignment node and has a single predecessor, no information is updated.
- If n has multiple predecessors, the information from the predecessors is merged conservatively. Specifically, if `x` is marked as both `null` and `notnull` by different predecessors, it is conservatively marked as `maybenull`.

The dataflow analysis just described is an instance of the mathematical concept of Monotone Frameworks [NNH10].

Monotone Frameworks

Monotone frameworks are a theoretical approach for reasoning about program dataflow properties. This approach provides a flexible and generic framework for expressing and solving dataflow equations, which can be used to reason about a wide range of dataflow properties, such as *live variables*, *reaching definitions* and *available expressions* analyses. Monotone frameworks are built on the concept of lattices [Don68].

A lattice $\mathcal{L} = (S, \leq)$ is a partially ordered set in which any two elements have a unique least upper bound (also known as a join or a supremum) and a unique greatest lower bound (also known as a meet or an infimum). This means that, for any elements a and b in S, there exists a unique element denoted as $a \sqcup b$ (or $a \vee b$) such that $a \leq a \sqcup b$ and $b \leq a \sqcup b$, and $a \sqcap b$ (or $a \wedge b$) such that $a \sqcap b \leq a$ and $a \sqcap b \leq b$. A complete lattice has a unique least element, commonly denoted as $\bot$, and a unique greatest element commonly denoted as $\top$. These elements satisfy the properties that for any element x in the lattice, $\bot \leq x$ and $x \leq \top$. In dataflow analysis, lattices are widely used to represent the information flow in a program. A common example of a lattice used in dataflow analysis is the *binary lattice* with elements *true* and *false*, which is used to represent the presence or absence of a property. Another example is the *interval lattice*, which is used to represent ranges of numbers. This lattice, compared to the binary lattice, is more complex but provides more precise information about the flow of numerical values in a program. Additionally, while the binary lattice is *finite* height, the interval lattice can be potentially *infinite*.

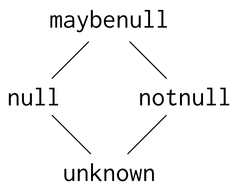


Figure 4: Diagram of the partial order in the example in Section 2.4, showing the order relation between `maybenull` ($\top$), `unknown` ($\bot$), `null`, and `notnull`.

Monotone frameworks include a join operator $\sqcup$, a monotone transfer function f, and a finite height lattice $\mathcal{L}$.

A monotone transfer function is a mathematical function that maps an element of a partially ordered set to another element in the same set such that the partial ordering is preserved under the function. More formally, if $(S, \leq)$ is a partially ordered set and $f : S \to S$ is a function, f is called a monotone transfer function iff, for all $x, y \in S$ such that $x \leq y$, it follows that $f(x) \leq f(y)$.

In our example, the lattice $\mathcal{L}$ is used to model the possible values, i.e., `null`, `notnull`, `maybenull`, and `unknown`, that a variable can assume. This lattice is shown in Figure 4. Here `maybenull` is the greatest element in the lattice, as it represents the set of all possible values that a variable can take. On the other hand, `unknown` is the least element representing the absence of information. The join operator $\sqcup$ merges the information of two nodes. For instance, if node n has information `null` and node n' has information `notnull`, then the information at the merged node $n \bigsqcup n'$ becomes `maybenull`. In the previous section, we explained how a node affects the flow of data using natural language. Now, we will present this concept in a formal manner as a transfer function. Let *Var* be the set of variables in the program, V be the set of nodes in the CFG, and $\Gamma : \mathit{Var} \to \mathcal{L}$ be the function that maps variables to elements of the lattice $\mathcal{L}$. The monotone transfer function $f_{\mathit{NULL}} : (\mathit{Var} \to \mathcal{L}) \times V \to (\mathit{Var} \to \mathcal{L})$ is defined as follows:

$$f_{\mathit{NULL}}(\Gamma, \mathsf{node}) = \begin{cases} \Gamma[\mathsf{v} \mapsto [\![\mathsf{e}]\!]^{\Gamma}] & \text{if } \mathsf{node} \text{ is } \mathsf{v = e} \\ \Gamma & \text{otherwise} \end{cases}$$

$$\text{where} \quad \begin{array}{lcl} [\![\mathsf{n}]\!]^{\Gamma} \text{ for } \mathsf{n} \in \mathit{Num} & = & \textbf{notnull} \\ [\![\mathsf{null}]\!]^{\Gamma} & = & \textbf{null} \\ [\![\mathsf{v}]\!]^{\Gamma} \text{ for } \mathsf{v} \in \mathit{Var} & = & \Gamma(\mathsf{v}) \end{array}$$

To propagate information from node n to its succeeding nodes (in the CFG) and to represent the effect of passing through a node, we define the following two equations:

$$\text{in}(n) = \begin{cases} \{v \to \bot \mid \forall v \in \mathit{Var}\} & \text{if } n \text{ is Entry} \\ \bigsqcup\limits_{p \in \text{pred}(n)} \text{out}(p) & \text{otherwise} \end{cases}$$

$$\text{out}(n) = f_{\mathit{NULL}}(\text{in}(n), n)$$

where $\text{pred}(n)$ is the set of predecessors of the node n, i.e., $\text{pred}(n) = \{p \mid (p,n) \in E\}$ and E is the set of edges in the CFG. We have defined the in and the out sets to model the information that is available before and after passing through a node, respectively. The in set gathers the available information before entering the node, while the out set captures the effect of applying the transfer function f_{NULL} on the in set. This kind of analysis is called a *forward analysis* because it propagates information from the entry node to the exit node.

These equations can be adapted to perform backward analyses, i.e., analyses that propagate information from the exit node to the entry node. Let us look at the general definition of a backward analysis. Given a monotonic transfer function f and a finite lattice $\mathcal{L}$, the equations for a backward analysis is defined as follow:

$$\text{out}(n) = \bigsqcup_{s \in \text{succ}(n)} \text{in}(s)$$
$$\text{in}(n) = f(\text{out}(n), n)$$

where succ(n) is the set of successors of the node n, i.e., succ$(n) = \{s \mid (n,s) \in E\}$. The boundary condition for the *out* set when n is the Exit node differs depending on the analysis.

The equations for both forward and backward analyses define a mutual dependency between the in and out sets of a node. A fixed-point computation is used to resolve circular dependencies in program analysis. Mathematically, a fixed point is an element x of a function f such that $f(x) = x$. In the context of static analysis, this refers to the stable state where further iterations of the analysis yield no changes, indicating that the analysis has converged.

Several conditions must be met for a fixed point to exist and be unique. According to the Knaster-Tarski theorem [Tar55], if the transfer function f is monotonic and operates over a complete lattice, a fixed point is guaranteed to exist. Monotonicity ensures that for any two elements x and y, if $x \leq y$, then $f(x) \leq f(y)$. The complete lattice structure provides a well-defined least and greatest element, which allows fixed-point computations to converge to the least fixed point. This least fixed point is both unique and minimal among all possible fixed points, ensuring that the analysis reaches the most conservative and sound result.

This result is essential not only for dataflow analysis, but also for the evaluation of circular attributes in Reference Attribute Grammars (RAGs) [Hed00].

2.5 Attribute Grammars

Attribute grammars [Knu68] (AGs) are a formalism for specifying the syntax and semantics of programming languages. This formalism is based on the concept of attributes, which are properties associated with the elements of a language's abstract syntax tree. Attribute grammars provide a powerful tool for specifying the behavior of a programming language and for verifying compile-time correctness of programs written in that language. AGs are composed of three components: a context-free grammar, which defines the language's syntax, a set of attributes, which are properties associated with the nodes of the abstract syntax tree, and attribute equations, used to compute the values of the attributes associated with each node in the abstract syntax tree.

Attribute grammars enable the description of the interdependence of syntactic and semantic elements of a programming language. For instance, the type of a variable may be determined by its declaration, but the type of an expression may be determined by the types of its sub-expressions. Attribute grammars provide a way to specify these rules.

We can distinguish two types of attributes: synthesized attributes and inherited attributes. For the sake of readability, we adopt the commonly used notation in Attribute Grammars, where attribute names are preceded by a symbol (e.g., $\uparrow$, $\downarrow$) to indicate the type of attribute, such as synthesized or inherited.

A *synthesized* attribute is a property of a node that is computed based on the attributes of the subtree rooted at that node. For example, the type of an expression in a programming language may be a synthesized attribute computed based on the types of sub-expressions in the expression. For example, in Java, the type of the expression "40 + 2" is determined to be an integer based on the types of its operands, whereas the type of the expression ""John" + 117" is determined to be a string due to the operands involved.

Synthesized attributes are composed by a declaration and an equation:

$$\begin{aligned} &\mathsf{T}\ \mathsf{A}.\!\uparrow\!\mathsf{x} \\ &\mathsf{A}.\!\uparrow\!\mathsf{x} = e \end{aligned}$$

where T is the type of the attribute, A is the node type, and x is the attribute name. The up-arrow symbol ($\uparrow$) is used to denote a synthesized attribute. The right-hand side of the equation is an expression, e, that may use other attributes of the A node or its children. If A has subtypes, say A1<:A and A2 <:A, different equations can be given for A2 and A1. If all subtypes will use the same equation, the declaration and the equation can be combined into a single line, as follows:

$$\mathsf{T}\ \mathsf{A}.\!\uparrow\!\mathsf{x} = e$$

An *inherited* attribute is a property of a node that gets its value from its parent in the abstract syntax tree. An example of inherited attribute is the expected type of an expression. The expected type of an expression is inherited from the context in which the expression is used. For example, the expected type of the condition in an `if`-statement is boolean. This information is inherited from the `if` statement node.

Inherited attributes are defined in two parts: a declaration and an equation.

$$\begin{aligned} &\mathsf{T}\ \mathsf{B}.\!\downarrow\!\mathsf{x} \\ &\mathsf{A}.\mathsf{B}.\!\downarrow\!\mathsf{x} = e \end{aligned}$$

where A and B are node types and the down-arrow symbol ($\downarrow$) is used to denote an inherited attribute. The equation defines the x attribute of A's child B. The expression e may use the attributes of A and any of its children.

2.6 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) were introduced by Hedin in [Hed00] and are an extension of AGs to Object-Oriented languages. While attributes in AGs can only refer to terminal values, RAGs allow attributes to refer to non-terminals, i.e., nodes in the AST. RAGs are well-suited for the analysis of programming languages since they enable the definition of relations between AST nodes. Attributes referring to AST nodes can declaratively construct relations, i.e., graphs, on the AST. Examples of the relations that can be constructed using RAGs are:

- Name and Type analysis [FH16; EH07a; FH15]: establishes a mapping between variable declarations and their corresponding uses, ensuring that each variable reference is resolved to its correct declaration. In addition to resolving names, type analysis ensures that expressions and variables conform to the correct types based on the program's type system.
- Control flow graph [Söd+13b; Rio+21]: a graph where nodes are statements or expressions, and edges are control flow relations, and,
- Call graph (see Paper 4): a graph where nodes are methods and edges are method calls.

RAGs are implemented by systems such as SILVER [Van+10a] and KIAMA [SKV10]. In this thesis, we used the JASTADD metacompiler [HM01].

2.7 The JASTADD Metacompiler

The JASTADD metacompiler [HM01] is a Java-based system that generates executable Java code from an abstract grammar and a RAGs specification. For each non-terminal in the grammar, JASTADD creates a corresponding Java class, where each attribute defined in the RAGs specification is translated into a method within the respective class. These methods come with additional mechanisms, such as memoization and attribute location, to optimize attribute handling.

One of the key strengths of JASTADD is its support for on-demand attribute evaluation. Attributes are computed only when they are explicitly needed, allowing the analysis to trigger computations dynamically. This feature is particularly useful in scenarios where the computation of an attribute is not always necessary.

The JASTADD metacompiler consists of two main components:

- The JASTADD language: a domain-specific language for defining RAGs. It allows for the specification of abstract grammars and attribute equations using a Java-like syntax.
- The JASTADD compiler: a compiler that generates Java code from the RAG specifications, transforming the abstract grammar and attribute definitions into Java classes and methods.

JASTADD supports synthesized and inherited attributes, as well as other specialized kinds of attributes. The most important attributes relevant to this thesis are: *Parametrised Attributes* [Hed00], *Higher-Order Attributes* [VSK89b], *Circular Attributes* [Far86; JS86], and *Collection Attributes* [Boy96].

Parametrized Attributes are attributes whose value may depend on one or more parameters supplied to them. In this thesis, we use parametrized attributes to pass the context Γ, which carries the information required for dataflow analysis from one `CFGNode` to another. They are also commonly used in type systems to check whether two types are compatible.

Higher-Order Attributes (HOA): Also referred to as *Non-Terminal Attributes (NTA)*, higher-order attributes are attributes whose values are fresh AST subtrees. They are called *higher-order* because they are both attributes and non-terminal nodes, meaning they can themselves be attributed. HOAs are commonly used to represent information that is not explicitly present in the source code or the AST. For example, we use HOAs to reify a method's *entry* and *exit* points, which are essential for CFG construction. Throughout this thesis, we use the right-arrow symbol (→) to denote HOA attributes, such as `FunDecl`.→entry and `FunDecl`.→exit.

Collection Attributes: Unlike other attributes that are defined by equations, collection attributes gather their values through contributions. These contributions can originate from any point in the AST and are aggregated to form the final value of the collection attribute. A contribution clause, associated with an AST node type, specifies what data should be contributed to the collection attribute and under what conditions.

Collection attributes are particularly useful in scenarios where data needs to be collected from different parts of the AST. A common use case is in compiler construction, where all semantic errors in a program must be collected regardless of their location in the AST.

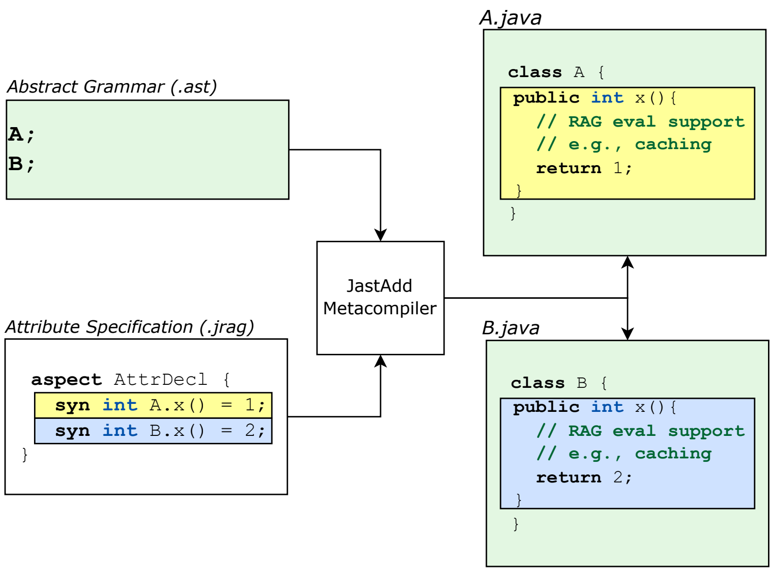
For example, the collection attribute `Program`.□errors, gathers all semantic errors in a program. Since multiple contribution clauses can be defined for a single collection attribute, new error types can be easily added by introducing additional clauses.

In this thesis, we use collection attributes to compute reverse relations in CFGs. For instance, the predecessors of a node (`CFGNode`.□pred) are computed as the reverse of its successors (`CFGNode`.↑succ).

Circular Attributes: Circular attributes are attributes whose definitions may depend on themselves, either directly or indirectly. In JastAdd, circular attributes are declared using the `circular` keyword and are evaluated through a fixed-point computation. This process starts with an initial value and continues until the attribute's value stabilizes, i.e., does not change between iterations.

To guarantee termination, certain conditions must be met, although JastAdd does not enforce these:

- The possible values of the attribute must form a lattice of finite height.



```
A;
B;
```

```
aspect AttrDecl {
  syn int A.x() = 1;
  syn int B.x() = 2;
}
```

```
class A {
public int x(){
  // RAG eval support
  // e.g., caching
  return 1;
}
}
```

```
class B {
public int x(){
  // RAG eval support
  // e.g., caching
  return 2;
}
}
```

Figure 5: JastAdd generates Java classes from the abstract grammar and weaves the code from the intertype declarations into the corresponding generated classes A and B. Additional RAG code for on-demand evaluation is omitted for clarity.

- The intermediate results of the fixed-point algorithm must increase or decrease monotonically[2].

We use the notation $\mathtt{A}.\circlearrowleft\uparrow x$ to denote a circular synthesized attribute. For example, consider the circular attribute $\mathtt{A}.\circlearrowleft\uparrow x$ defined as `Math.min(3, A.`$\circlearrowleft\uparrow x$` + 1)`. In this case, the attribute begins with an initial value (e.g., 0), and through fixed-point iteration, it stabilizes at 3 after three iterations.

Computing circular attributes both correctly and efficiently is challenging. Given their relevance to this thesis, as they impact not only the accuracy of the computed values but also the overall performance, we will explore the existing algorithm for circular attribute evaluation in detail in Section 2.8.

Another important feature of JastAdd is its support for AspectJ-style [Kic+97] intertype declarations, which allows attributes to be defined in a highly modular fashion. This approach enables attribute declarations and

[2]In this thesis, boolean circular attributes start as *false* and grow monotonically using $\vee$, while set-typed attributes start as the empty set and grow using $\cup$.

equations to be written in separate aspects, increasing modularity by decoupling the attribute logic from the core structure of the abstract grammar. As a result, JastAdd automatically generates the corresponding Java methods and weaves them into the appropriate classes derived from the abstract grammar, simplifying maintenance and extension of the code.

Figure 5 shows an intertype declaration for attributes. In this example, the attributes A.↑x and B.↑x are declared in the aspect `AttrDecl` and are woven into the respective classes `A` and `B`.

2.8 Circular Attributes

In the traditional formulation of Attribute Grammars by Donald Knuth [Knu68], a grammar with circular dependencies is considered ill-formed. However, the inability to define circular dependencies in Attribute Grammars presents a limitation, as many language properties are inherently recursive, and circular dependencies often provide a natural way to express such relationships.

This limitation was addressed independently and concurrently by Farrow [Far86] and Jones et al. [JS86]. Farrow demonstrated that circular attribute grammars can be well-defined under certain constraints by ensuring monotonicity and boundedness in attribute values. This allows the computation of least fixed points through successive approximation [Far86]. Jones et al., extended attribute grammars to support circular dependencies in the context of hierarchical VLSI design, where cycles are inherent in circuits. Their approach involved partitioning the dependency graph into strongly connected components and using fixed-point computations to ensure a correct evaluation of circular attributes [JS86].

Sasaki and Sassa [SS03] extended circular attribute grammars by introducing *Circular Remote Attribute Grammars*, which support both circular dependencies and non-local attribute references. This extension allows attributes to reference values from distant nodes in the syntax tree, overcoming the limitation of traditional AGs that restrict dependencies to parent-child relationships. Their approach introduced remote links, a restricted form of reference attributes, and described how to perform exhaustive circular evaluation over these links. However, these remote links must be set prior to evaluation, meaning they cannot be dynamically computed during the evaluation process.

Boyland [Boy96] expanded the understanding of circular attributes by introducing techniques for modular demand evaluation, which allows attributes to be evaluated only when needed. His approach ensures termination even in the presence of circular dependencies, by using a combination of demand-driven evaluation and monotonicity constraints. However, Boyland did not provide the actual algorithm for implementing this evaluation strategy.

Magnusson et al. [MH07c] proposed several algorithms to handle circular dependencies in Circular Reference Attributed Grammars. The most important algorithms are the BasicMonolithic and BasicStacked$_{\text{OLD}}$[3] algorithms. The BasicMonolithic algorithm computes attribute values using an iterative fixed-point process. All attributes involved in circular dependencies are initialized to a bottom value specified by the user and updated iteratively until no further changes occur, ensuring the computation of the least fixed point. However, this approach evaluates all attributes in a unified process, which can lead to inefficiencies when nested circular dependencies or complex interdependencies are present.

For example, consider the attribute system on the left side of Figure 6. In one cycle, attributes a_1, a_2, a_3 depend on each other, and they all, directly or indirectly, depend on the attribute b, which in turn depends on another cycle involving c_1 and c_2. Let us suppose evaluation starts in a_1. In the BasicMonolithic approach, the evaluation of these cycles would be handled in a single, monolithic component and all attributes involved in the cycle, i.e., $(a_1, a_2, a_3, b, c_1, c_2)$ are updated together in each iteration. This leads to inefficiencies because changes in a_1, a_2, a_3 may unnecessarily trigger re-evaluations of c_1, c_2 through their dependency on b, even though they form separate cycles. This inefficiency is illustrated by the computation tree on the right side of Figure 6.

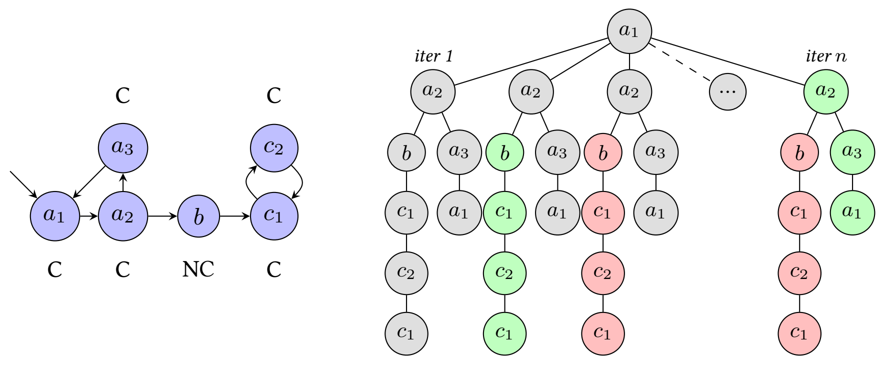


Figure 6: Example of two strongly connected components separated by the NonCircular attribute b. On the left, the attributes dependencies are shown. C denotes Circular and NC denotes NonCircular. On the right, the tree of recursive call is showed. The requested attribute a_1 will drive a fixed-point iteration, repeatedly evaluating the equation defining a_1 until all downstream attributes have converged. Here *grey* nodes represent attributes that have not yet converged, *green* nodes indicate attributes that have converged in that iteration, and *red* nodes are attributes that have converged but are still recomputed in subsequent iterations.

[3]In this thesis, the term BasicStacked$_{\text{OLD}}$ refers to the [MH07b] version of the algorithm, in contrast to the newer BasicStacked, which is discussed introduced later in this section.

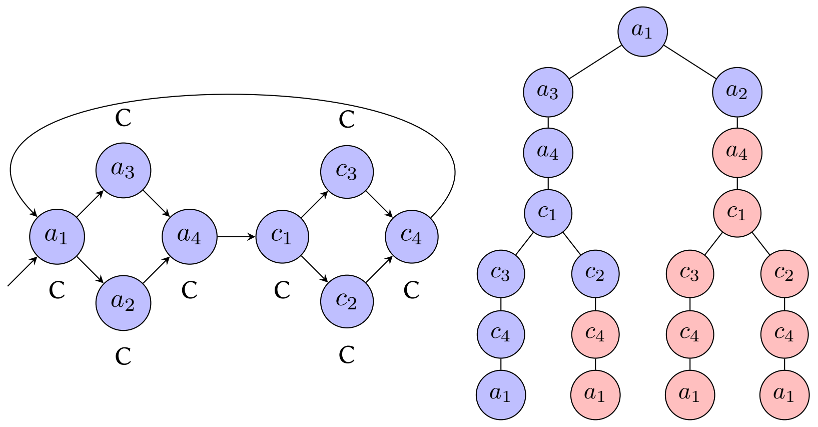


Figure 7: Example of a circular dependency that can lead to exponential attribute evaluation time. The left side shows the dependency graph between attributes, while the right side shows the order in which the attributes are evaluated. Red nodes indicate the redundant attribute evaluation within the same fixed-point iteration.

The BasicStacked$_{\mathrm{OLD}}$ algorithm optimizes this process by identifying distinct strongly connected components and evaluating them separately.

The BasicStacked$_{\mathrm{OLD}}$ applies several improvements to the BasicMonolithic approach. The major improvement is the introduction of a stack-based evaluation strategy that allows for the separation of strongly connected components (SCCs) in the dependency graph. Strongly connected components are identified and evaluated separately, if they are separated by a NonCircular attribute, which acts as a bridge between different cycles. When a NonCircular attribute triggers the evaluation of a circular attribute, the evaluation of any ongoing circular evaluation is suspended, allowing the start of a new circular evaluation. For instance, in the earlier example, where attributes a_1, a_2, a_3 depend on each other and on b, which in turn depends on c_1, c_2, c_3, BasicStacked$_{\mathrm{OLD}}$ would identify the two interdependent cycles separately as strongly connected components. The first SCC would contain a_1, a_2, a_3, while the second SCC would consist of c_1, c_2. The connecting node b would block any ongoing circular evaluation, allowing the start of new ones. When b calls c_1, the evaluation of the second SCC would begin as a new circular evaluation.

The BasicStacked$_{\mathrm{OLD}}$ algorithm, however, suffers from a major inefficiency, which can lead to exponential evaluation time in certain cases. The inefficiency arises when the same attribute can be reached along more than one path in the dependency graph. In each iteration, the attribute will then be evaluated once for each path, although one evaluation would have been sufficient. To illustrate the problem, consider the example shown in Figure 7. The left side of the figure displays the dependency graph between attributes, and the right side shows the evaluation tree of recursive calls for one iteration in the fixed-point computation.

When the attribute a_1 is requested, the fixed-point iteration starts, driven by the a_1 attribute. In each iteration, a_1 computes a new approximation of its value. To do so, it first calls a_3. This leads to a chain of recursive calls to a_4, c_1, c_3, and c_4. As shown on the right side of the figure, when the evaluation reaches c_4, it loops back to a_1. However, since a_1 is the driver of the fixed-point computation, the recursion stops, and the current value of a_1 is returned (the bottom value for the first iteration).

When control is returned to c_1, it continues by calling c_2, leading to a new call to c_4 and on to a_1, where the recursion is again stopped. A similar chain of redundant calls happens when a_2 calls a_4. However, the a_1 value has not been updated yet, so the second call to c_4 is redundant.

In the worst-case scenario, the same attribute may be recomputed 2^n times, where n is the number of possible paths leading back to the driver attribute.

During the years, the JastAdd team has proposed several improvements to the BasicStacked$_{\text{OLD}}$ algorithm, resulting in the BasicStacked algorithm. In the BasicStacked, we track the iteration during which each attribute was last evaluated in the fixed-point computation. This avoids redundant evaluations when an attribute's value is used multiple times in the same iteration, improving efficiency. This approach, furthermore, allows for the detection of attributes that are incorrectly classified as non-circular but are actually part of a cycle at runtime. In comparison, Magnusson's algorithm did not include this detection and could yield incorrect results in such cases. While Magnusson proposed a fix by tracking sets of attribute instances for each fixed-point component, it would have introduced significant performance overhead.

Nevertheless, the BasicStacked algorithm still has limitations. One significant challenge is that developers must manually identify circular dependencies, which can be prone to errors and difficult to manage, especially in large, continuously evolving codebases. In large projects, such as the ExtendJ Java compiler, even minor modifications to attribute definitions can unintentionally introduce new circular dependencies. This makes it hard to predict which attributes will require circular declarations. As a result, developers often over-annotate attributes as Circular, leading to unnecessary fixed-point computations and associated performance costs.

Öqvist [ÖH17; Öqv18] introduced a new algorithm, RelaxedMonolithic, which aims to overcome some of the limitations of the BasicStacked algorithm. One of its features is the introduction of a new type of attribute, called *Agnostic*. Agnostic attributes are only treated as Circular when they are part of a cycle, and otherwise, they are treated as NonCircular. This approach offers two main benefits. First, it simplifies the development process since the developer does not need to carefully classify every attribute as circular or non-circular. The only requirement is that at least one attribute in a cycle is declared Circular. Second,

1968 - Knuth [Knu68]: Traditional attribute grammars, do not support circular dependencies, considering them ill-formed.

1986 - Farrow [Far86] & Jones et al. [JS86]: Independently introduced circular but well-defined attribute grammars. Farrow's approach statically analyzes dependencies, while Jones's relies on a dynamic dependency graph to identify strongly connected components and supports incremental evaluation.

1996 - Boyland [Boy96]: Describes demand-driven evaluation for circular attributes in the presence of so-called remote attributes (similar to reference attributes), but gives no explicit evaluation algorithm.

2002 - Sasaki and Sassa [SS03]: Extend attribute grammars with remote links, a restricted form of reference attributes, and describe exhaustive circular evaluation over them.

2003 - Magnusson et al. [MH07b]: Proposed the first circular evaluation algorithm for RAGs, BasicMonolithic, which lacked SCC separation, resulting in inefficiencies. $\text{BasicStacked}_{\text{OLD}}$ addressed this by introducing SCC separation, though some cases still exhibited exponential evaluation time.

2003 - 2024 - JastAdd Team: Several improvements were implemented, e.g., fixing the exponential evaluation time issue to the $\text{BasicStacked}_{\text{OLD}}$ algorithm, resulting in the BasicStacked algorithm.

2017 - Öqvist et. al.[ÖH17]: Introduced RelaxedMonolithic with Agnostic attributes for flexible circular/non-circular classification. However, once circular evaluation starts, SCCs cannot be separated, leading to inefficiencies similar to BasicMonolithic.

2024 - Riouak et. al. (Paper 4): We presented a new algorithm, RelaxedStacked, which combines the strength of BasicStacked and RelaxedMonolithic using static analysis, allowing the coexistence of Circular, NonCircular and Agnostic attributes.

Figure 8: Timeline of circular attribute grammar developments, with key challenges and improvements over time.

if an Agnostic attribute is not part of a cycle, it is treated as NonCircular, avoiding the overhead of circular evaluations using the fixed-point evaluation strategy.

However, this comes with a trade-off. Once a circular evaluation starts, the algorithm cannot separate strongly connected components, leading to the same inefficiencies found in the BasicMonolithic approach. This results in the same issue described in Figure 6, where unrelated dependencies are evaluated together, increasing computational overhead.

Figure 8 shows a timeline of circular attribute grammar developments, with key challenges and improvements over time.

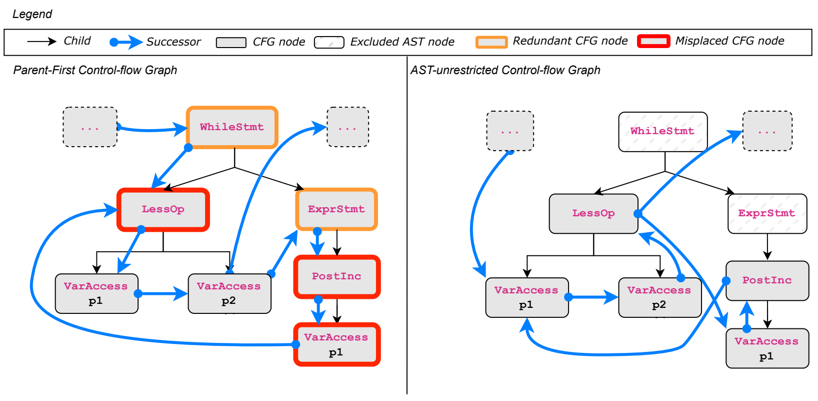


Figure 9: Example of a *Parent-First* (left) and *AST-Unrestricted* (right) CFGs.

3 IntraCFG: Intraprocedural Framework for Source-Level Control-Flow Analysis

The construction of control-flow graphs has advanced significantly in recent years, leading to the emergence of multiple frameworks that support the creation of precise intraprocedural CFGs [SWV20; Söd+13a]. However, despite these advancements, many existing frameworks either lack the efficiency required for practical, real-time analysis or do not provide the precision necessary for advanced, source-level analyses.

To address these challenges, we developed IntraCFG, a declarative, RAG-based, and language-independent framework designed to construct precise intraprocedural CFGs directly from the source code, avoiding the additional step of generating intermediate representations, e.g., bytecode or LLVM IR. By operating at the source level, IntraCFG allows developers to customize the inclusion or exclusion of relevant information, such as specific AST nodes, based on the needs of the application. This flexibility helps balance both performance and precision of the analysis.

3.1 Motivation and Challenges

The primary goal driving the development of IntraCFG was to enhance the flexibility and precision of intraprocedural analyses by constructing CFGs on the source-level that are not bound by the rigid structure of the AST. The previous RAG-based framework, JastAddJ-Intraflow [Söd+13b], generates CFGs that include all AST nodes in a parent-first order. This approach can result in imprecision, as the CFGs may not accurately reflect the true control flow of

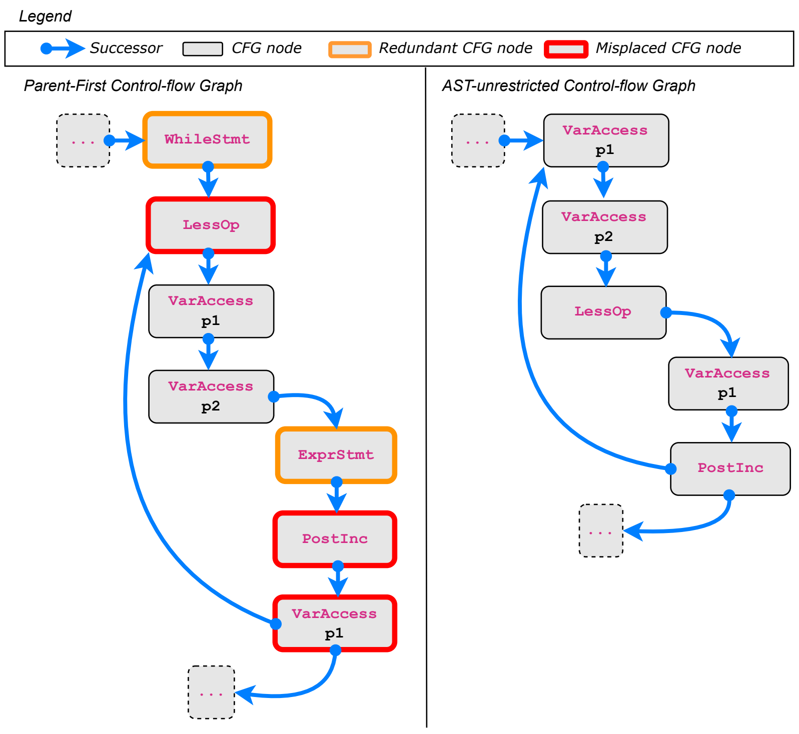


Figure 10: The CFGs shown in Figure 9 without AST structure.

the program, leading to inefficiencies and reduced accuracy, especially when users require more granular control over which nodes are included in the analysis. As an example, consider the incomplete Java code snippet in Listing 1. The JastAddJ-Intraflow framework would generate a CFG that includes all the nodes of the AST in a parent-first order, as shown on the left side of Figure 9. On the other hand, IntraCFG can generate a CFG that is not constrained to follow the structure of the AST (AST-Unrestricted), as shown on the right side of the figure. Instead, it includes only the relevant nodes for the control flow of the program, resulting in a more precise representation. Figure 10 shows the CFGs without the AST structure, highlighting the difference in precision between the two approaches.

Listing 1: Java code snippet.

```java
while(p1<p2){
  p1++;
}
```

In contrast to source-level CFG construction, many existing frameworks operate at the intermediate representation level. While IntraCFG can be applied to construct CFGs at the intermediate representation level, the primary focus of this thesis is on source-level CFG construction to avoid the overhead associated with IR generation and to preserve the source semantics.

3.2 The IntraCFG Framework

IntraCFG is a declarative, RAG-based, and language-agnostic framework that constructs AST-Unrestricted CFGs directly at the source code level. The framework addresses the limitations of existing frameworks by providing an interface that allows users to define the precision of the CFGs by selectively including or excluding specific AST nodes.

Unlike many other frameworks that build control-flow graphs at the intermediate representation level, such as bytecode, IntraCFG adopts an approach that superimposes the CFGs on the abstract syntax tree. This design choice offers several advantages, particularly in the context of Java programming.

The advantages and disadvantages of constructing CFGs at the AST level were discussed in Section 2.3. For convenience, we provide a summary of the main points here. One significant benefit of constructing CFGs directly on the source code level is the preservation of certain features that may be lost during compilation to bytecode. For example, annotations with source retention policies are retained, allowing for more precise client analysis and better alignment with the original source semantics. Additionally, the source-level CFGs can be more easily understood by developers and error reports can be more accurately mapped back to the source code, simplifying the debugging process. However, this approach is not without its challenges. The engineering effort required to implement CFGs at the AST level can be considerable. While desugaring the AST to a more precise representation is possible [OH13], it may require additional effort to ensure that the CFGs accurately reflect the control flow of the program.

IntraCFG consists of several key components, including interfaces, attribute equations that define the default behavior, and APIs. The interfaces provide the structure for the CFG, and the attribute equations define the default behavior for the CFG construction. In the language dependent control-flow module, implementations of the IntraCFG interfaces are added to the AST types of the language. This can be done according to the precision desired for the CFG.

IntraCFG has APIs in the form of attributes, allowing clients to access entry and exit nodes and to traverse the CFG. The language-independent nature of IntraCFG allows for easy integration with various programming languages and enables the construction of precise CFGs for those languages. The use of attribute equations and interfaces also allows for a high degree of flexibility in the CFG construction process, enabling the customization of the CFG to fit the specific needs of the analysis being performed.

IntraCFG provides three different interfaces: `CFGRoot`, `CFGNode`, and `CFGSupport`. These interfaces are implemented by AST types to construct the CFG. Each interface has a set of attributes, and default equations that are used to construct the CFGs.

The `CFGRoot` interface is intended to be implemented by AST nodes that represent subroutines, e.g., `MethodDecl` or `ConstructorDecl`. The `CFGRoot` interface defines two higher-order attributes (HOAs): `CFGRoot`.→entry and `CFGRoot`.→exit. These HOAs are used to represent the unique entry and unique exit points of the CFG.

The `CFGNode` is the most important interface in IntraCFG. The purpose of the `CFGNode` interface is to represent AST nodes that can be part of the CFG. This interface defines the synthesised attributes `CFGNode`.↑succ and `CFGNode`.↑pred, which are used to represent the sets of successors and predecessors of a CFG node, respectively.

Finally, the `CFGSupport` interface is implemented by all the AST nodes that may contain `CFGNode`s in their subtrees. Indeed, all `CFGNode`s are `CFGSupport` nodes, but `CFGSupport` nodes that are not `CFGNode`s can help steer the construction of the CFG.

To compute the `CFGNode`.↑succ attribute, the framework uses the helper attributes `CFGNode`.↑firstNodes and `CFGNode`.↓nextNodes. The `CFGNode`.↑firstNodes attribute of a `CFGNode` n contains the first `CFGNode` within or after the AST subtree rooted in n. The default definitions provided by IntraCFG for the `CFGNode`.↑firstNodes attribute are the empty set for a `CFGSupport` node and the node itself for a `CFGNode`.

The inherited attribute `CFGNode`.↓nextNodes is used to keep track of the `CFGNode`s that are outside the tree rooted in n, and that would immediately follow the last `CFGNode` within n. By default, the `CFGNode`.↑succ attribute is defined as equal to `CFGNode`.↓nextNodes.

To illustrate how these attributes are used in practice, consider the example of an `AddExpr` node with operands `Left` and `Right`, and where the operands are evaluated right-to-left. The following equations define how the ↑firstNodes and ↓nextNodes attributes are computed for the `AddExpr` and its operands:

$$
\begin{aligned}
\texttt{AddExpr}.\uparrow\text{firstNodes} &= \texttt{Right}.\uparrow\text{firstNodes} \\
\texttt{AddExpr.Right}.\downarrow\text{nextNodes} &= \texttt{Left}.\uparrow\text{firstNodes} \\
\texttt{AddExpr.Left}.\downarrow\text{nextNodes} &= \{\mathit{this}\} \text{// ``this'' refers to} \\
&\qquad \text{the AddExpr node.}
\end{aligned}
$$

Because of the default definitions provided by the framework, the ↑succ attribute for each node is automatically set to the corresponding ↓nextNodes value. Figure 11 illustrates this example applied to an abstract syntax tree. Both

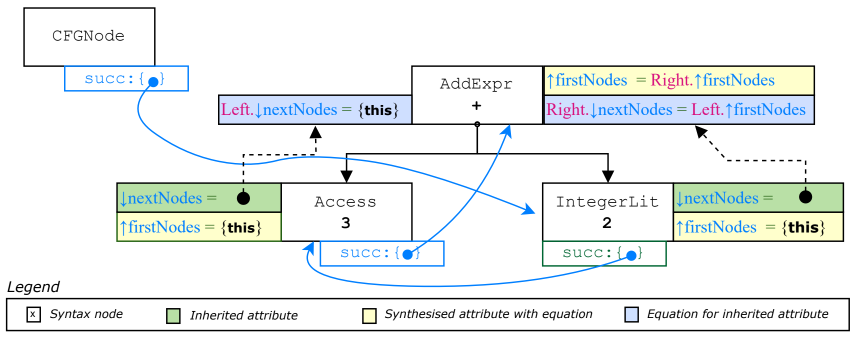


Figure 11: Example of definition of ↓nextNodes and ↑firstNodes for the Teal AddExpr using right-to-left operand evaluation. All the nodes are CFGNodes. Dashed arrows indicates the location of the equation for an inherited attribute.

the AddExpr and its operands implement the CFGNode interface. The ↑firstNodes attribute for an AddExpr is set to the ↑firstNodes of its right operand. The ↓nextNodes attribute for the right operand is defined as the ↑firstNodes of the left operand, while for the left operand, it is a singleton set containing the AddExpr node itself. Since the ↑succ attribute is not overridden, it defaults to the node's ↓nextNodes value.

To support both forward and backward analyses, the framework provides a predecessor attribute that captures the inverse of the successor attribute, i.e., CFGNode.↑succ. However, CFGNode.↑succ is also defined for CFGNodes that are not reachable from Entry by following CFGNode.↑succ (i.e., that are "dead code"). The framework computes predecessor edges CFGNode.↑pred by not only inverting CFGNode.↑succ into the collection attribute CFGNode.☐succInv, but also by filtering out "dead" nodes from CFGNode.☐succInv with a boolean circular attribute, CFGNode.↻↑reachable.

3.3 IntraJ: IntraCFG for Java

To demonstrate IntraCFG applicability, we developed IntraJ, an instance of the framework for the Java programming language. We built IntraJ upon the ExtendJ extensible Java compiler [EH07b].

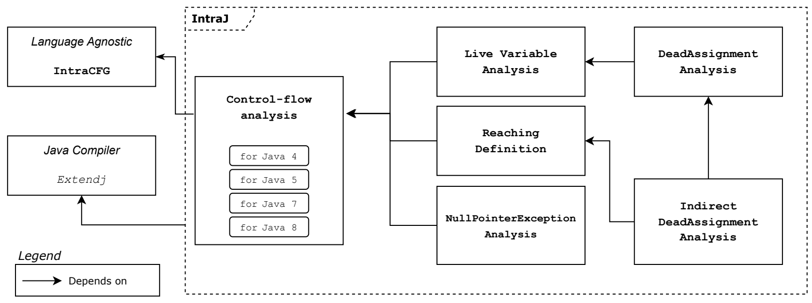


Figure 12: Overall architecture of instantiating INTRACFG for the Java language.

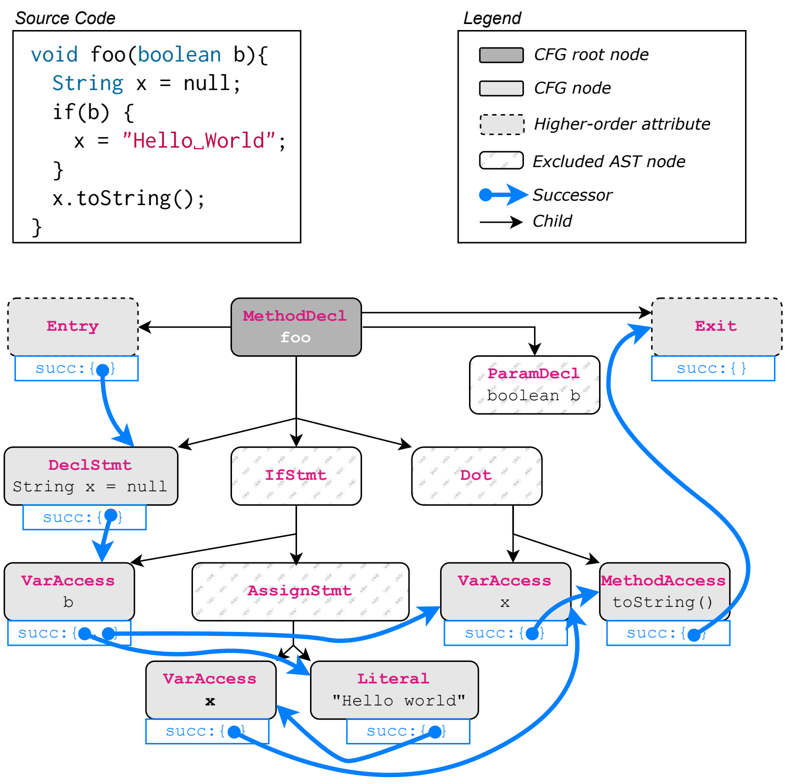
As seen in Figure 12, we designed INTRAJ following a modular approach, and we separately instantiated the framework for different versions of Java, such as Java 4, Java 5, Java 7 and Java 8. EXTENDJ has a similar modularisation of different Java versions. When EXTENDJ is extended to support new versions of Java, this approach will allow us to extend INTRAJ in a corresponding way. This approach allows us and the users of INTRAJ to easily extend the framework to support new versions of Java.

The degree of precision in creating CFGs using INTRACFG can differ in order to meet the requirements of a given application. This flexibility allows the framework users to optimise the analysis's efficiency by selectively excluding/including specific AST nodes from the CFG. For example, nodes such as `WhileStmt`, which are essential for the construction of CFGs, as they are used to define the shape of CFGs, can be excluded from the CFGs as all relevant execution points for the `WhileStmt` are already captured by other AST nodes, i.e., evaluation of the condition, and execution of the body.

The example in Figure 13 is a visual representation of the AST and CFG of the `foo` Java method. The figure illustrates the ability of the framework to tailor the CFG to the specific requirements of the analysis and eliminate unnecessary complexity for improved performance. In this example, nodes like `IfStmt` or `Dot` are not included in the CFG, resulting in a more concise but precise representation of the control flow of the program. On the other hand, the precision of the CFGs can be improved by synthesising new nodes and subtrees as HOAs. For instance, we designed INTRAJ to compute an exception-sensitive control-flow analysis, i.e., new AST subtrees are synthesized for each possible exceptional path. The resulting CFGs are more precise but also more complex, resulting in higher memory consumption and a more extended analysis time.

In INTRAJ, we implemented five different dataflow analyses:

- *Live Variable Analysis*: computes the set of variables that live at each point in the program.



```
void foo(boolean b){
  String x = null;
  if(b) {
    x = "Hello␣World";
  }
  x.toString();
}
```

Figure 13: AST with superimposed CFG for the `foo` Java method.

- *Reaching Definition*: computes the set of definitions that reach each point in the program.
- *Null Pointer Analysis*: detects possible null pointer dereferences.
- *Dead Assignment Analysis*: detects assignments to l-values (the left-hand side of an assignment) that are never used from that point on.
- *Indirect Dead Assignment Analysis*: detects assignments to l-values which uses always flow to a dead assignment.

All the analyses rely on the result of the control-flow analysis. The analyses use the `CFGRoot`.→entry and `CFGRoot`.→exit attributes of the CFG, as well as the `CFGNode`.↑succ and `CFGNode`.↑pred attributes of each node. Each analysis is implemented as a separate aspect. Still, some analyses' results are used as input for other analyses. For instance, the result of Live Variable Analysis is used as input for the Dead Assignment Analysis. Similarly, the result of Dead Assignment Analysis is used to compute Indirect Dead Assignment Analysis.

The implemented analyses are instances of the monotone frameworks (see Section 2.2). Each analysis defines its abstract domain, transfer function, and *in* and *out*[4] circular attributes for each `CFGNode` (e.g., `CFGNode`.↻↑in and `CFGNode`.↻↑out). While the core of each dataflow analysis is language-independent, relying solely on the attributes defined by IntraCFG, language dependencies arise in the transfer function, which is modeled as a parametrised attribute.

The transfer function is defined for each AST node in the CFG to capture the semantics of passing through that node. The specific implementation of the transfer function varies depending on the type of AST node.

For example, in the case of a `NullPointerException` analysis, the transfer function is initially defined in a language-independent manner, passing the context Γ, which carries information about variable states, unchanged through nodes that do not affect the analysis.

For an assignment statement (`AssignStmt`), the transfer function behaves differently. It retrieves the variable on the left-hand side of the assignment and checks whether the right-hand side is assigned a null value. Based on this check, the transfer function updates the context Γ to reflect the nullable status of the variable. The left-hand side variable is marked as *NULL* or *NOTNULL*, depending on whether the right-hand side is *NULL* or not.

In this example, the transfer function operates by updating the mapping of variables to their nullable status within the context Γ (see Section 2.4).

[4] Or *kill* and *gen*.

3.4 Performance and Precision

We evaluated the performance and precision of IntraJ using four well-known open-source Java projects: Antlr, Pmd, JFreeChart, and Fop. The selection of these projects was aimed at representing diverse types of projects, including libraries, frameworks, and applications, as well as varying sizes, ranging from 40K for Antlr to 100K for Fop.

We compared IntraJ with the RAG-based framework JastAddJ-Intraflow [Söd+13a] and the SonarQube static code analyzer [Son]. Here, we summarize the comparison with SonarQube for the `DeadAssignment` and `NullPointerException` analyses.

In terms of performance, IntraJ generally outperforms SonarQube. For the `DeadAssignment` analysis, IntraJ has a lower baseline time compared to SonarQube, although SonarQube's analysis execution time is faster. This speed discrepancy is likely due to SonarQube pre-computing the control-flow graph, adding overhead to the baseline time.

For the `NullPointerException` analysis, IntraJ also shows better performance than SonarQube across all benchmarks, even when excluding baseline measurements.

Regarding precision, IntraJ performs better than SonarQube in the `DeadAssignment` analysis, detecting more true positives and fewer false positives. However, for the `NullPointerException` analysis, IntraJ is slightly less precise than SonarQube. Generally, IntraJ detects at least as many reports as SonarQube. However, there is one exception: in the case of PMD, where SonarQube can identify three additional true positives by exploiting path sensitivity. On the other hand, IntraJ does report some additional true positives that SonarQube does not. The false positives reported by IntraJ are a result of our analysis lacking path sensitivity.

Overall, the results suggest that IntraJ enables practical dataflow analyses, with run-times and precision comparable to state-of-the-art tools.

4 IntraTeal: IntraCFG for Teal

As a further demonstration[5] of the applicability of IntraCFG, we developed IntraTeal[6], an implementation of IntraCFG for the Teal programming language. Teal v0.4 (Typed Easily Analysable Language) is a programming language designed by Christoph Reichenbach and used in the program analysis[7] course at Lund University. Teal aims to provide a language that allows students to focus on the challenges of performing program analysis on a real-world language without being overwhelmed by the details of a fully-featured language. Teal is divided in

[5]This application of IntraCFG has not been peer-reviewed by the scientific community.
[6]The complete source code of IntraTeal is available at 10.5281/zenodo.7649171.
[7]`https://fileadmin.cs.lth.se/cs/Education/EDAP15/2022/web/index.html`

layers, with each version building upon the features of the previous one. Teal-0 is the most basic version and includes support for variable declarations and use, procedures, and basic control structures such as if and while statements. Teal-1 introduces the enhanced-for loop, and Teal-2 introduces user-defined classes. In this section, we will use Teal-0 to exemplify the use of RAGs and IntraCFG.

IntraTeal is our implementation of IntraCFG for the Teal language. In line with the approach taken for the implementation of the control-flow analysis for IntraJ and different versions of Java, we created separate aspects for the control-flow analysis of Teal-0 and Teal-1. The control-flow analysis is then used to implement the `NullPointerException` analysis. The overall architecture of IntraTeal is shown in Figure 14.

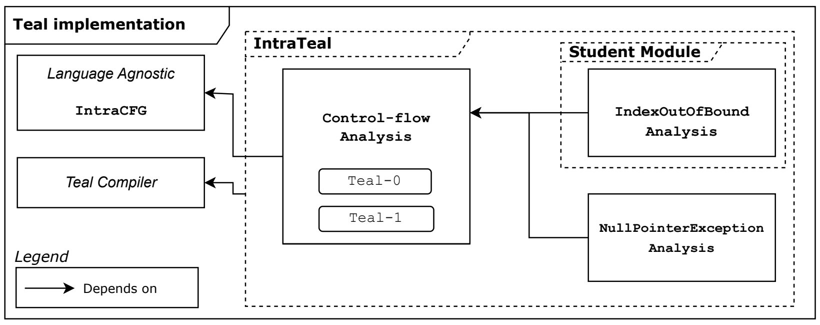


Figure 14: Overall architecture of IntraCFG instantiated for the Teal language.

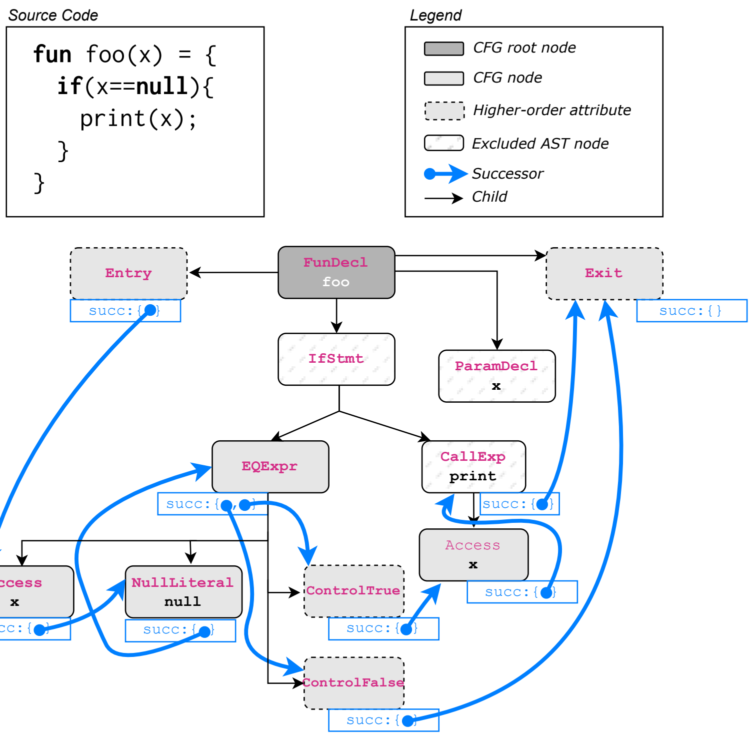
As part of the course, the complete source code of the IntraTeal control-flow analysis was provided to students, along with instructions and guidelines for utilizing the API to implement their analyses. We also made available a reference implementation of the `NullPointerException` analysis. To extend their understanding and skills, we then asked students to implement an `IndexOutOfBound` analysis on the interval abstract domain. This exercise allowed the students to apply the concepts they had learned in a practical setting and gain a deeper understanding of dataflow analysis.

```
if(x!=null){
  x.toString();
}
```

Listing 2: Control sensitivity to improve null pointer analysis.

Another key aspect of the IntraTeal and IntraJ implementations are their ability to construct enhanced control-sensitive CFGs. For instance, they track that, in the code shown in Listing 2, the dereference of `x` inside the body of the if-statement is safe to execute without throwing a `NullPointerException`.

Control sensitive CFGs are a more precise representation of the control flow



```
fun foo(x) = {
  if(x==null){
    print(x);
  }
}
```

Figure 15: Example of control sensitivity in IntraTeal.

of a program, as they take into account the different behavior when a branching condition is true or false. This is achieved by using HOAs to synthesise two AST nodes, i.e., `ControlTrue` and `ControlFalse`, for each comparison operator, e.g., "$\leq$", "$==$", "$!=$", inside a conditional expression. These HOAs are used to enhance the control flow of the program with the information that can be inferred from it.

Figure 15 shows IntraTeal's control sensitive CFG for a simple program with an if-statement. The `ControlTrue` and `ControlFalse` HOAs are used to distinguish the execution path when the condition in the if-statement evaluates to true or false, respectively. The `NullPointerException` analysis can benefit from this more refined CFG.

Listing 3: Control sensitivity to improve null pointer analysis.

```
if(x!=null){
//x is not null here
}
```

Listing 4: Control sensitivity to improve interval analysis.

```
if(x > 4 and x <= 6){
  //x is [5,6] here
}
```

For the example in Listing 3 the `ControlTrue` and `ControlFalse` HOAs are used to keep track of the information that the object x is not null in the *then* branch and null in the *else* branch. This information is used to improve the precision of the analysis and provide more accurate results without affecting the performance of the analysis.

We also asked students to use the `ControlTrue` and `ControlFalse` HOAs to improve the precision of the interval analysis. Similarly to the previous example, in Listing 4 the `ControlTrue` and `ControlFalse` are used to keep track of the information that the object x is in the interval [5,6].

5 IDE Integration

In this Section, we focus on the integration of IntraJ and IntraTeal with different IDEs and developer tools. We first describe the integration of IntraJ with IDEs that support the Language Server Protocol (LSP) [Mica], such as Visual Studio Code [Micb], Emacs [Fou], and Vim[Moo], using the MagpieBridge [LDB19a] framework. We then describe the integration of IntraJ and IntraTeal with CodeProber [Ala+24a], a tool for visualising and exploring the results of compilers and static analysis tools. Our work on integrating IntraJ with different IDEs via the MagpieBridge framework has gained attention from the MagpieBridge maintainer and resulted in an invitation to present at the PRIDE[8] workshop, held in conjunction with ECOOP 2022, with the title *"Source-Level Dataflow-Based Fixes: Experiences From Using IntraJ and MagpieBridge"*.

The integration process for the Teal language is not covered in any of the papers included in this thesis. However, it was developed as an application of the research and methods previously described in these papers, and was carried out subsequently.

5.1 LSP support via MagpieBridge: warnings, quick-fixes and bug explanations

Initially, IntraJ was developed as a command-line tool, which performance was competitive compared to existing industrial tools. However, we recognized the potential for further improvement, by exploiting the on-demand evaluation feature of JastAdd. On-demand evaluation enables the execution of dataflow anal-

[8]**P**ractical **R**esearch **IDE**s.

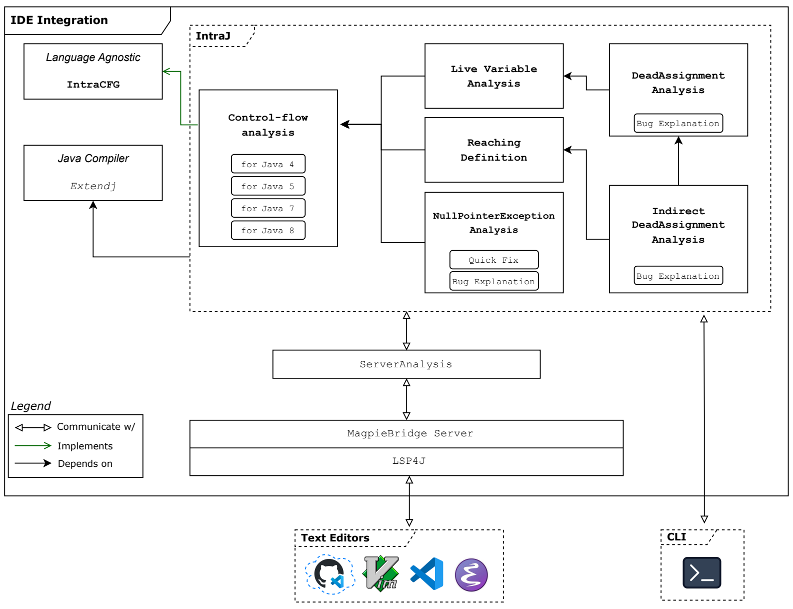


Figure 16: Integration of IntraJ with IDEs through the use of the MagpieBridge framework.

yses on methods within open files, as opposed to the entire codebase. This approach allows for local and in real-time feedback on complex bugs, providing developers with instantaneous insights, facilitating the debugging process. To achieve this, we used the MagpieBridge framework, which facilitates the integration of static analysers with IDEs that support the LSP. MagpieBridge provides an abstraction layer between the IDE and the static analysis tool, simplifying the integration process and allowing for the development of IDE plug-ins with minimal effort. MagpieBridge provides an abstraction layer between the IDE and the static analysis tool allowing the display of warnings, quick-fixes, and explanations for bugs within the IDE, providing developers with an immediate and convenient way to access and interact with analysis results, while also facilitating communication between the static analysis tool and the IDE. Additionally, the framework allows for the display of web pages within the IDE, providing developers with a new level of support for visualization, customizable user interfaces, and a better way to interact with analysis results.

Figure 16 illustrates the integration of IntraJ with different IDEs. ServerAnalysis is a component that we developed to handle the communication between IntraJ and the MagpieBridge Server. It is responsible for maintaining a record of the active analyses and forwarding events in the editor, such as the save command or opening of a file, to IntraJ. The results of the analysis are then sent back to the MagpieBridge, which subsequently forwards them to the editor, displaying warnings, quick-fixes, and explanations to the developer. To enable a better user experience, we extended the functionality of the existing analysis. Specifically, we enhanced the `NullPointerException` analysis to not only detect issues but also provide developers with quick fixes and explanations, allowing them to address the problems more efficiently. Additionally, we enhanced the `(Indirect) Dead Assignment` analysis to provide explanations, giving developers a deeper understanding of the issues detected. Figure 17 illustrates an example of interaction between IntraJ and Visual Studio Code. It illustrates an instance of a `NullPointerException` detected by IntraJ and its representation within the IDE. The "💡" icon indicates that an quick-fix is available, which can be applied by clicking on the icon.

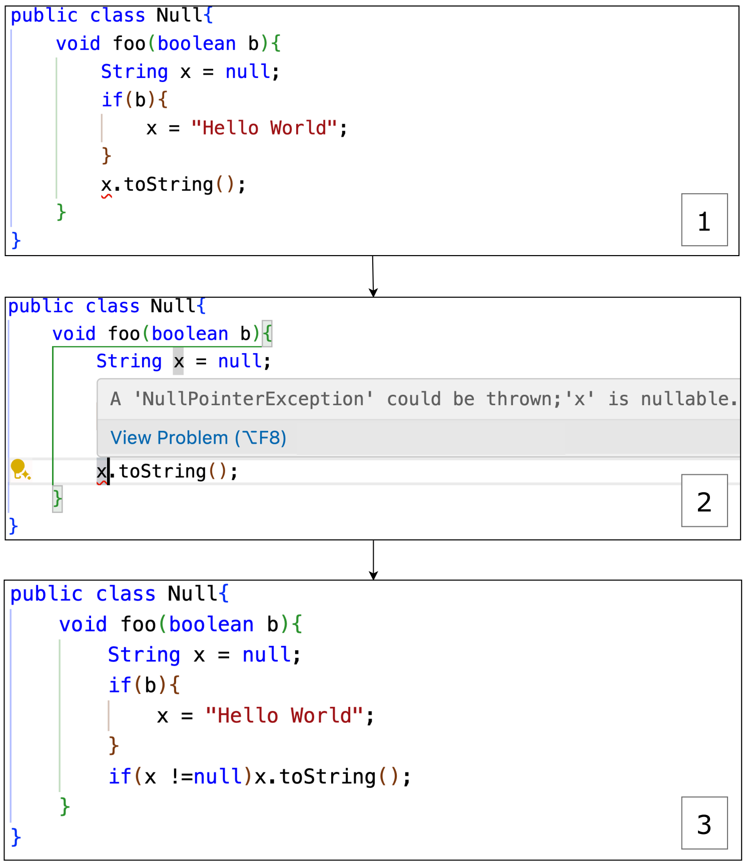
5.2 Visualisation via CodeProber

In this Section, we will give an overview of the integration of IntraTeal with CodeProber, a tool for visualizing and exploring the results of compilers and static analysers. CodeProber allows developers to interact with the results of the analysis in a visual and intuitive manner. It enables real-time interaction with the AST node's attributes and the source code, enabling analysis developers to explore results and partial results, making debugging and troubleshooting more efficient in comparison to the traditional debugging approaches. As a browser-based tool that is not restricted to the Language Server Protocol, CodeProber enables the visualisation of analysis results in different formats, including, but not limited to, graph representation and other visual forms, beyond simply displaying warnings. The example in Figure 18 shows the visual representation of the CFG on top of the source code. The CFG is generated by IntraTeal and visualized by CodeProber. The graph is rendered automatically at each change in the source code, allowing developers to understand the flow of the program and all the possible execution paths of the analysis in real-time.

We used the IntraTeal and CodeProber integration in the Program analysis course. Students were able to understand the CFG and the flow of the program and were asked to identify IndexOutOfBound exceptions. Students were able to observe their progress and the outcomes of their analysis within a realistic IDE.

6 JFeature: Java Feature Extractor

JFeature is a RAG-based static analysis tool for the Java programming language that extracts syntactic and semantic features from Java programs. The tool is designed to assist researchers and developers in selecting appropriate software corpora to better evaluate the robustness and performance of software tools, such as static analysers. JFeature is implemented as an extension of the ExtendJ Java compiler. It is declarative and extensible, allowing for the easy addition of new

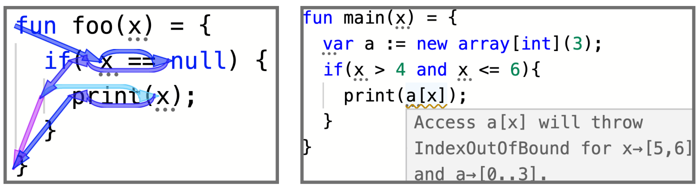


```
public class Null{
    void foo(boolean b){
        String x = null;
        if(b){
            x = "Hello World";
        }
        x.toString();
    }
}
```

```
public class Null{
    void foo(boolean b){
        String x = null;
        A 'NullPointerException' could be thrown;'x' is nullable.
        View Problem (⌥F8)
        x.toString();
    }
}
```

```
public class Null{
    void foo(boolean b){
        String x = null;
        if(b){
            x = "Hello World";
        }
        if(x !=null)x.toString();
    }
}
```

Figure 17: Bug detection and quick-fix in Visual Studio Code using IntraJ. 1. The `NullPointerException` is detected by IntraJ (squiggly line under `x`) with a quick-fix available (💡). 2. The user can hover over the warning to see an explanation of the issue. 3. The user can click on the quick-fix icon (💡) to apply the fix.



```
fun foo(x) = {
  if(x == null) {
    print(x);
  }
}
```

```
fun main(x) = {
  var a := new array[int](3);
  if(x > 4 and x <= 6){
    print(a[x]);
  }
}
```

```
Access a[x] will throw
IndexOutOfBound for x→[5,6]
and a→[0..3].
```

Figure 18: Interaction of IntraTeal in CodeProber. Colors are randomly assigned to make it easier to distinguish individual arrows.

queries. In this Section, we give an overview of this work, and more details are available in Paper 3.

The need for JFeature arose during the evaluation of IntraJ. While analysing Java projects from the DaCapo Benchmark suite [Bla+06] corpus to evaluate the precision of IntraJ on Java 8 projects, it became apparent that there were no Java 8 projects in the Da Capo Benchmark suite. Further investigation revealed that many commonly used software corpora in the field of static analysis were lacking representation of Java 8 projects.

To address this problem, we developed JFeature, a tool that extracts features from Java programs categorised by the Java version they were introduced in. The goal of JFeature is to provide insight and an overview of the composition of a Java project or corpus, specifically in terms of the different Java features categorized by the Java version in use. JFeature comes with twenty-six predefined queries and can be easily extended with new ones. Since JFeature is built on top of the ExtendJ compiler, JFeature has access to all the information computed by the compiler, allowing the definition of complex queries. In Figure 19, we show the architecture of JFeature.

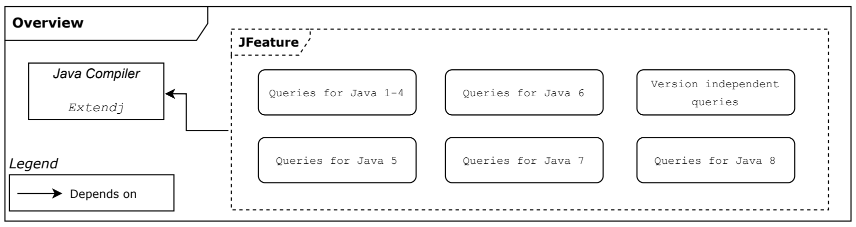


Figure 19: JFeature architecture.

We conducted a case study, applying JFeature to four widely used corpora in the program analysis area: the Da Capo Benchmark suite [Bla+06], Defects4J [JJE14], Qualitas Corpus [Tem+10a], and XCorpus [Die+17]. The results showed that Java 1-5 features were predominant among the corpora, suggesting that some of the corpora may be less suited for the evaluation of tools that

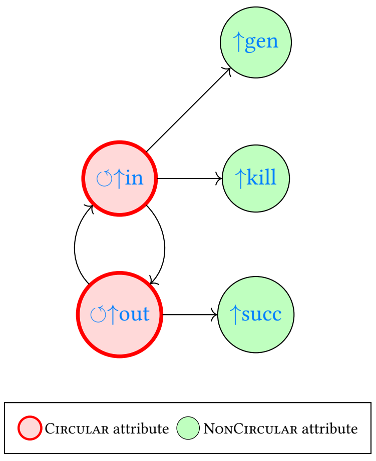


Figure 20: Circular dependencies in backward dataflow analysis between `in` and `out`, and their relationships with other attributes.

address features in Java 7 and 8. In addition to evaluating corpora, we showed how JFeature could also be used for other applications such as longitudinal studies of individual Java projects and the creation of new corpora. In Paper 3, we also demonstrate a practical application of how JFeature can be extended to capture more complex semantic features by writing queries using the RAGs formalism.

7 Using Static Analysis to Improve the Efficiency of Circular Attributes

Circular attributes, whose values may directly or indirectly depend on themselves, allow developers to declaratively express complex relationships. These attributes are particularly useful in representing dataflow analyses.

As illustrated in Figure 20, the `in` and `out` attributes in backward dataflow analysis may depend on one another. However, the evaluation of these attributes requires fixed-point computations to ensure convergence, which, if not handled correctly, can be inefficient. This section explores the challenges posed by circular attributes and their evaluation, and how static analysis techniques can be applied to improve their efficiency.

7.1 Circular Attribute Evaluation for RAGs

For the RAGs compiler, in this case JastAdd, to generate correct evaluation code, it must first recognize that certain attributes, such as `in` and `out`, may exhibit circular dependencies and require fixed-point computations. However, this process is not always straightforward, as the compiler must take into account the following challenges:

- **Dynamic dependencies:** The compiler cannot always statically determine which attributes are circular because their circularity depends on the specific structure of the AST being analyzed. This dynamic nature means that circular dependencies may only be discovered during evaluation, making it difficult to predict in advance. For instance, let us consider the following example:

$$\texttt{A}.\circlearrowright\!\uparrow\!\text{foo}(\texttt{boolean b}) = \begin{cases} \texttt{A}.\circlearrowright\!\uparrow\!\text{foo}(\text{false}) + 1 & \text{if } b = \text{true} \\ 0 & \text{otherwise} \end{cases}$$

  Here the `foo` attribute is circular when `b` is `true`, but not when `b` is `false`. A conservative and safe approach would be to treat `foo` as Circular in all cases.

- **Conservative approach in handling circular attributes:** Even when attributes are not circular for certain types, they might still be treated as circular due to potential circularity in subtypes. Let us consider the following example:

$$\begin{aligned} \texttt{A}.\circlearrowright\!\uparrow\!\text{bar} &= 0 \\ \texttt{B}.\circlearrowright\!\uparrow\!\text{bar} &= \min(\texttt{A}.\circlearrowright\!\uparrow\!\text{bar} + 1, 5) \end{aligned}$$

  In this example, the `bar` attribute is circular only for the `B` type, but never for the `A` type. However, if the compiler treats `bar` as Circular for instances of `A`, it will unnecessarily introduce fixed-point computations for attributes that are NonCircular in most cases. This is the case for the ExtendJ compiler, where the `type` attribute may only form a circular dependency in specific language constructs, such as lambda expressions. However, applying fixed-point computations for `type` in all cases would lead to unnecessary computational overhead in instances where no circularity exists.

- **Performance inefficiencies:** Fixed-point computations can introduce significant performance overhead, particularly when applied to a large connected components. If the component has smaller strongly connected components (SCCs), it is more efficient to isolate these smaller components and evaluate them separately. Smaller components can often reach their fixed points and terminate earlier, reducing the overall computational cost.

7.2 Challenges with Existing Solutions

The first attempt to address the challenges of circular attributes was the BasicStacked algorithm, proposed by Magnusson et al. [MH07b], which required developers to explicitly declare attributes as Circular when they could participate in a cycle. While this approach allows certain strongly connected components to be isolated, it introduced new inefficiencies and impracticalities.

In particular, the BasicStacked algorithm required developers to manually identify all attributes that may be on a cycle for any possible AST. This process can be error-prone and difficult to maintain, especially in large and evolving codebases. Figure 21 shows the callgraph of the ExtendJ compiler. Here each red node represents an attribute that is explicitly declared as Circular. Modifying a single attribute within the compiler can potentially introduce new circular dependencies, making it challenging to anticipate in advance which attributes will require circular declarations. Most of the time, this results in developers over-annotating attributes as Circular, leading to unnecessary fixed-point computations and performance overhead.

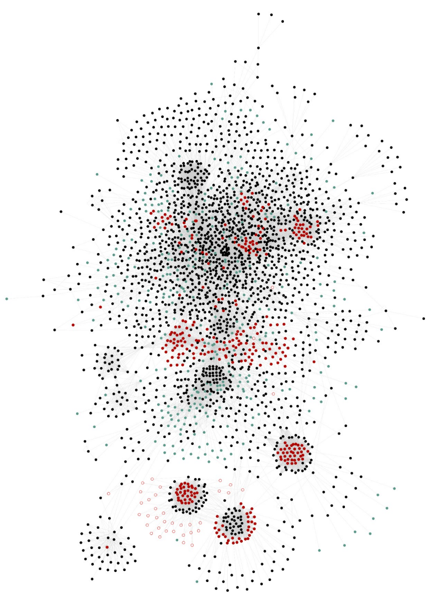


Figure 21: ExtendJ's call graph. Red nodes represent attributes explicitly declared as Circular. Other colors represents different types of attributes.

In response to these issues, a new algorithm called RelaxedMonolithic was proposed by Öqvist [ÖH17; Öqv18]. The RelaxedMonolithic algorithm improves upon BasicStacked by removing the requirement to declare all attributes in a cycle as Circular. Instead, only one attribute within the cycle needs to be explicitly marked as Circular, while the remaining attributes are automatically labeled as Agnostic. An attribute classified as Agnostic can be either Circular or NonCircular, depending on the context in which it is evaluated. While this approach provides more flexibility in handling circular attributes, it introduces new inefficiencies. For example, the RelaxedMonolithic algorithm does not recognize strongly connected components, evaluating all downstream attributes from a circular attribute in a single, large fixed-point computation. This can

Algorithm	Annotation Effort	Decompose SCCs	Supported Attributes
BasicStacked	High	Yes	Circular NonCircular
RelaxedMonolithic	Low	No	Circular Agnostic
relaxedstacked	Low	Yes	Circular NonCircular Agnostic

Table 1: Algorithm comparison based on their support for different attribute declarations.

lead to multiple recomputations of downstream attributes during fixed-point iterations, even when their values remain unchanged.

7.3 Static Analysis for Efficiency Improvement

To address the limitations of existing solutions, we propose a new evaluation algorithm, called RelaxedStacked, that combines the strengths of the BasicStacked and RelaxedMonolithic algorithms. This algorithm supports three kinds of attributes, Circular, NonCircular, and Agnostic. We then use static analysis to detect attributes that can be safely classified as NonCircular based on the static dependency graph of the attributes.

In order to identify NonCircular attributes, we developed the Callgraph Analysis Tool (CAT), which constructs a call graph of the Java evaluation code generated by JastAdd.

From this callgraph, CAT filters out non-attribute methods and then applies Tarjan's [Tar72] strongly connected component analysis to determine which attributes can be safely marked as NonCircular (i.e., they are guaranteed to not be on a cycle for any AST). Attributes that are part of a single-node SCC and do not directly call themselves are considered NonCircular and are evaluated without iterative fixed-point computations. Table 1 presents a comparison of the three algorithms based on their support for different attribute declarations.

To better understand the differences between the BasicStacked, RelaxedMonolithic, and RelaxedStacked algorithms, let us consider how the attributes in Figure 20 can be classified by each algorithm. Figure 22 illustrates the classification of these attributes which are part of a backward dataflow analysis for each algorithm.

With the BasicStacked algorithm, developers are required to explicitly declare the `in` and `out` attributes as Circular as they depend on each other. With-

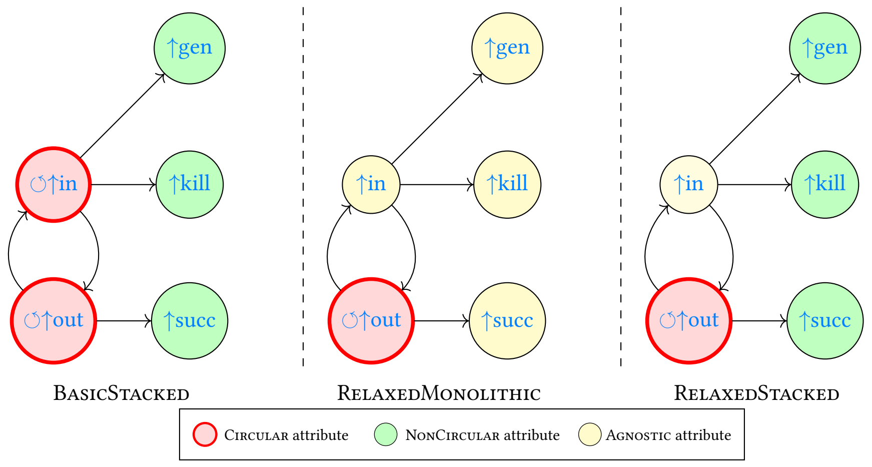


Figure 22: Attribute classification using the BasicStacked, RelaxedMonolithic, and RelaxedStacked algorithms.

out this explicit declaration, the algorithm may fail to recognize circular dependencies, which could lead to infinite recursion and result in a runtime error, such as a stack overflow. Therefore, the correctness of the evaluation process is highly dependent on the developer's annotations.

The RelaxedMonolithic algorithm simplifies the specification of circular attributes by requiring only one attribute in a cycle to be explicitly declared as Circular; for instance, the `CFGNode.↻↑out` attribute, while the remaining attributes can be left as Agnostic. When the `in` attribute requires multiple iterations to reach a fixed point, attributes such as `gen`, `kill`, and `succ` are recomputed in each iteration. For example, if `out` requires n iterations, these downstream attributes—`gen`, `kill`, and `succ`—are unnecessarily recalculated n times, even though their values remain constant after the first iteration.

The RelaxedStacked algorithm builds upon the RelaxedMonolithic classification for Circular attributes but reclassifies attributes such as `gen`, `kill`, and `succ` as NonCircular. This approach computes the values of the `gen`, `kill`, and `succ` attributes only once, and these values are memoized for all subsequent iterations when computing `out`, thereby avoiding redundant recomputation.

While effective, our approach does have some limitations. Its effectiveness is constrained by the accuracy of the static analysis tool. For instance, our call graph analysis, which is based on *Class Hierarchy Analysis*[DGC95], conservatively treats method calls that appear recursive in the AST as circular, even though they not exhibit circular behavior in practice. Furthermore, this approach cannot detect dynamically strongly connected components, which may only become apparent during evaluation.

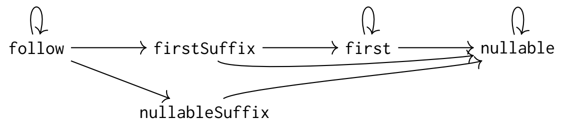


Figure 23: Simplified static call graph of the attributes defining the *nullable*, *first*, and *follow* sets.

7.4 Evaluation

To compare the performance of the RelaxedStacked algorithm with that of the BasicStacked and RelaxedMonolithic algorithms, we conducted three distinct experiments which we describe below.

LL(1) Parser Construction

We tested the algorithms on a simple LL(1) parser construction scenario, where the `follow`, `first`, and `nullable` properties are defined using circular attributes.

This experiment was the only scenario where all three algorithms could be directly compared, as the complexity of the ExtendJ compiler and the IntraJ static analyser made them unsuitable for evaluation with the BasicStacked algorithm.

The static call graph of the attributes is shown in Figure 23. In this scenario, the RelaxedStacked algorithm matches the efficiency of the BasicStacked algorithm while providing the flexibility of the RelaxedMonolithic algorithm, and it outperforms the RelaxedMonolithic algorithm with a speedup of approximately 2.8x. This because, from the callgraph, CAT is able to identify `firstSuffix` and `nullableSuffix` as NonCircular attributes, which are evaluated without fixed-point computations. This allows to break the evaluation of the circular attributes into smaller SCCs, which can be evaluated more efficiently. In contrast, the RelaxedMonolithic algorithm evaluates all the attributes in a single large SCC, leading to unnecessary recomputations of all the downstream attributes.

ExtendJ

The development of the RelaxedMonolithic algorithm was initially driven by the requirements of the ExtendJ Java compiler, which contains relatively few circular attributes. This made it an ideal candidate to determine if the RelaxedStacked algorithm introduces any overhead compared to the RelaxedMonolithic algorithm.

In this case study, the RelaxedStacked algorithm performs equally as the RelaxedMonolithic algorithm, demonstrating that it does not introduce any additional overhead, even in scenarios with relatively few circular attributes.

IntraJ

We evaluated the RelaxedMonolithic and RelaxedStacked algorithms using the IntraJ static analysis tool, focusing on two different dataflow analyses: *Null-Pointer Dereference Analysis* and *Dead Assignment Analysis*. Unlike ExtendJ, IntraJ involves a high number of circular attributes, making it a robust test for the RelaxedStacked algorithm's capabilities.

In the IntraJ case study, the RelaxedStacked algorithm significantly outperforms the RelaxedMonolithic algorithm, achieving a median steady-state performance speedup of approximately 2.5x for dead-assignment analysis and about 18x for null-pointer dereference analysis.

We also evaluated the performance of our algorithm compared to the RelaxedMonolithic algorithm in demand-driven scenarios, focusing on execution time and the frequency of successor attribute recomputations. Due to space constraints in Paper 4, we were unable to present the full evaluation results. In this section, we provide the complete results. The data, presented in Figures 24 and 25, demonstrate that the RelaxedStacked algorithm significantly outperforms the RelaxedMonolithic algorithm. The results are consistent across all projects and align with the outcomes observed in the whole program analysis. As shown in the figures, the RelaxedStacked algorithm significantly reduces the number of successor attribute evaluations, which in turn leads to a substantial reduction in execution time.

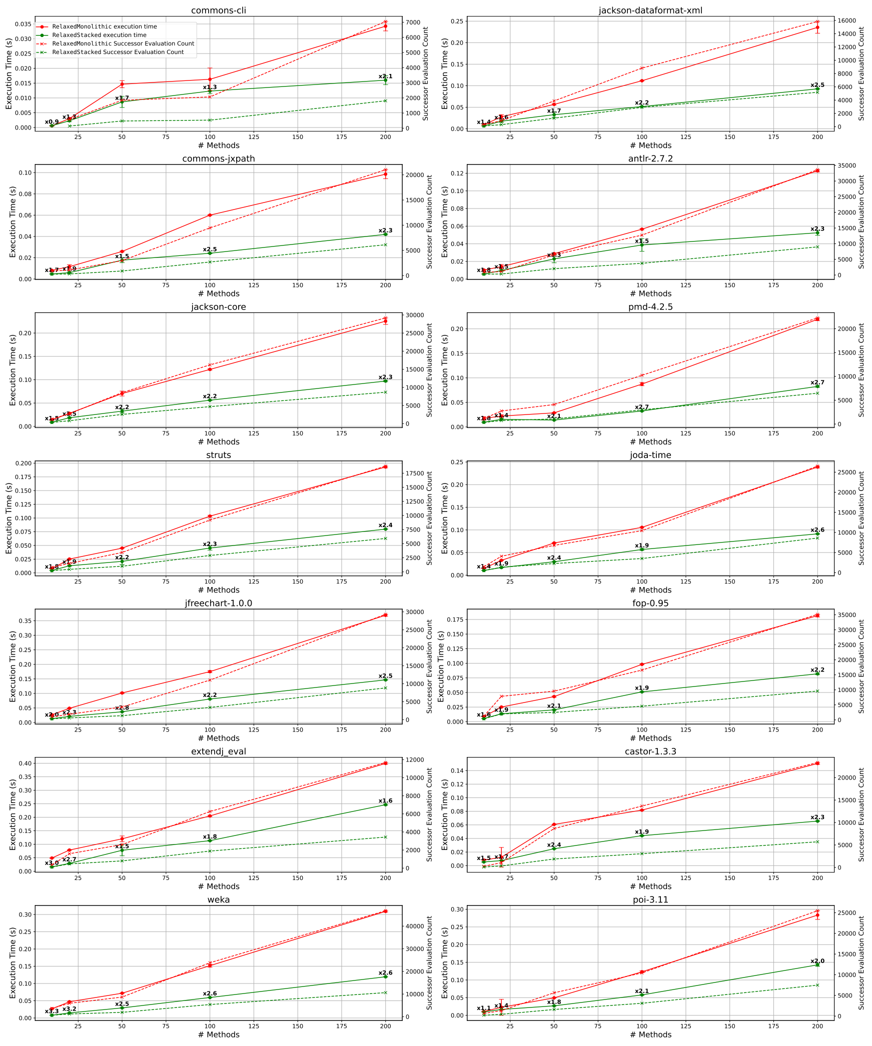


Figure 24: Steady-state performance of dead assignment analysis for randomly selected sets of methods of 14 real world projects. In red we report the performance of the RelaxedMonolithic algorithm, while in green we report the performance of the RelaxedStacked algorithm. Solid lines represent execution time (left axis, seconds). Dashed lines represent successor attribute evaluations (right axis, count).

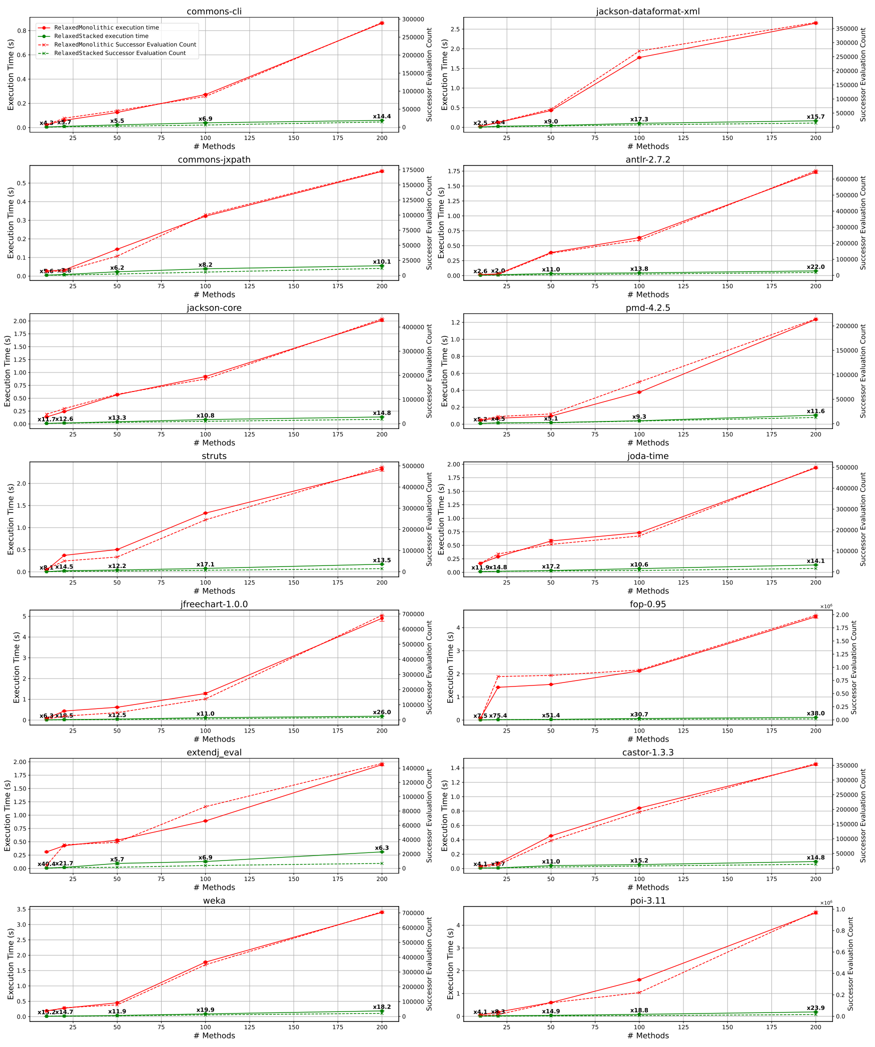


Figure 25: Steady-state performance of null-pointer dereference for randomly selected sets of methods of 14 real world projects. In red we report the performance of the RelaxedMonolithic algorithm, while in green we report the performance of the RelaxedStacked algorithm. Solid lines represent execution time (left axis, seconds). Dashed lines represent successor attribute evaluations (right axis, count).

8 Conclusions

This thesis has explored several challenges in static analysis, particularly focusing on improving control-flow and dataflow analyses within the context of Reference Attribute Grammars. The solutions developed offer a practical approach to improving both the precision and efficiency of these analyses, especially in interactive settings like integrated development environments.

Our main contribution is the development of a new framework for precise construction of source-level control-flow graphs, called IntraCFG. IntraCFG is language-agnostic and is designed to be easily extensible to support new languages. It provides a flexible and efficient way to construct arbitrarily precise control-flow graphs. This framework was applied successfully to two different languages resulting in two different tools: IntraJ for analyzing Java code and IntraTeal, an educational tool aimed at teaching program analysis concepts. The framework has proven flexible and adaptable across different use cases.

We have demonstrated that IntraJ can effectively be executed both as a standalone tool and as an integrated plugin within an IDE. In both modes, IntraJ has demonstrated faster performance than a well-known commercial tool. Even for non-trivial analyses, it consistently meets the goal of providing feedback in under 0.1 seconds.

In this thesis, we also presented JFeature, an extensible tool for automatically extracting and summarising the key features of a corpus of Java programs. JFeature allows researchers and developers to gain a deeper understanding of the composition and suitability of software corpora for their particular research or development needs. By applying JFeature to four widely-used corpora in the program analysis area, we demonstrated its potential for use in corpus evaluation, the creation of new corpora, and longitudinal studies of individual Java projects. Together, these contributions provide frameworks and practical tools for improving the development and maintenance of software systems.

In addition to these contributions, this thesis also presented a new demand-driven evaluation algorithm for circular attributes, named RelaxedStacked. This algorithm combines the strengths of previous approaches, such as the BasicStacked and RelaxedMonolithic algorithms, while introducing optimizations that reduce redundant recomputation of circular attributes. By identifying strongly connected components within the attribute dependency graph, the RelaxedStacked algorithm improves the efficiency of attribute evaluation, especially in the presence of complex circular dependencies. In particular, we used call graph analysis to identify attributes that can be safely classified as NonCircular, allowing them to be evaluated only once and immediately memoised. The results of our evaluation demonstrate the effectiveness of the RelaxedStacked algorithm in real-world scenarios. The benchmarking results showed a significant improvement in performance, with a median speedup of $\sim$2.5x for dead assignment analysis and a median speedup of $\sim$18x for null-pointer dereference analysis, compared to RelaxedMonolithic.

Throughout this work, we have placed a strong emphasis on ensuring that our results are open, easily accessible, and reproducible. The artifacts created, including code, documentation, and Docker images, are all publicly available, allowing others to replicate and build upon this research.

In conclusion, the tools and methods developed in this thesis contribute to making static analysis more practical for interactive use. By focusing on both precision and efficiency, and ensuring that our work is reproducible, we hope it will serve as a foundation for future research in this area.

References

[Aho+07] Alfred V Aho et al. *Compilers: principles, techniques, & tools.* Pearson Education India, 2007.

[Ala+24a] Anton Risberg Alaküla et al. “Property probes: Live exploration of program analysis results”. In: *J. Syst. Softw.* 211 (2024), p. 111980.

[App04] Andrew W Appel. *Modern compiler implementation in C.* Cambridge university press, 2004.

[Arz+14] Steven Arzt et al. “FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps”. In: *SIGPLAN Not.* 49.6 (2014), 259–269.

[Aye+08a] Nathaniel Ayewah et al. “Using Static Analysis to Find Bugs”. In: *IEEE Software* 25.5 (2008), pp. 22–29.

[Aye+08b] Nathaniel Ayewah et al. “Using static analysis to find bugs”. In: *IEEE software* 25.5 (2008), pp. 22–29.

[Bla+06] S. M. Blackburn et al. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. In: *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications.* Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190.

[Bla+02] Bruno Blanchet et al. “Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software”. In: *The Essence of Computation: Complexity, Analysis, Transformation.* Ed. by Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 85–108.

[Bod18] Eric Bodden. “The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them)”. In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops.* ISSTA ’18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, 85–93.

[Boy96] John Tang Boyland. “Descriptional Composition of Compiler Components”. PhD thesis. University of California, Berkeley, 1996.

[Cou+05] Patrick Cousot et al. “The ASTRÉE analyzer”. In: *European Symposium on Programming*. Springer. 2005, pp. 21–30.

[DR+11] Coen De Roover et al. "The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Kongens Lyngby, Denmark: Association for Computing Machinery, 2011, pp. 71–80.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP '95. Berlin, Heidelberg: Springer-Verlag, 1995, 77–101.

[Die+17] Jens Dietrich et al. "XCorpus - An executable Corpus of Java Programs". In: *Journal of Object Technology* 16.4 (2017), 1:1–24.

[Don68] Thomas Donnellan. *Lattice Theory*. Pergamon, 1968.

[DR24] Alexandru Dura and Christoph Reichenbach. "Clog: A Declarative Language for C Static Code Checkers". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. , Edinburgh, United Kingdom: Association for Computing Machinery, 2024, 186–197.

[DRS21] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. "JavaDL: Automatically Incrementalizing Java Bug Pattern Detection". In: *Proceedings of the ACM on Programming Languages*. Virtual: ACM, 2021.

[EH07a] Torbjörn Ekman and Görel Hedin. "The jastadd extensible Java compiler". In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. 2007, pp. 1–18.

[EH07b] Torbjörn Ekman and Görel Hedin. "The jastadd extensible java compiler". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[Far86] Rodney Farrow. "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars". In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*. ACM, 1986, pp. 85–98.

[FD12] Stephen Fink and Julian Dolby. *WALA–The TJ Watson Libraries for Analysis*. 2012.

[FH16] Niklas Fors and Görel Hedin. “Bloqqi: modular feature-based block diagram programming”. In: *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*. Ed. by Eelco Visser, Emerson R. Murphy-Hill, and Cristina V. Lopes. ACM, 2016, pp. 57–73.

[FH15] Niklas Fors and Görel Hedin. “A JastAdd implementation of Oberon-0”. In: *Science of Computer Programming* 114 (2015). LDTA (Language Descriptions, Tools, and Applications) Tool Challenge, pp. 74–84.

[Fou] Free Software Foundation. *GNU Emacs*. `https://www.gnu.org/software/emacs/`. Accessed: 2023-02-27.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000).

[HM01] Görel Hedin and Eva Magnusson. “JastAdd - a Java-based system for implementing front ends”. In: *Electron. Notes Theor. Comput. Sci.* 44.2 (2001), pp. 59–78.

[Int] *Intel VTune Amplifier*. `https://software.intel.com/en-us/vtune`. Accessed: 2023-01-27. 2021.

[JS86] Larry G. Jones and Janos Simon. “Hierarchical VLSI Design Systems Based on Attribute Grammars”. In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. ACM Press, 1986, pp. 58–69.

[JJE14] René Just, Darioush Jalali, and Michael D Ernst. “Defects4J: A database of existing faults to enable controlled testing studies for Java programs”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 437–440.

[Kan+10] Sean Kane et al. “Update Report: Toyota Sudden Unintended Acceleration”. In: *Safety Research & Strategies (SRS)* (2010).

[Kic+97] Gregor Kiczales et al. “Aspect-oriented programming”. In: *ECOOP’97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[Kil73] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: Association for Computing Machinery, 1973, 194–206.

[Knu68] Donald E Knuth. “Semantics of context-free languages”. In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[Lab03] Open Source Development Lab. *Valgrind: A Framework for Memory Debugging, Profiling and Analysis.* `http://www.valgrind.org/`. Accessed: 2023-01-27. 2003.

[Mica] *Language Server Protocol.* `https://microsoft.github.io/language-server-protocol/`. Accessed: 2023-02-27.

[LA04] C. Lattner and V. Adve. “LLVM: a compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86.

[LT93] Nancy G Leveson and Clark S Turner. “An investigation of the Therac-25 accidents”. In: *Computer* 26.7 (1993), pp. 18–41.

[LZZ18] Jun Li, Bodong Zhao, and Chao Zhang. “Fuzzing: a survey”. In: *Cybersecurity* 1.1 (2018), pp. 1–13.

[Liv+15] Benjamin Livshits et al. “In defense of soundiness: A manifesto”. In: *Communications of the ACM* 58.2 (2015), pp. 44–46.

[LDB19a] Linghui Luo, Julian Dolby, and Eric Bodden. “MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper)”. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019).* Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 21:1–21:25.

[MYL16b] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. “From Datalog to flix: a declarative language for fixed points on lattices”. In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 194–208.

[MH07b] Eva Magnusson and Görel Hedin. “Circular reference attributed grammars—their evaluation and applications”. In: *Science of Computer Programming* 68.1 (2007), pp. 21–37.

[MH07c] Eva Magnusson and Görel Hedin. “Circular reference attributed grammars — their evaluation and applications”. In: *Science of Computer Programming* 68.1 (2007). Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA ’03), pp. 21–37.

[MS18] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis.* `http://cs.au.dk/~amoeller/spa/`. Department of Computer Science, Aarhus University. 2018.

[Moo] Bram Moolenaar. *VIM - Vi IMproved*. `https://www.vim.org/`. Accessed: 2023-02-27.

[Nie94] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[Öqv18] Jesper Öqvist. "Contributions to Declarative Implementation of Static Program Analysis". PhD thesis. Lund University, Sweden, 2018.

[OH13] Jesper Öqvist and Görel Hedin. "Extending the JastAdd extensible Java compiler to Java 7". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '13. Stuttgart, Germany: Association for Computing Machinery, 2013, 147–152.

[ÖH17] Jesper Öqvist and Görel Hedin. "Concurrent circular reference attribute grammars". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, 2017, pp. 151–162.

[PKB21] Goran Piskachev, Ranjith Krishnamurthy, and Eric Bodden. "SecuCheck: Engineering configurable taint analysis for software developers". In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2021, pp. 24–29.

[Pis+22] Goran Piskachev et al. "How far are German companies in improving security through static program analysis tools?" In: *2022 IEEE Secure Development Conference (SecDev)*. IEEE. 2022, pp. 7–15.

[Ric53] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[Rio+21] Idriss Riouak et al. "A Precise Framework for Source-Level Control-Flow Analysis". In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2021, pp. 1–11.

[SS03] Akira Sasaki and Masataka Sassa. "Circular Attribute Grammars with Remote Attribute References and their Evaluators". In: *New Generation Computing* 22.1 (2003), pp. 37–60.

[Saw99] Kathy Sawyer. “Mystery of Orbiter Crash Solved”. In: *Washington Post* (Oct. 1, 1999). Accessed: 2023-01-27, A1.

[Say+22] Imen Sayar et al. “An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities”. In: *ACM Transactions on Software Engineering and Methodology* (2022).

[SKV10] Anthony M. Sloane, Lennart C.L. Kats, and Eelco Visser. “A Pure Object-Oriented Embedding of Attribute Grammars”. In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010). Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 205–219.

[SWV20] Jeff Smits, Guido Wachsmuth, and Eelco Visser. “FlowSpec: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis”. In: *Journal of Computer Languages* 57 (2020), p. 100924.

[Söd+13a] Emma Söderberg et al. “Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level”. In: *Sci. Comput. Program.* 78.10 (Oct. 2013), 1809–1827.

[Söd+13b] Emma Söderberg et al. “Extensible intraprocedural flow analysis at the abstract syntax tree level”. In: *Sci. Comput. Program.* 78.10 (2013), pp. 1809–1827.

[Son] SonarSource. *SonarQube: Continuous Code Quality.* `https://www.sonarqube.org/`. Accessed: 2023-01-27.

[SEV16] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. “IncA: a DSL for the definition of incremental program analyses”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ASE ’16. Singapore, Singapore: Association for Computing Machinery, 2016, 320–331.

[Tar72] Robert Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM Journal on Computing* 1.2 (June 1972), pp. 146–160.

[Tar55] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications.” In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285 –309.

[Tem+10a] Ewan Tempero et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. In: *2010 Asia Pacific Software Engineering Conference (APSEC2010).* Dec. 2010, pp. 336–345.

[VR+10] Raja Vallée-Rai et al. “Soot: A Java Bytecode Optimization Framework”. In: *CASCON First Decade High Impact Papers.* CASCON ’10. Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224.

[Van+10a] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1 (2010). Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA ’06 and ’07), pp. 39–54.

[Micb] *Visual Studio Code*. `https://code.visualstudio.com/`. Accessed: 2023-02-27.

[VSK89b] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. “Higher Order Attribute Grammars”. In: *SIGPLAN Not.* 24.7 (1989), 131–145.

Included Papers

A Precise Framework for Source-Level Control-Flow Analysis

Abstract

This paper presents IntraCFG, a declarative and language-independent framework for constructing precise intraprocedural control-flow graphs (CFGs) based on the reference attribute grammar system JastAdd. Unlike most other frameworks, which build CFGs on an Intermediate Representation level, e.g., bytecode, our approach superimposes the CFGs on the Abstract Syntax Tree, enabling accurate client analysis. Moreover, IntraCFG overcomes expressivity limitations of an earlier RAG-based framework, allowing the construction of AST-Unrestricted CFGs: CFGs whose shape is not confined to the AST structure. We evaluate the expressivity of IntraCFG with IntraJ, an application of IntraCFG to Java 7, by comparing two data flow analyses built on top of IntraJ against tools from academia and from the industry. The results demonstrate that IntraJ is effective at building precise and efficient CFGs and enables analyses with competitive performance.

Idriss Riouak, Christoph Reichenbach, Görel Hedin and Niklas Fors.
"A Precise Framework for Source-Level Control-Flow Analysis".
In *21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 1–11.*
Virtual, 2021, IEEE.
Paper DOI: 10.1109/SCAM52516.2021.00009.

1 Introduction

Static program analysis plays an important role in software development, and may help developers detect subtle bugs such as null pointer exceptions [HSP05] or security vulnerabilities [Smi+15].

Many client analyses make use of intraprocedural control-flow analysis, and are dependent on its precision and efficiency for useful results. Bug checkers and other clients that report to the user must be able to link their results to the source code, so the control-flow analysis itself must also connect to a representation close to the source code, such as an abstract syntax tree (AST). Current mainstream program analysis tools and IDEs, like SonarQube, ErrorProne, and Eclipse JDT, take this exact approach.

However, building analyses at the AST level typically ties the analysis closely to a particular language and thereby reduces opportunities for reuse. Furthermore, language semantics can require highly intricate control flow, e.g. for object initialisation and exception handling.

In this paper, we present an approach for developing control-flow analyses and client analyses at the AST level that is based on reference attribute grammars (RAGs) [Hed00] and addresses these challenges. We build on an earlier approach that also used RAGs [Söd+13a] and remove its two main limitations: imprecision and bloat, both caused by limited flexibility in the shape of control-flow graphs (CFGs) that could be built. Our approach introduces a new generalised framework, IntraCFG, that is unrestricted in the shape of the CFGs that it can build. This improves precision as well as conciseness, in that IntraCFG connects only AST nodes of interest in the CFG. As a case study, we applied IntraCFG to the Java language, implementing IntraJ, a CFG constructor for Java, as an extension of the Java compiler ExtendJ [EH07a]. To evaluate the precision and performance of IntraJ, we implemented two client data flow analyses, one forward and one backward, namely *Null Pointer Exception* analysis and *Dead Assignment* analysis, respectively.

More precisely, our contributions are as follows:

- We present IntraCFG, a modular and precise language-independent framework for intraprocedural CFG construction, implemented using RAGs.
- We present IntraJ, an application of the framework to construct concise and precise CFGs for Java 7. We discuss design decisions for what facts to include, and how to reify implicit facts that the AST does not expose directly.
- We provide two different client analyses to validate and evaluate the framework: *Dead Assignment* analysis, which detects unnecessary assignments, and *Null Pointer Exception* analysis, which detects if there exists a path in which a `NullPointerException` can be thrown.

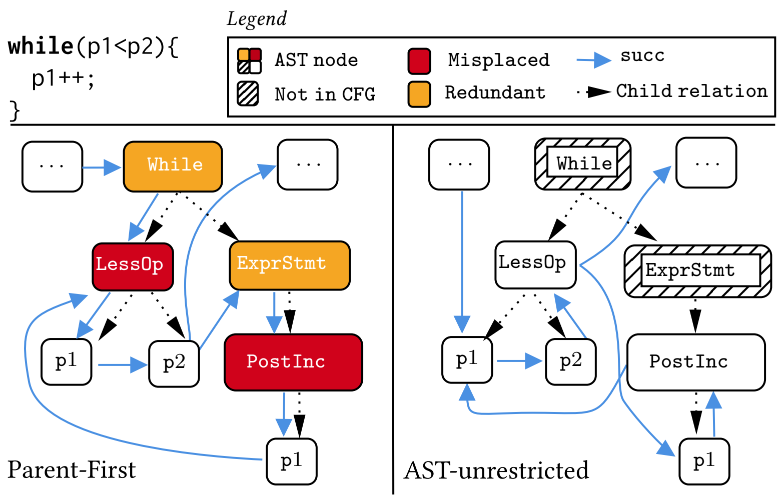


Figure 1: In the Parent-First CFG (left) a parent always precedes its children, resulting in redundant and misplaced nodes. The AST-unrestricted CFG (right) is correct and minimal.

- We provide an evaluation of performance and precision for a number of Java subject applications, and compare performance and precision both to the earlier RAGs-based approach and to SonarQube, a current mainstream program analysis tool.

In the rest of this paper, we review RAGs and introduce IntraCFG (Section 2) and IntraJ, along with underlying design decisions and implementation details (Section 3), present our client analyses (Section 4) and evaluation (Section 5), discuss related work (Section 6) and conclude (Section 8).

2 RAGs and the IntraCFG framework

Attribute grammars, originally introduced by Knuth [Knu68], are declarative specifications that decorate AST nodes with attributes. Each AST node type can declare attributes and define their values through equations. There are two main kinds of attributes: ***synthesised attributes***, defined in the same node, and ***inherited attributes***, defined in a parent or an ancestor node. Synthesised attributes are useful for propagating information upwards in an AST, e.g. for basic type analysis of expressions. Inherited attributes are useful for propagating information downwards, e.g., for environment information.

Reference Attribute Grammars (RAGs) [Hed00] extend Knuth's attribute grammars with ***reference attributes***, whose values are references to other AST nodes. Attributes that compute references to AST nodes can declaratively construct graphs that are superimposed on the AST, e.g., CFGs, so that RAGs can propagate information directly along these graph references.

For our implementation, we have used the JastAdd metacompilation system [HM03], which supports RAGs as well as the following attribute grammar extensions that we use here:

Higher-order attributes (HOAs) [VSK89b] have a value that is a fresh AST subtree, which can itself have attributes. HOAs are useful for reifying implicit structures not available in the AST constructed by the parser. We use HOAs to reify, for example, control flow for unchecked exceptions and implicit `null` assignments.

Circular attributes are attributes whose equations may transitively depend on their own values [MH07b]. They support declarative fixpoint computations and can e.g. express data flow properties on top of a CFG.

Collection attributes are attributes that aggregate any number of *contributions* from anywhere in the AST, or from a bounded AST region [MEH07a]. They simplify e.g. error reporting and the computation of the predecessor relation from the successor relation in a CFG.

Node type interfaces are similar to Java interfaces and can be mixed into AST node types. They declare e.g. attributes and equations, and enable language-independent plugin components in attribute grammars [FSH20].

Attribution aspects are modules that use inter-type declarations to declare a set of attributes, equations, collection contributions, etc. for specific node types [HM03], and mix in interfaces to existing node types. They provide a modular extension mechanism for RAGs.

On-demand evaluation, where attributes are evaluated only if they are used, and with optional caching that prevents reevaluation of attributes used more than once [Jou84]. JastAdd exclusively uses this evaluation strategy.

2.1 RAG frameworks for control flow

Our work is the second to construct CFGs in a RAG framework, following the earlier JASTADDJ-INTRAFLOW [Söd+13a]. JASTADDJ-INTRAFLOW constructs *Parent-First* CFGs, in the sense that all AST nodes involved in the CFG computation are also part of the CFG and impose their nesting structure, so that the CFG must always pass through all of a node's ancestors before it can reach the node itself. By contrast, our INTRACFG framework is *AST-unrestricted*, in that the resulting CFG need not follow the syntactic nesting structure.

Figure 1 illustrates this difference between the two approaches for a `while` loop in Java. The left (Parent-First) CFG from JASTADDJ-INTRAFLOW first flows through the `While` node to reach the loop condition. However, the CFG already encodes the flow properties of `While`, so this flow is unnecessary for data flow analysis. The same holds for `ExprStmt`. We therefore consider these nodes *redundant* for the CFG. By contrast, our system's AST-unrestricted CFG on the right skips these two nodes entirely.

The second, more severe concern is that the control flow in the left CFG in Figure 1 cannot follow Java's evaluation order due to the Parent-First constraint: flow passes through the `PostUnaryInc` node, which represents an update, before passing through the node's subexpression `p1`. This flow would represent an inversion of the actual order of evaluation: the nodes are *misplaced* in the CFG. Typical client analyses on such a flawed CFG must add additional checks to compensate or otherwise sacrifice soundness or precision in programs where `p1` also has a side effect. By contrast, our AST-unrestricted CFG on the right addresses this limitation and accurately reflects Java's control flow.

We note that recent work on program analysis [Sza; Hel+20] has asserted that attribute grammars restrict computations to be tightly bound to the AST structure. Our work demonstrates that this generalization does not hold, and that RAGs are an effective framework for efficiently deriving precise CFGs that deviate from the AST structure and for expressing client analyses directly in terms of such derived structures.

2.2 The IntraCFG framework

IntraCFG is our new RAG framework for constructing intraprocedural AST-unrestricted CFGs, superimposing the graph on the AST. Figure 2 shows the framework as a UML class diagram. IntraCFG is language-independent, and includes interfaces that AST types in an abstract grammar can mix in and specialise to compute the CFG for a particular language. The figure shows five types: the `CFGRoot` interface is intended for subroutines, e.g., methods and constructors, to represent a local CFG with a unique entry and exit node. We represent the latter as synthetic AST node types `Entry` and `Exit`. The `CFGNode` interface marks nodes in the CFG, and each node has reference attributes succ and pred to represent the successor and predecessor edges. The `CFGSupport` interface marks AST nodes in a location that may contain `CFGNode`s. All `CFGNode`s are `CFGSupport` nodes, but `CFGSupport` nodes that are not `CFGNode`s can help steer the construction of the CFG.

Figure 2 also shows the AST node types' attributes and their types (middle boxes), as well the defining equations (bottom boxes). Here, we write $\mathcal{P}$(`CFGNode`) for the type of sets over `CFGNode`s. We optionally prefix attribute names with $\uparrow$, $\downarrow$, $\rightarrow$, $\square$, or $\circlearrowright$ to highlight the AST traversal underlying their computation. For the different kinds of attributes, we use the following equations, for attributes x

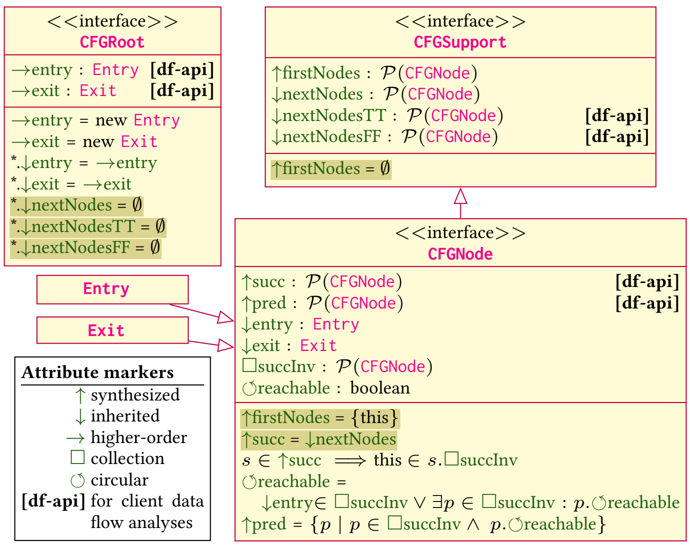


Figure 2: The IntraCFG framework with interfaces CFGRoot, CFGSupport, CFGNode, and synthetic AST types Entry, Exit. Highlighted attribute equations are default equations, intended for overriding.

and expressions e:

Synthesised attributes: $\uparrow$x $=$ e defines attribute $\uparrow$x for the local AST node (which we call this).

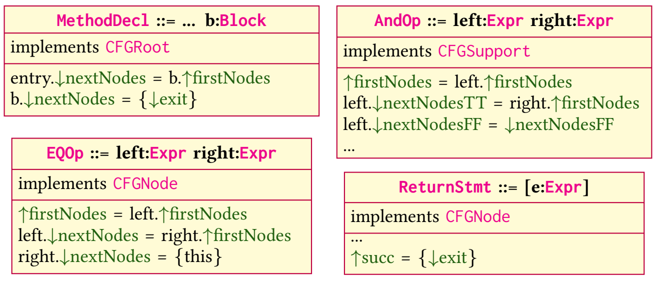
Inherited attributes: $c.\downarrow$x $=$ e gives AST child node c and its descendants access to e through $\downarrow$x, where e is evaluated in the context of the this node (c's parent). We use the wildcard $*$ for c to broadcast to all children, $*.\downarrow\text{x} = e$.

Higher-order attributes: $\rightarrow\text{x} = e$ where e must construct a fresh AST subtree.

Circular attributes: ↻x $=$ e, where e computes a fixpoint. In this paper, boolean circular attributes start at *false* and monotonically grow with $\vee$, while set-typed circular attributes start at $\emptyset$ and monotonically grow with $\cup$.

Collection attributes have no equations, but *contributions*. We write $P \implies e \in n.\square$x to contribute the value of expression e to collection attribute $\square$x in node n if P holds. In this paper, all collection attributes are sets.

This pseudocode translates straightforwardly to more verbose JastAdd code



```
MethodDecl ::= … b:Block
implements CFGRoot
entry.↓nextNodes = b.↑firstNodes
b.↓nextNodes = {↓exit}

AndOp ::= left:Expr right:Expr
implements CFGSupport
↑firstNodes = left.↑firstNodes
left.↓nextNodesTT = right.↑firstNodes
left.↓nextNodesFF = ↓nextNodesFF
…

EQOp ::= left:Expr right:Expr
implements CFGNode
↑firstNodes = left.↑firstNodes
left.↓nextNodes = right.↑firstNodes
right.↓nextNodes = {this}

ReturnStmt ::= [e:Expr]
implements CFGNode
…
↑succ = {↓exit}
```

Figure 3: Example application of the IntraCFG framework.

```
EQOp ::= Left:Expr Right:Expr; // Abstract grammar
EQOp implements CFGNode;
eq EQOp.firstNodes() = getLeft().firstNodes();
eq EQOp.getLeft().nextNodes() = getRight().firstNodes();
eq EQOp.getRight().nextNodes() =
    SmallSet<CFGNode>.singleton(this);
```

Listing 1: JastAdd translation of EQOp in Figure 3.

that uses Java for the right-hand sides in our equations. IntraCFG is 45 LOC of JastAdd code.[1]

The equations in the framework define some of the attributes, and provide default definitions for others. To specialise the framework to a particular language, the default equations can be overridden for specific AST node types to capture the control flow of the language.

Client analyses can then use attributes marked as **[df-api]** in Figure 2, such as, ↑succ and ↑pred, to analyze the CFG. Since CFG nodes are also AST nodes, it is easy for these analyses to also access syntactic information and attributes from, e.g., type analysis, as we illustrate in Section 4.

2.3 Computing the successor attributes

To compute the ↑succ attributes, we use the helper attributes ↑firstNodes and ↓nextNodes. Given an AST subtree t, its ↑firstNodes contain the first CFGNode *within or after* t that should be executed, if such a node exists. If not, ↑firstNodes is empty. The framework in Figure 2 shows the default definitions for this attribute: the empty set for a CFGSupport node, and the node itself for a CFGNode.

[1] https://github.com/lu-cs-sde/IntraJSCAM2021/

The ↓nextNodes attribute contains the `CFGNode`s that are *outside* t, and that would immediately follow the last executed `CFGNode` within t, disregarding abrupt execution flow like returns and exceptions. By default, the ↑succ attribute is defined as equal to ↓nextNodes.

Figure 3 shows how the framework can be specialised to some example AST node types to define the desired CFG. JastAdd expresses these additions in a modular attribution aspect. For illustration, we again encode the JastAdd specification into UML and include the abstract syntax of each node type. Listing 1 also illustrates how the pseudocode can be translated to JastAdd code.

Here, `MethodDecl` exemplifies a `CFGRoot`. It defines the flow between its →entry and →exit HOAs and its children. `EQOp` exemplifies a `CFGNode`. It defines a pre-order flow: `left`, then `right`, then the node itself. Each type defines its own synthesised attributes as well as the inherited attributes of its children and HOAs.

All `CFGNode`s have immediate access to the `Entry` and `Exit` nodes of the CFG, through the inherited ↓entry and ↓exit attributes declared in `CFGNode` and defined by the nearest `CFGRoot` ancestor (Figure 2). This allows e.g., the `ReturnStmt` to point its ↑succ edge directly to the `Exit` node.

For boolean expressions that affect control-flow, IntraCFG supports path-sensitive analysis, splitting the successor set into two disjoint sets for the *true* and *false* branches. We provide attributes ↓nextNodesTT and ↓nextNodesFF, respectively, to capture these branches. The `AndOp` type illustrates how these attributes can capture short-circuit evaluation on the left child. These attributes are relevant only for boolean branches, and must ensure the following property:

$$\downarrow\text{nextNodesTT} \cup \downarrow\text{nextNodesFF} = \downarrow\text{nextNodes}$$

Figure 4 illustrates these attributes on a small program in a language with methods, statements, and expressions. Here, `MethodDecl` is a `CFGRoot` and thus automatically has fresh `Entry` and `Exit` nodes. Nodes in the control flow, e.g., identifiers and the equality-check operator, `EQOp`, are `CFGNode`s, and thus have the ↑succ attribute. Nodes that do not belong to the control-flow but live in AST locations below a `CFGRoot` that may contribute to control flow are `CFGSupport` nodes. The left-hand-side variable of the assignment `p1 = 0` (i.e., $\mathtt{p}_1$) is not part of the flow (cf. Section 3.1).

2.4 Computing predecessors

To support both forward and backward analyses, we provide a predecessor attribute that captures the inverse of the successor attribute ↑succ. However, ↑succ is also defined for `CFGNode`s that are not reachable from `Entry` by following ↑succ (i.e., that are "dead code"). Our framework therefore computes predecessor edges ↑pred by not only inverting ↑succ into a collection attribute □succInv, but also by filtering out such "dead" nodes from □succInv with a boolean circular attribute ↻reachable (Figure 2).

```
void foo(int p1, int p2, boolean b1){
  if (p1==p2 && b1) p1 = 0;
}
```

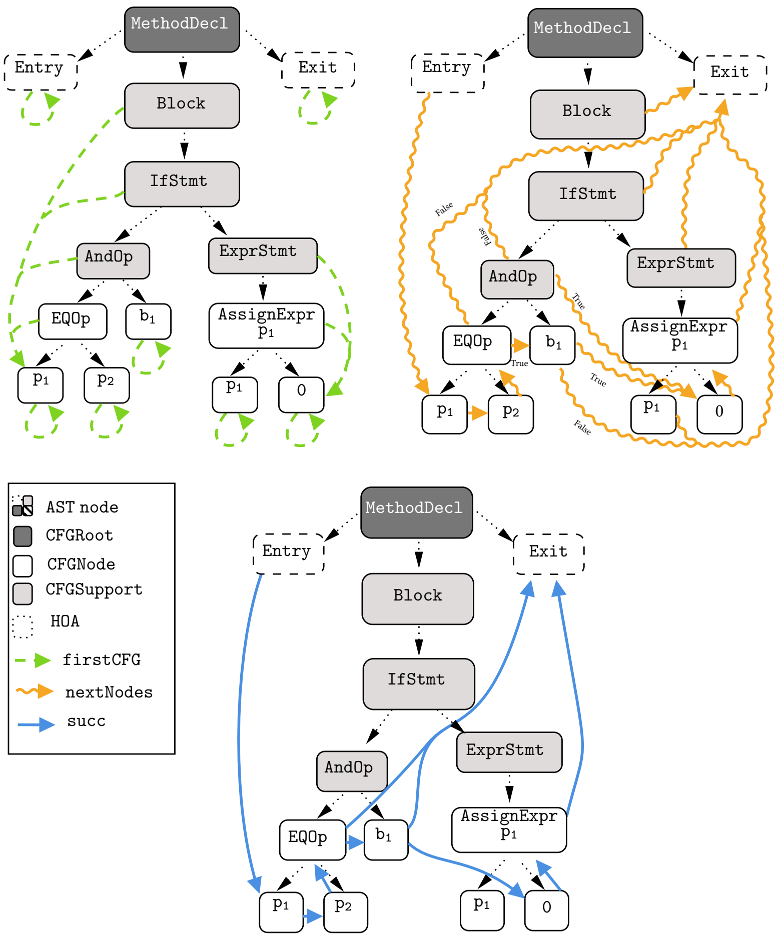


Figure 4: Visualization of the attributes ↑firstNodes, ↓nextNodes and ↑succ. For boolean expressions (AndOp and EQOp), the subsets ↓nextNodesTT and ↓nextNodesFF are shown instead of ↓nextNodes, marked by True and False, respectively.

3 IntraJ: IntraCFG implementation for Java 7

IntraJ is our implementation of a precise intraprocedural CFG for Java 7, extending the IntraCFG framework and the ExtendJ Java compiler. IntraJ exploits the ExtendJ front-end, which performs name-, type-, and compile-time error analysis. ExtendJ produces an attributed AST[2] on top of which IntraJ superimposes the CFG.

In this Section, we discuss the most important design decisions for IntraJ, and in particular, how we used HOAs to improve the precision of the CFG. Our two main goals were:

1. minimality: build a concise CFG by excluding AST nodes that do not correspond to any runtime action. This improves client analysis performance, in particular for fixpoint computations.

2. high precision: the constructed CFGs should capture most program details. We exploit HOAs to reify implicit structures in the program, such as calls to static and instance initialisers and implicit conditions in `for` loops.

We gave particular importance to exceptions, modelling them as accurately as possible and weighing the trade-off between precision and minimality.

IntraJ consists of a total of 989 LOC (598 for Java 4; 11 for Java 5; 380 for Java 7). We have constructed a systematic benchmark test suite for IntraJ, consisting of 151 tests in total (126 for Java 4; 5 for Java 5; 20 for Java 7). The test suite reads source code as input and produces CFGs as `dot` files as output. We validated the result of each test manually.

3.1 Statements and Expressions

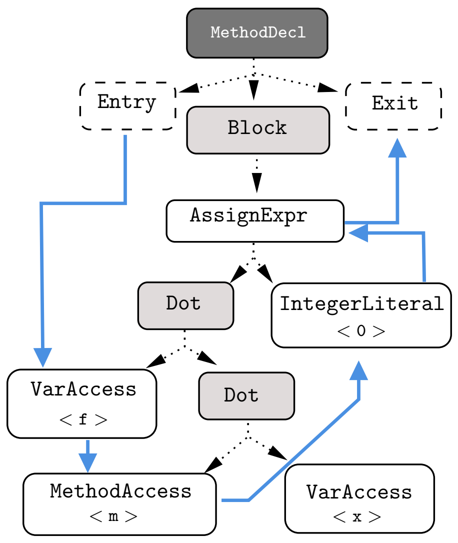
When a language implementer specialises IntraCFG for a given language, they must decide which AST nodes should be part of the CFG, i.e., mix in (implement) the `CFGNode` interface. As a general design principle, we included AST nodes that correspond to a single action at runtime. This includes operations on values, like additions, comparisons, and read operations on variables and fields.

We also included nodes that are interesting points in the execution that a client analysis might want to use. This includes nodes that redirect flow outside of the CFG, like method calls, return statements, and throw statements.

For assignments, the choice of nodes to include in the CFG was not obvious. The left-hand side of an assignment can be a chain of named accesses and method calls, e.g., `f.m().x`, with the rightmost named access, `x`, corresponding to the write. Here, we chose to not include `x` in the CFG but instead use the assignment node itself to represent the write operation, see Figure 5. We argue that this gives

[2]The full abstract grammar for Java 7 can be found at `https://extendj.org`

a simpler client interface, since the same AST node type, VarAccess, otherwise represents all named accesses on the left- and right-hand side of an assignment.



```
void foo() {
  f.m().x = 0;
}
```

We represent the write to x by the AssignExpr node in the CFG.

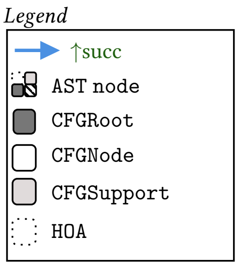
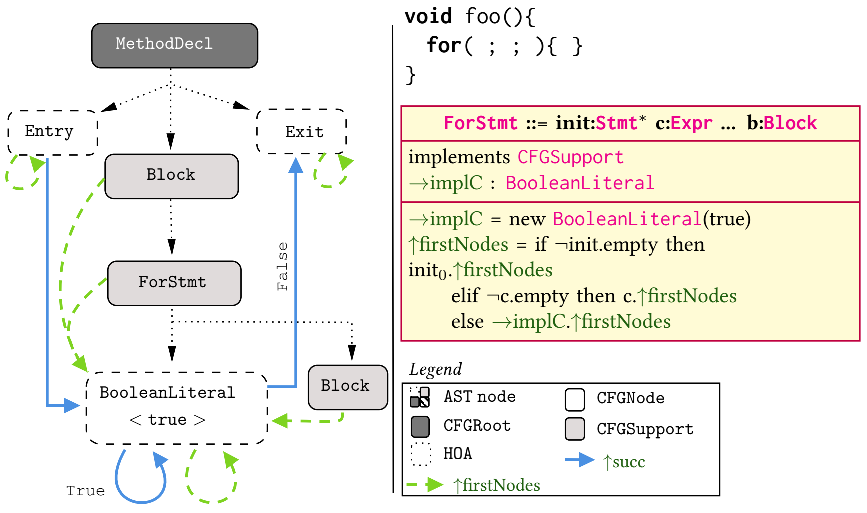


Figure 5: An assignment with a complex left-hand side.

We do not include purely structural nodes, like Block or type information nodes, in the CFG. We also exclude nodes that redirect internal flow, like `while` statements and conditionals. While these nodes do represent runtime actions, the CFG already reflects their flow through successor edges.

MethodDecl and the analogous ConstructorDecl for constructors mix in the CFGRoot interface, thus representing a local CFG. A CFGSupport node defines the inherited attributes for its CFGNodes children, if any. For example, a Block defines the ↓nextNodes attribute for all its children.

As an example of the flexibility of IntraCFG, consider the Java ForStmt, which is composed of variable initialisation, termination condition, post-iteration instruction, and loop body. The CFG should include a loop over these components. However, it is legal to omit all the components, i.e., to write: '`for ( ; ; ){}`'. The condition is implicitly `true` in this case, resulting in an infinite loop. To construct a correct CFG, we still need a node to loop over; we therefore opt to reify this implicit condition. We construct an instance of the boolean literal `true` as the HOA →implC. Figure 6 shows how the ↑firstNodes attribute then uses →implC only if both the initialisation statements and the condition are missing.



```
void foo(){
  for( ; ; ){ }
}
```

```
ForStmt ::= init:Stmt* c:Expr ... b:Block
implements CFGSupport
→implC : BooleanLiteral
→implC = new BooleanLiteral(true)
↑firstNodes = if ¬init.empty then
init_0.↑firstNodes
      elif ¬c.empty then c.↑firstNodes
      else →implC.↑firstNodes
```

Figure 6: CFG for method with empty *ForStmt*. The HOA →implC reifies the implicit *true* condition.

Another interesting corner case is the EmptyStmt. This node represents e.g. the semicolon in the trivial block {;}. The EmptyStmt is a CFGSupport node since it does not map to a runtime action. Since EmptyStmt has no children, its ↑firstNodes will be the following CFG node. We achieve this by defining ↑firstNodes as equal to ↓nextNodes, overriding the default equation from CFGSupport. In this manner, the CFG skips the EmptyStmt, and if there are occurrences of multiple EmptyStmts, we skip them transitively and link to the next concrete CFGNode. The example in Figure 7 shows how we exclude two EmptyStmts from the CFG and obtain a CFG with only a single edge from method Entry to Exit. Let us call the two EmptyStmts e_1 and e_2, from left to right. The equations give that Entry.↑succ = Exit since

$$\begin{aligned}
\texttt{Entry}.\uparrow\text{succ} &= \texttt{Entry}.\downarrow\text{nextNodes} = \texttt{Block}.\uparrow\text{firstNodes} \\
&= \quad e_1.\uparrow\text{firstNodes} \quad = e_1.\downarrow\text{nextNodes} \\
&= \quad e_2.\uparrow\text{firstNodes} \quad = e_2.\downarrow\text{nextNodes} \\
&= \texttt{Block}.\downarrow\text{nextNodes} = \texttt{Exit}
\end{aligned}$$

3.2 Static and Instance Initialisers

When a Java program accesses or instantiates classes, it executes static and instance initialisers. We will use the example in Figure 8 to explain how we handle initialisers. As seen in the example, static and instance initialisers can be syntactically interleaved: The instance field foo is followed by the static field bar,

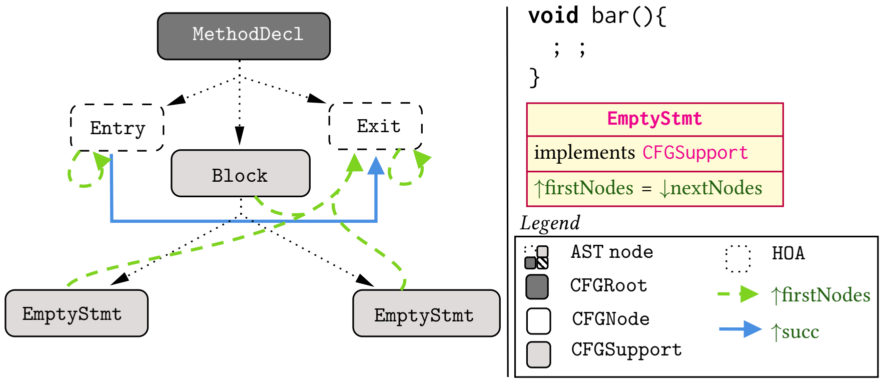


Figure 7: The CFG can entirely skip AST nodes.

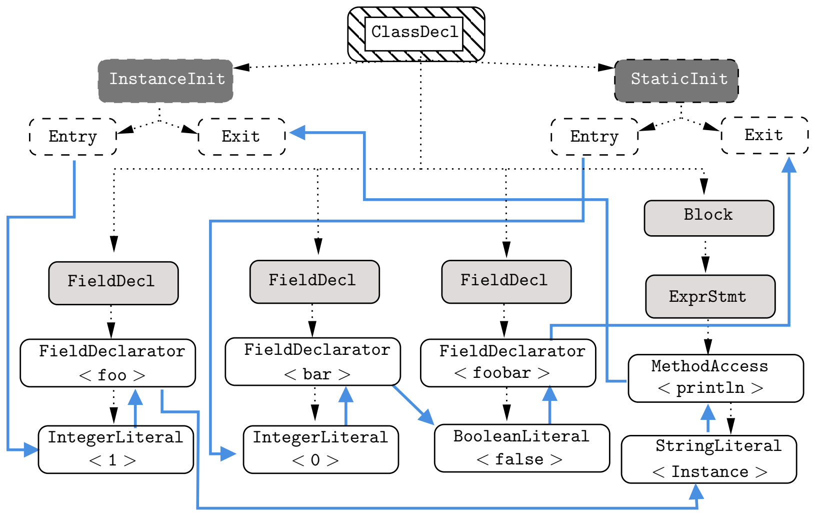
another static field `foobar`, and by an instance initialiser block printing the string `"Instance"`.

The Java Language Specification specifies that when a class is instantiated, the static initialisers are executed first (unless already executed), then the instance initialisers, and finally the constructor. During the execution of the static initialisers, the ones in a superclass are executed before those in a subclass, and similarly for the instance initialisers.

To handle this execution order, our solution is to use HOAs to construct two independent CFGs for each `ClassDecl`: one for the static initialisations, →staticInit, and one for the instance initialisations, →instanceInit. The →staticInit connects all the static field declarations and all static initialisers. →instanceInit analogously connects instance fields and initialisers. →instanceInit and →staticInit mix in the `CFGRoot` interface, and automatically get `Entry` and `Exit` nodes. The equations for ↑firstNodes and ↓nextNodes are overridden to include the initialisers in the same order as they appear in the source code. To connect the initialisation CFGs, we view them as implicit methods and use HOAs to insert implicit method calls to them. For example, if a class has a superclass, the implicit static/instance initialiser method will start by calling the corresponding initialiser in the superclass.

3.3 Exceptions Modelling

Control flow for exceptions is complex to model and often requires non-trivial approximations [Ami+16; JC04; Cho+99]. In Java, there are two kinds of exceptions: *checked* and *unchecked*. If an expression can throw a checked exception, then Java's static semantics require that the method that contains this expression must surround the expression with an exception handler, or declare the excep-



```
public class A {
  int foo = 1; //instance field
  static int bar = 0; //static field
  static boolean foobar = false; //static field
  { println("Instance"); } //instance initialiser
}
```

Figure 8: Example of class that interleaves static and instance initialisers. The →instanceInit and →staticInit HOAs represent the CFGs for each kind of initialisers.

tion in the method signature (using the `throws` keyword). If the exception is unchecked, it is optional for the method to handle or declare the exception. Some methods still declare unchecked exceptions, possibly to increase readability or to follow coding conventions.

For the IntraJ CFG, we decided to explicitly represent all *checked* exceptions, and, in addition, all *unchecked* exceptions that are explicitly thrown in the method or declared in the method signature. For unchecked exceptions, we represent only those that may escape from a `try-catch` statement. Within the `try` block of such a statement, we introduce individual CFG edges for each represented exception whenever it may be thrown, and separate edges for regular (non-exceptional) control flow. This design allows us to avoid conservative overapproximation, and enables client analyses to distinguish whether control reached a `finally` block through exceptional control flow or through regular control flow.

Consider the following example with two nested `try` blocks:

```
void ex(Long x) throws Exn {
  try {
    try {
      if (x < 10)          NPE
        array[x] = 0;      OOB
      else throw new Exn(); Exn
      return;              R
    } finally     { . . . } F1
  } catch (Exn e) { . . . } CExn
  catch (Alt e)   { . . . } CAlt
  finally         { . . . } F2
}
```

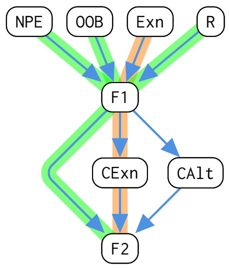


Figure 9: Complex exception flow in a conservative CFG. Only the flow paths in green and orange are realisable.

Calling `ex(null)` from Figure 9 triggers a null pointer exception at NPE. Control then flows from the exception to the first and then to the second `finally` block, NPE→F1→F2. Calling `ex(-1)` similarly triggers an out-of-bounds exception at OOB, with analogous flow. The explicit exception at Exn takes the path Exn→F1→CExn→F2, and no path can go through CAlt assuming that F1 does not throw `Alt`. Note that `finally` also affects `break`, `continue`, and `return`, as we see in the path R→F1→F2.

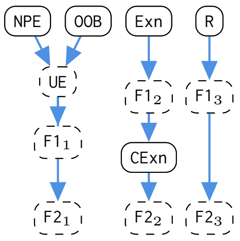


Figure 10: Path-sensitive variant of the CFG from Figure 9, used in IntraJ.

If we represent the CFG as on the right in Figure 9, client analyses will process many unrealisable paths, such as R→F1→CAlt→F2. Instead, we exploit an existing feature in ExtendJ, originally intended for code generation [Öqv18], that clones `finally` blocks. We incorporate the HOAs that represent each cloned block into our CFG. In our example, this yields the CFG from Figure 10, and leaves CAlt as dead code.

This path sensitivity heuristic gives us increased precision in exception handling and resource cleanup code, which in our experience is often more subtle and less well-tested than the surrounding code. For unchecked exception edges (NPE, OOB), we follow Choi et al. [Cho+99], who observe that these edges are '*quite frequent*'; we therefore funnel control flow for these exceptions through a single node UE in the style of Choi et al.'s factorised exceptions. Each `try` block provides one such node through a HOA. Section 5 shows some of the practical strengths and weaknesses of our heuristic.

We take an analogous approach for `try`-with-resources, which automatically releases resources (e.g., closes file handles) in the style of an implicit `finally` block. Our treatment differs from that of `finally` only in that we synthesise the implicit code and suitably chain it into the CFG.

4 Client Analysis

We demonstrate our framework with two representative data flow analyses: *Null Pointer Exception* Analysis (NPA), a forward analysis, and *Live Variable* Analysis (LVA), a backward analysis that helps detect useless ('dead') assignments. These analyses are significant for bug checking and therefore benefit from a close connection to the AST.

We first recall the essence of these algorithms on a minimal language that corresponds to the relevant subset of Java:

$$\begin{array}{lcl} \mathbf{e} \in E & ::= & \mathsf{new()} \mid \mathsf{null} \mid \mathit{id} \mid \mathit{id}.\mathsf{f} \mid \mathit{id} = E \\ \mathbf{v} \in \mathit{id} & ::= & \mathsf{x}, \ldots \end{array}$$

An expression e can be a `new()` object, `null`, the contents of another variable, the result of a field dereference (`x.f`), or an assignment `x = e`. The values in our language are an unbounded set of objects O and the distinct `null`. Expressions have the usual Java semantics. Since IntraJ already captures control flow (on top of IntraCFG) and name analysis (via ExtendJ), we can ignore statements and declarations, and safely assume that each *id* is globally unique.

4.1 Null Pointer Exception Analysis

In our simplified language, a field access `x.f` fails (in Java: throws a Null Pointer Exception) if `x` is `null`. Null Pointer Exception Analysis (NPA) detects whether a given field dereference *may* fail (e.g. in the SonarQube NPA variant) or *must* fail (e.g. in the Eclipse JDT NPA variant) and can alert programmers to inspect and correct this (likely) bug.

In our framework, writing *may* and *must* analyses requires the same effort; we here opt for a *may* analysis over a binary lattice $\mathcal{L}_2$ in which $\top$ = **nully** signifies *value may be* `null` and $\bot$ = **nonnull** signifies *value cannot be* `null`.

More precisely, we use a product lattice over $\mathcal{L}_2$ that maps each *access path* $a \in \mathcal{A}$ (e.g. `x`; `x.f`; `x.f.f`; ...) to an element of $\mathcal{L}_2$. Our analysis then follows the usual approach for a join data flow analysis [CC77]. Our monotonic transfer function $f_{NPA} : (\mathcal{A} \to \mathcal{L}_2) \times E \to (\mathcal{A} \to \mathcal{L}_2)$ is straightforward:

$$\begin{array}{rcl} f_{NPA}(\Gamma, \mathbf{v} = \mathbf{e}) & = & \Gamma[\mathbf{v} \mapsto [\![\mathbf{e}]\!]^{\Gamma}] \\ \text{where} \quad [\![\mathsf{new()}]\!]^{\Gamma} & = & \textbf{nonnull} \\ [\![\mathsf{null}]\!]^{\Gamma} & = & \textbf{nully} \\ [\![\mathbf{v}]\!]^{\Gamma} & = & \Gamma(\mathbf{v}) \\ [\![\mathbf{v}.\mathsf{f}]\!]^{\Gamma} & = & \Gamma(\mathbf{v}.\mathsf{f}) \\ [\![\mathbf{v} = \mathbf{e}]\!]^{\Gamma} & = & [\![\mathbf{e}]\!]^{\Gamma} \end{array}$$

We do not need to write a recursive transfer function for assignments nested in other assignments (e.g., `x = y = z`), since the CFG already visits these in evaluation order.

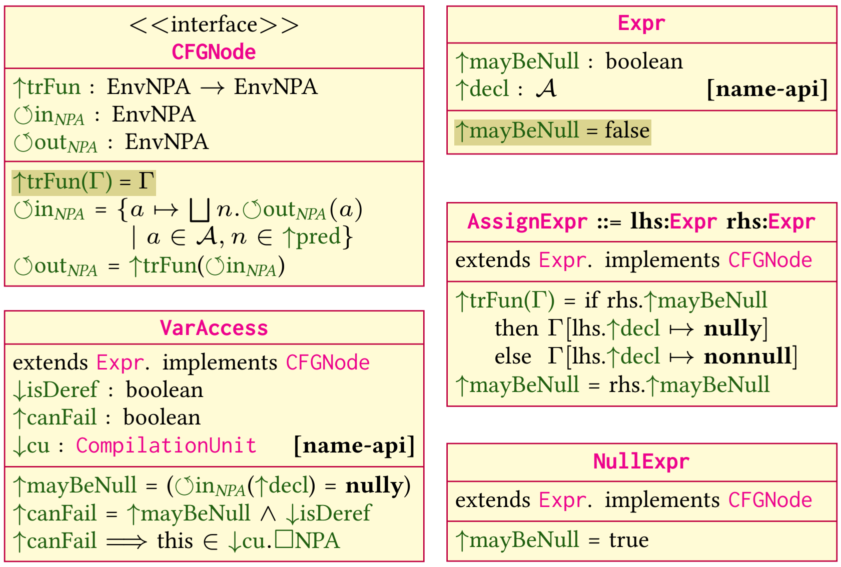

<<interface>> CFGNode
$\uparrow$trFun : EnvNPA $\to$ EnvNPA
$\circlearrowleft$in$_{NPA}$: EnvNPA
$\circlearrowleft$out$_{NPA}$: EnvNPA
$\uparrow$trFun(Γ) = Γ
$\circlearrowleft$in$_{NPA}$ = $\{a \mapsto \bigsqcup n.\circlearrowleft\text{out}_{NPA}(a) \mid a \in \mathcal{A}, n \in \uparrow\text{pred}\}$
$\circlearrowleft$out$_{NPA}$ = $\uparrow$trFun($\circlearrowleft$in$_{NPA}$)
VarAccess
extends Expr. implements CFGNode
$\downarrow$isDeref : boolean
$\uparrow$canFail : boolean
$\downarrow$cu : CompilationUnit **[name-api]**
$\uparrow$mayBeNull = ($\circlearrowleft$in$_{NPA}$($\uparrow$decl) = **nully**)
$\uparrow$canFail = $\uparrow$mayBeNull $\wedge$ $\downarrow$isDeref
$\uparrow$canFail $\Longrightarrow$ this $\in$ $\downarrow$cu.$\square$NPA
Expr
$\uparrow$mayBeNull : boolean
$\uparrow$decl : $\mathcal{A}$ **[name-api]**
$\uparrow$mayBeNull = false
AssignExpr ::= **lhs**:Expr **rhs**:Expr
extends Expr. implements CFGNode
$\uparrow$trFun(Γ) = if rhs.$\uparrow$mayBeNull
then Γ[lhs.$\uparrow$decl $\mapsto$ **nully**]
else Γ[lhs.$\uparrow$decl $\mapsto$ **nonnull**]
$\uparrow$mayBeNull = rhs.$\uparrow$mayBeNull
NullExpr
extends Expr. implements CFGNode
$\uparrow$mayBeNull = true

Figure 11: Partial implementation of our NPA. We obtain $\uparrow$decl and $\downarrow$cu from ExtendJ's name analysis API.

Our implementation is field-sensitive and control-sensitive (i.e., it understands that `if (x != null){x.f=1;}` is safe), but array index-insensitive and alias-insensitive. Field sensitivity is reached by considering the entire access path chain, while control sensitivity is given by defining new HOAs representing implicit facts, e.g., `x != null`.

Figure 11 shows how we compute environments $\Gamma \in \text{EnvNPA} = \mathcal{A} \to \mathcal{L}_2$ that capture access paths that may be `null` at runtime. We extend `CFGNode` with $\circlearrowleft$in$_{NPA}$, which merges all evidence that flows in from control flow predecessors, and $\circlearrowleft$out$_{NPA}$, which applies the local transfer function $\uparrow$trFun to $\circlearrowleft$in$_{NPA}$. While NPA is a forward analysis, JastAdd's on-demand semantics mean that we query *backwards*, following $\uparrow$pred edges, when we compute $\circlearrowleft$in$_{NPA}$ on demand. $\circlearrowleft$in$_{NPA}$ and $\circlearrowleft$out$_{NPA}$ are circular, i.e., can depend on their own output and compute a fixpoint.

The attributes for `VarAccess` show how we use this information. Each `VarAccess` contributes to $\downarrow$cu.$\square$NPA, the compilation unit-wide collection attribute of likely `null` pointer dereferences, whenever $\uparrow$mayBeNull holds and when the `VarAccess` is also a proper prefix of an access path and must therefore be dereferenced ($\downarrow$isDeref, not shown here).

Our full Java 7 implementation takes up 142 lines of JastAdd code, excluding data structures but including control sensitive analysis handling and reporting.

4.2 Live Variable Analysis

Given a `CFGNode` `n`, a variable is *live* iff there exists at least one path from `n` to `Exit` on which `n` is read without first being redefined. An assignment to a variable that is not live (i.e., *dead*) wastes time and complicates the source code, which generally means that it is a bug [Rei21]. We can detect this bug with *Live Variable/Liveness* analysis (LVA), a data flow analysis that computes the live variables for each CFG node.

We express LVA as a Gen/Kill analysis, on the powerset lattice over the set of *live* (local) variables. Each transfer function adds variables to the set (marks them *live*) or removes them (marks them *dead*). LVA is a *backward* analysis, starting at the *Exit* node with the assumption that all variables are dead (i.e., with the set of live variables $L = \emptyset$). The transfer function thus maps from node exit to entry and has the form:

$$f_{LVA}(L, \mathbf{e}) = (L \setminus \mathit{def}(\mathbf{e})) \cup \mathit{use}(\mathbf{e})$$

where $\mathit{def}(\mathbf{e})$ is the set of variables that $\mathbf{e}$ assigns to, and $\mathit{use}(\mathbf{e})$ is the set of variables that $\mathbf{e}$ reads.

We encode the f_{LVA} using RAGs in a similar way as done in [Söd+13a]: Figure 12 shows our computation where circular attributes $\circlearrowleft\mathrm{in}_{LVA}$ and $\circlearrowleft\mathrm{out}_{LVA}$ represent variables live before/after a `CFGNode`. Here, $\circlearrowleft\mathrm{out}_{LVA}$ reads from $\uparrow$succ nodes, since we are implementing an on-demand backward analysis. `VarAccess` and `AssignExpr` override $\uparrow$use and $\uparrow$def, respectively. Since the CFG traverses through the right-hand side of each assignment, this specification suffices to capture the analysis of our Java language fragment. Our full implementation for Java 7 takes up 38 lines of code.

4.3 Dead Assignment Analysis

We use dead assignment analysis (DAA) as a straightforward client analysis for LVA. Our implementation of DAA refines the results of LVA with a number of heuristics that we have adopted from the SonarQube checker. Specifically, these heuristics suppress warnings in code like the following:

```
String status = ""; // WARNING: unused assignment
if (...) status = "enabled";
else status = "disabled";
```

Here, the initial assignment to `status` reflects a defensive coding pattern that ensures that all variables are initialised to some safe default. We (optionally) suppress warnings like the above under two conditions: (1) the assignment must be in a variable initialisation, and (2) the initialiser must be a *common default value*, i.e., one of {`null`, `1`, `0`, `-1`, `""`, `true`, `false`}. Our DAA implementation takes up 62 lines of code.

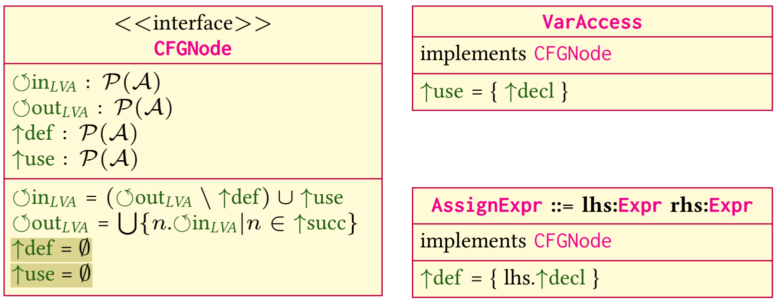

<<interface>>
CFGNode
$\circlearrowleft \text{in}_{LVA} : \mathcal{P}(\mathcal{A})$
$\circlearrowleft \text{out}_{LVA} : \mathcal{P}(\mathcal{A})$
$\uparrow\text{def} : \mathcal{P}(\mathcal{A})$
$\uparrow\text{use} : \mathcal{P}(\mathcal{A})$
$\circlearrowleft \text{in}_{LVA} = (\circlearrowleft \text{out}_{LVA} \setminus \uparrow\text{def}) \cup \uparrow\text{use}$
$\circlearrowleft \text{out}_{LVA} = \bigcup\{n.\circlearrowleft \text{in}_{LVA} \mid n \in \uparrow\text{succ}\}$
$\uparrow\text{def} = \emptyset$
$\uparrow\text{use} = \emptyset$
VarAccess
implements CFGNode
$\uparrow\text{use} = \{ \uparrow\text{decl} \}$
AssignExpr ::= lhs:Expr rhs:Expr
implements CFGNode
$\uparrow\text{def} = \{ \text{lhs}.\uparrow\text{decl} \}$

Figure 12: Partial implementation of our LVA.

5 Evaluation and Results

We demonstrate the utility of IntraCFG and IntraJ[3] by evaluating the client analyses that we describe in Section 4 against similar source-level analyses from the Parent-First framework JASTADDJ-INTRAFLOW[4] (JJI) and the commercial static analyser SonarQube, version 8.9.0.43852 (SQ).

Our evaluation targets DaCapo benchmarks ANTLR, FOP, and PMD [Bla+06], as well as JFreeChart (JFC), which is a superset of the Chart benchmark. These benchmarks mostly subsume the ones used by JJI [Söd+13a], except for replacing Bloat by the more readily available and larger PMD. Table 1 summarise key metrics for the benchmarks and compares CFGs against JJI. Here, IntraJ's AST-unrestricted strategy for building CFGs reduces the number of nodes and edges by more than 30%.

5.1 Precision

To ensure that our analyses yield useful results, we compared them against the results that JJI and SQ report.

Dead Assignment Analysis JJI and SQ provide subtly different DAA variants. JJI's DAA corresponds largely to our LVA (Section 4.2) with minimal filtering, while SQ additionally applies the default value filtering heuristic from Section 4.3. We therefore ran two variants of our DAA, the JJI-style IntraJ-NH (*non-heuristic*), and the SQ-style IntraJ-H (*heuristic*). For SQ's reports, we filtered reports that involved multiple methods (FOP: 24; JFC: 5; PMD: 8), since SQ can use interprocedural analysis within one file.

The Venn diagrams in the upper part of Figure 13 show the number of DAA reports for each project, categorised by their overlap among the different check-

[3]Based on ExtendJ commit `a56a2c2` and JastAdd commit `faf36d2`

[4]Using JastAdd2 release `2.1.4-36-g18008bb` and JastAddJ-intraflow commit `b0b7c00`, restored with the original authors' generous help

	LOC	Qty	IntraJ	JJI	%
ANTLR v. 2.7.2	33˙737	Roots	2˙667	2˙329	+14.5
		Nodes	76˙925	116˙523	-39.9
		Edges	85˙028	136˙528	-37.7
PMD v. 4.2	49˙610	Roots	6˙215	5˙960	+4.26
		Nodes	103˙739	182˙864	-43.2
		Edges	108˙639	202˙842	-46.4
JFC v 1.0.0	95˙664	Roots	9˙271	7˙889	+17.5
		Nodes	219˙419	331˙368	-33.7
		Edges	220˙256	363˙642	-39.4
FOP v 0.95	97˙288	Roots	11˙327	8˙921	+26.9
		Nodes	239˙096	347˙125	-31.1
		Edges	240˙068	379˙269	-36.6

Table 1: Benchmark size metrics, LOC from `cloc`. The rest are CFG sizes. Roots is the number of intraprocedural CFGs. For IntraJ, this includes static and instance initialisers.

ers. For each category with 20 or fewer reports, we manually inspected all reports. For other categories, we sampled and manually inspected at least 20 reports or 20% of the reports (whichever was higher).

The Venn diagrams are dominated by two bug report categories: reports from the intersection of IntraJ-NH and JJI, which are initialisations of variables with default values, and reports from the intersection of all tools. For these two categories, we found all inspected reports to be true positives, modulo the DAA heuristic (Section 4.3). The remaining cases are often false positives: SQ reports 8 and 44 false positives in PMD and FOP that seem to largely stem from imprecision in handling `try-catch` blocks. Meanwhile, JJI reports 9 false positives in PMD while handling `break` statements. IntraJ reports two false positives, due to missing two exceptional flow edges for unchecked exceptions (Section 3.3). These do not affect JJI (and possibly SQ), since JJI conservatively merges exceptional and regular control flow.

Null Pointer Analysis For NPA (lower part of Figure 13), IntraJ detects at least as many reports as SQ, except for PMD, where SQ is able to exploit path sensitivity to identify three additional true positives. Similarly, the false positives reported only by IntraJ are mostly due to the lack of path-sensitivity. Listing 2 shows a simplified example.

We found that most of the false positives in the intersection of IntraJ and SQ are due to the lack of interprocedural knowledge. Listing 3 gives a simplified example. The code here checks if `rs` is `null` and, if so, calls `panic()` to halt

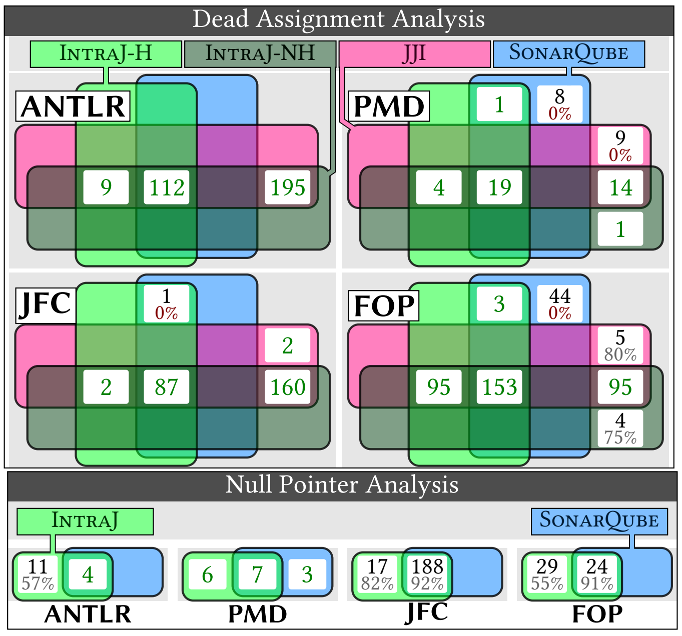


Figure 13: Venn diagram: number of reports shared across checkers, and percentage of true positives (unless 100%).

execution. IntraJ and SQ treat `panic()` as a regular method call and infer that `rs` may be `null` when dereferenced.

```
void bar(boolean flag){
  Object o = null;
  if (flag)
    o = new Object();
  if (flag)
    println(o.toString());
}
```

Listing 2: Simplified false positive reported by IntraJ

```
void foo(){
  Object rs = getRS();
  if(rs==null)
     // rs can be null
     panic(); //exit(1)
  println(rs.toString());
}
```

Listing 3: False positive due to intraprocedural limitations

5.2 Performance

We evaluated IntraJ's runtime performance with the above benchmarks on an octa-core Intel i7-11700K 3.6 GHz CPU with 128 GiB DDR4-3200 RAM, running Ubuntu 20.04.2 with Linux 5.8.0-55-generic and the OpenJDK Runtime Environment Zulu 7.44.0.11-CA-linux build 1.7.0_292-b07.

We separately measured both *start-up* performance on a cold JVM (restarting the JVM for each run) and *steady-state* performance (for a single measurement after 49 warmup runs). We measured each for 50 iterations (i.e., 2500 analysis runs for steady-state) and report median and 95% confidence intervals for IntraJ, JJI, and SQ, where applicable.

2 and Table 3 summarise our results. The Baseline column in Table 2, gives the times for each tool to load each benchmark, without data flow analyses. For SQ, we report the command line tool run time, with checkers disabled. For IntraJ and JJI, this time includes parsing, name, and type analysis. As JJI uses old versions of JastAdd and ExtendJ (formerly JastAddJ) from 2013, it reports different baselines. We speculate that the delta is due to bug fixes and other changes to JastAdd and ExtendJ.

We measured DAA and NPA, as well as CFG construction time, on separate runs (column An.sys). Table 3 has some missing values since JJI does not provide an implementation for NPA analysis, and since for SQ, we were unable to trigger the construction of the CFG only. Further, we could not measure steady state for SQ, since we ran it out of the box.

For start-up measurements, we then subtracted the baseline timings. DAA and NPA timings include on-demand CFG construction time. For the CFG measurements, we iterated over the entire AST and computed the ↑succ attribute.

The $\%_{\mathrm{JJI}}$ and $\%_{\mathrm{SQ}}$ columns summarise IntraJ's performance against JJI and SQ as slowdown (in percent), i.e. IntraJ was faster whenever we report less than 100.

We see that IntraJ is often slower than JJI for small benchmarks, but outperforms it as the benchmarks grow in size, especially in steady-state. This trend mirrors the additional overhead that IntraJ expends on computing smaller, more accurate CFGs: the difference between the CFG and DAA timings is consistently smaller for IntraJ than it is for JJI, and becomes more significant for larger benchmarks.

For the industrial-strength SQ, we observe that its baseline is longer than IntraJ's, and an explanation might be that it includes computations that for IntraJ would be attributed to the analyses. A strict comparison to SQ is therefore difficult, but we observe that IntraJ is considerably faster including the baseline, at most 3.12 times slower for DAA only, and considerably faster for NPA only, though the latter is likely due to SQ's more expensive interprocedural analysis.

Overall, our results support that IntraJ enables practical data flow analyses, with run-times and precision similar to state-of-the-art tools. Moreover, the results support that the overhead that IntraJ invests in refining CFG construction over jastaddj-intraflow pays off: client analyses can amortise this cost, and we expect this benefit to grow for analyses on taller lattices (e.g., interval or typestate analyses).

Table 2: Measure the baseline execution time and 95% confidence intervals using 50 data points per reported number.

Benchmark	Baseline(s)		
ANTLR	IntraJ	JJI	SQ
	2.14±0.01	1.34±0.01	4.91±0.05
PMD	IntraJ	JJI	SQ
	3.56±0.01	2.34±0.02	10.76±0.09
JFC	IntraJ	JJI	SQ
	4.29±0.01	3.14±0.02	10.81±0.11
FOP	IntraJ	JJI	SQ
	4.42±0.00	3.32±0.00	17.20±0.12

Table 3: Benchmark mean execution time (seconds) and 95% confidence intervals over 50 data points per reported number. We are reporting only confidence intervals greater than 0.02.

Benchmark	Start-up						Steady state		
	An.sys	IntraJ	JJI	SQ	$\%_{JJI}$	$\%_{SQ}$	IntraJ	JJI	$\%_{JJI}$
ANTLR	CFG	0.29	0.16	-	181	-	0.05	0.04	125
	DAA	0.53	0.43	0.24±0.05	123	220	0.12	0.13	92
	NPA	0.90	-	12.35±0.10	-	7	0.27	-	-
PMD	CFG	0.28	0.11	-	120	-	0.07	0.06	116
	DAA	0.47	0.39	0.18±0.08	120	261	0.12	0.16	75
	NPA	0.80	-	12.40±0.13	-	6	0.26	-	-
JFC	CFG	0.45	0.45±0.04	-	100	-	0.12	0.12	100
	DAA	0.75	1.07±0.03	0.24±0.11	70	312	0.25	0.34	73
	NPA	1.62	-	10.71±0.12	-	13	0.60	-	-
FOP	CFG	0.36	0.33	-	109	-	0.14	0.17	82
	DAA	0.67	0.74	0.34±0.12	90	197	0.26	0.39	66
	NPA	1.42	-	19.25±0.14	-	7	0.67	-	-

6 Related Work

Our work is most similar to JASTADDJ-INTRAFLOW [Söd+13a], the earlier RAG-based control- and data flow framework. As demonstrated, our CFG framework is more general, leading to more concise CFGs, avoiding misplaced nodes, and handling control flow that does not follow the AST structure, like initialisation code. Furthermore, our framework is formulated as a complete language-independent framework (Fig 2) with interfaces and default equations for all nodes involved in the CFG computation, and it has a more precise predecessor relation, excluding unreachable nodes. Our application of the framework to Java is more precise than the earlier work, making use of HOAs for reifying implicit structure, e.g., in connection to `finally` blocks. Additionally, we implemented the analyses for Java 7, including complex flow for `try-with-resources`, whereas [Söd+13a] only supported Java 5.

Earlier work on adding control flow to attribute grammars includes a language extension to the Silver attribute grammar system [VWK07; Van+10b] which supports that AST nodes are marked as CFG nodes, and successors are defined using an inherited attribute. Data flow is implemented by exporting data flow properties as temporal logic formulas, and using model checking to implement the analysis. The approach is demonstrated on a small subset of C. No performance results are reported, and scalability issues are left for future work.

Other declarative frameworks for program analysis have also demonstrated flow-sensitive analysis support. SOUL [De 11] exposes data flow information for Java 1.5 from Eclipse through a SmallTalk dialect combined with Prolog, though we were unable to obtain performance numbers for bug checkers or related analyses based on SOUL. Like our system, SOUL uses on-demand evaluation. DeepWeaver [Fal+07] supports data flow analysis and program transformation on byte code. Meanwhile, Flix [ML20] combines Datalog-style fixpoint computations and functional programming for declarative data flow analysis, and can scale IFDS/IDE-style interprocedural data flow analysis to nontrivial software [MYL16b]. To the best of our understanding, Flix does not connect to any compiler frontend, and we assume that Flix users rely on Datalog-style fact extractors to bridge this gap. MetaDL [DBR19] illustrates how to synthesise fact extractors from a JastAdd-based language, and we expect that it can directly expose INTRAJ edges.

FlowSpec [SWV20] is a DSL for data flow analysis based on term rewriting. To the best of our knowledge, FlowSpec has only been demonstrated on educational and domain-specific languages. Rhodium [Ler+05] uses logical declarative specifications for data flow analysis and transformation, to optimise C code and to prove the correctness of the transformations.

Other declarative systems that do not handle data flow include logic programming based techniques [BS09], term rewriting systems [Vis04], and XPath processors [Cop05].

Our work has focused on intraprocedural data flow analyses [Kil73; KU77; CC77]. However, existing (IR-based) program analysis tools like Soot [VR+10], Wala [FD12], or Opal [Hel+20] include provisions for interprocedural analysis, too. We currently see no fundamental challenge towards scaling our techniques to interprocedural analysis and expect only minor changes to the IntraCFG interfaces, for context-sensitivity. Such an effort would require additional analyses (call graph, points-to). We hypothesise that our implicit handling of recursive dependencies can eliminate the need for pre-analyses or complex worklist schemes [LH03], analogously to Datalog-based analyses [SB10]. While we expect that it is possible to integrate highly scalable data flow algorithms like IFDS, IDE [RHS95; SRH96], or SPPD [SAB19] into RAG interfaces, such interfaces may require a different design than IntraCFG and IntraJ to e.g. accommodate procedure summaries and to better enforce and exploit the invariants of these more specialised algorithms.

7 Conclusions

We presented IntraCFG, a RAG-based declarative language-independent framework for constructing intraprocedural CFGs. IntraCFG superimposes CFGs on the AST, allowing client analyses to take advantage of other AST attributes, such as type information and precise source information. We validated our approach by implementing IntraJ, an application of IntraCFG to Java 7, and demonstrated how IntraCFG overcomes the limitations of an earlier RAG-based framework, jastaddj-intraflow (JJI), by allowing the CFG to not be constrained by the AST structure. Compared to JJI, IntraJ can faithfully capture execution order and improve CFG conciseness and precision, removing more than 30% of the CFG edges in our benchmarks. We evaluated IntraJ by implementing two data flow analyses: Null Pointer Exception Analysis (NPA) and Dead Assignment Analysis (DAA), comparing both to JJI (for DAA), and to the highly tuned commercial tool SonarQube (SQ) (for DAA and NPA). Our results show that the IntraJ-based analyses offer precision that is comparable to that of JJI and SQ. Compared to JJI, IntraJ pays some overhead for computing more precise CFG but can amortise this effort for larger programs by speeding up client analyses, outperforming JJI. Compared to SQ, IntraJ's NPA analysis is substantially faster, although this is likely due to SQ's more advanced interprocedural analysis. IntraJ's DAA analysis seems slower than SQ's, but SQ has a much larger baseline, which might include computations that we would attribute to the analysis for IntraJ. Overall, we find that our results demonstrate that IntraJ-based data flow analyses are practical, that IntraJ enables precise data flow analyses on Java source code, and that IntraCFG is effective for constructing CFGs for Java-like languages. Moreover, we demonstrate for the first time how RAGs can build and exploit graph structures over an AST without being restricted by the AST's structure.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[Ami+16] Afshin Amighi et al. "Provably correct control flow graphs from Java bytecode programs with exceptions". In: *International journal on software tools for technology transfer* 18.6 (2016), pp. 653–684.

[Bla+06] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190.

[BS09] Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses". In: *Proceedings of OOPSLA '09*. Orlando, Florida, USA: ACM, 2009, pp. 243–262.

[Cho+99] Jong-Deok Choi et al. "Efficient and precise modeling of exceptions for the analysis of Java programs". In: *ACM SIGSOFT Software Engineering Notes* 24.5 (1999), pp. 21–31.

[Cop05] Tom Copeland. *PMD applied*. Vol. 10. Centennial Books Arexandria, Va, USA, 2005.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252.

[De 11] Coen De Roover. "A Logic Meta-Programming Foundation for Example-Driven Pattern Detection in Object-Oriented Programs". English. In: *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011). 2011.

[DBR19] Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. "MetaDL: Analysing Datalog in Datalog". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2019, pp. 38–43.

[EH07a] Torbjörn Ekman and Görel Hedin. "The jastadd extensible Java compiler". In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. 2007, pp. 1–18.

[Fal+07] Henry Falconer et al. "A Declarative Framework for Analysis and Optimization". In: *Compiler Construction*. Ed. by Shriram Krishnamurthi and Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 218–232.

[FD12] Stephen Fink and Julian Dolby. *WALA–The TJ Watson Libraries for Analysis*. 2012.

[FSH20] Niklas Fors, Emma Söderberg, and Görel Hedin. "Principles and patterns of JastAdd-style reference attribute grammars". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*. Ed. by Ralf Lämmel, Laurence Tratt, and Juan de Lara. ACM, 2020, pp. 86–100.

[Hed00] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000).

[HM03] Görel Hedin and Eva Magnusson. "JastAdd—an aspect-oriented compiler construction system". In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[Hel+20] Dominik Helm et al. "Modular collaborative program analysis in OPAL". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 184–196.

[HSP05] David Hovemeyer, Jaime Spacco, and William Pugh. "Evaluating and tuning a static analysis to find null pointer bugs". In: *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2005, pp. 13–19.

[JC04] Jang-Wu Jo and Byeong-Mo Chang. "Constructing control flow graph for java by decoupling exception flow from normal flow". In: *International Conference on Computational Science and Its Applications*. Springer. 2004, pp. 106–113.

[Jou84] Martin Jourdan. “An Optimal-time Recursive Evaluator for Attribute Grammars”. In: *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings.* Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 167–178.

[KU77] John B Kam and Jeffrey D Ullman. “Monotone data flow analysis frameworks”. In: *Acta informatica* 7.3 (1977), pp. 305–317.

[Kil73] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL ’73. Boston, Massachusetts: Association for Computing Machinery, 1973, 194–206.

[Knu68] Donald E Knuth. “Semantics of context-free languages”. In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[Ler+05] Sorin Lerner et al. “Automated soundness proofs for dataflow analyses and transformations via local rules”. In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 364–377.

[LH03] Ondřej Lhoták and Laurie Hendren. “Scaling Java points-to analysis using Spark”. In: *International Conference on Compiler Construction.* Springer. 2003, pp. 153–169.

[ML20] Magnus Madsen and Ondřej Lhoták. “Fixpoints for the masses: programming with first-class Datalog constraints”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–28.

[MYL16b] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. “From Datalog to flix: a declarative language for fixed points on lattices”. In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 194–208.

[MEH07a] Eva Magnusson, Torbjorn Ekman, and Gorel Hedin. “Extending Attribute Grammars with Collection Attributes–Evaluation and Applications”. In: *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007).* IEEE. 2007, pp. 69–80.

[MH07b] Eva Magnusson and Görel Hedin. “Circular reference attributed grammars—their evaluation and applications”. In: *Science of Computer Programming* 68.1 (2007), pp. 21–37.

[Öqv18] Jesper Öqvist. “Contributions to Declarative Implementation of Static Program Analysis”. PhD thesis. Lund University, 2018.

[Rei21] Christoph Reichenbach. “Software Ticks Need No Specifications”. In: *Proceedings of the 43rd International Conference on Software Engineering: New Ideas and Emerging Results Track.* Virtual, 2021.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 1995, pp. 49–61.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. “Precise interprocedural dataflow analysis with applications to constant propagation”. In: *Theoretical Computer Science* 167.1-2 (1996), pp. 131–170.

[SB10] Yannis Smaragdakis and Martin Bravenboer. “Using Datalog for fast and easy program analysis”. In: *International Datalog 2.0 Workshop.* Springer. 2010, pp. 245–251.

[Smi+15] Justin Smith et al. “Questions developers ask while diagnosing potential security vulnerabilities with static analysis”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 2015, pp. 248–259.

[SWV20] Jeff Smits, Guido Wachsmuth, and Eelco Visser. “FlowSpec: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis”. In: *Journal of Computer Languages* 57 (2020), p. 100924.

[Söd+13a] Emma Söderberg et al. “Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level”. In: *Sci. Comput. Program.* 78.10 (Oct. 2013), 1809–1827.

[SAB19] Johannes Späth, Karim Ali, and Eric Bodden. “Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019).

[Sza] Tamás Szabó. “Incrementalizing Static Analyses in Datalog”. PhD thesis. Johannes Gutenberg-Universität Mainz.

[VR+10] Raja Vallée-Rai et al. “Soot: A Java Bytecode Optimization Framework”. In: *CASCON First Decade High Impact Papers.* CASCON ’10. Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224.

[VWK07] Eric Van Wyk and Lijesh Krishnan. “Using verified data-flow analysis-based optimizations in attribute grammars”. In: *Electronic Notes in Theoretical Computer Science* 176.3 (2007), pp. 109–122.

[Van+10b] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1 (2010). Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA ’06 and ’07), pp. 39–54.

[Vis04] E. Visser. “Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9”. In: *Lecture Notes in Computer Science* 3016 (2004). Ed. by C. Lengauer et al., pp. 216–238.

[VSK89b] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. “Higher Order Attribute Grammars”. In: *SIGPLAN Not.* 24.7 (1989), 131–145.

IntraJ: An On-Demand Framework for Intraprocedural Java Code Analysis

Abstract

Static analysis tools play a crucial role in software development by detecting bugs and vulnerabilities. However, running these tools separately from the code editing process often leads to context switching for developers, resulting in decreased productivity. To address this issue, we present IntraJ, a responsive and extensible framework for intraprocedural control-flow and dataflow analysis for Java source code. By using on-demand evaluation and Reference Attribute Grammars (RAGs), IntraJ can provide real-time analysis results directly in the editor, similar to compile-time error detection. The framework is implemented on top of the ExtendJ extensible Java compiler, which provides a complete Java compiler implementation. We demonstrate the use of IntraJ in various development environments, including the command line, an editor integration based on the Language Server Protocol, and an integration into the debugging tool CodeProber. Finally, we showcase the extensibility of IntraJ by illustrating how new client analyses and language constructs can be added to the framework through RAG specifications. We evaluate the performance of IntraJ on a set of real-world

Idriss Riouak, Niklas Fors, Görel Hedin, and Christoph Reichenbach.
"IntraJ: An On-Demand Framework for Intraprocedural Java Code Analysis".
Submitted for publication.

Java benchmarks, demonstrating that IntraJ can provide real-time feedback to developers by analyzing most compilation units in under 0.1 seconds.

1 Introduction

Detecting software bugs early in the development process is crucial. Early detection not only significantly reduces costs but can also prevent severe consequences, such as system failures, security breaches, and financial losses [Saw99].

As a result, software development is increasingly relying on static analysis tools to identify defects and vulnerabilities. Tools like *Infer* [Dis+19], *PMD* [Cop05], and *SpotBugs* (successor of FindBugs [Aye+08b]), are becoming integral parts of the development process, providing automated analysis of code quality and security issues. These tools are typically run as part of an automated pipeline enabling the detection of post-editing issues [Sad+15]. While this approach is effective, it is prone to errors as developers often switch their focus to other tasks while waiting for the analysis results [Nad22]. Such context switching can introduce new errors and issues into the codebase.

Detecting and displaying static analysis problems directly in the editor provides real-time feedback analogous to compile-time error detection. This integration reduces context switching and helps maintain the developer's continuous focus and productivity. Achieving this requires low response times; ideally, analysis results should be available within 0.1 seconds to ensure the system feels instantaneous to the user [Nie94]. Despite the evident benefits, the challenge of delivering such responsive static analysis remains largely unaddressed.

In this paper we revisit our previous work on IntraJ [Rio+21], a framework designed for responsive and extensible intraprocedural control-flow and dataflow analysis of Java source code. Here, our emphasis is on describing the architecture and demonstrating the extensibility and efficiency of IntraJ for interactive analyses, highlighting its modularity and responsiveness. IntraJ achieves low latency through two key approaches: *on-demand evaluation*, computing only the necessary parts of the analysis relevant to the current editing context, and *source-level analysis*, performing analysis directly on the abstract syntax tree rather than bytecode, avoiding the overhead of bytecode generation. This approach minimizes computational resources and ensures efficient and responsive analysis.

IntraJ is built using Reference Attribute Grammars (RAGs) [Hed00], a high-level declarative formalism that inherently supports on-demand evaluation. Our framework is implemented as an extension of the ExtendJ [EH07b] Java compiler, which already uses RAGs for name binding and type analysis. The IntraJ API includes methods for traversing the control-flow graph and computing dataflow analyses, enabling the development of custom analyses using RAGs.

IntraJ offers an extensible API, allowing developers to integrate additional analyses as RAG specifications. These extensions can leverage both IntraJ and

ExtendJ APIs, benefiting from automatic on-demand evaluation.

We showcase IntraJ across multiple development environments, including a command line tool, an editor integrated with the Language Server Protocol, and the debugging tool CodeProber [Ala+24b]. Our previous work [Rio+21] showed that IntraJ achieves precision comparable to industrial-strength tools like *SonarQube* and has efficient performance on whole program analyses.

In this paper, we focus on the performance of IntraJ in interactive environments. We present results that highlight its ability to deliver real-time analysis with no noticeable latency, even in large codebases. Our results show that for most projects, IntraJ can analyze 99% of the compilation units (one at a time) in less than 0.1 seconds. This demonstrates IntraJ's suitability for integration into modern development workflows where immediate feedback is crucial.

The paper is structured as follows: we begin with a brief overview of RAGs and Monotone Frameworks (Section 2), followed by a presentation of our contributions:

- We present the architecture and APIs of IntraJ (Section 3).
- We demonstrate the integration and use of IntraJ into three different development tools, highlighting its flexibility and responsiveness (Section 4).
- We illustrate the framework's extensibility through the addition of new client analyses written as RAG specifications (Section 5).
- We evaluate the performance of IntraJ in interactive environments (Section 6).

The paper concludes by discussing related work (Section 7). Finally, we present our conclusions and outline future work (Section 8).

2 Background

This section introduces Reference Attribute Grammars for defining language semantics and monotone frameworks for reasoning about program dataflow properties.

2.1 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) [Hed00] are a formalism for specifying how programming languages should be evaluated and analyzed. RAGs represent programs as abstract syntax trees (ASTs) decorated with computed properties called *attributes*. RAGs extend Knuth's Attribute Grammars [Knu68] by allowing attributes to reference other AST nodes. This extension allows RAGs to define relations between nodes in the abstract syntax tree and to superimpose graphs

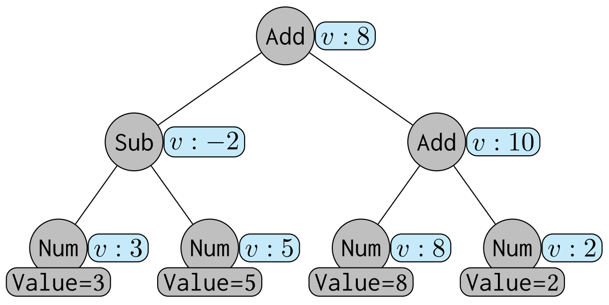


Figure 1: Decorated AST representing the expression `(3-5) + (8+2)`. Attribute `v` is the value of the expression.

Listing 1: Abstract grammar for the expression language.

```
abstract Expr;
Add : Expr ::= Left:Expr Right:Expr;
Sub : Expr ::= Left:Expr Right:Expr;
Num : Expr ::= <Value:int>;
```

over the AST, including name bindings, type hierarchies, and control-flow graphs (Section 2.2).

To illustrate this concept, we will consider a very simple arithmetic expression language with additions, subtractions, and numeric literals. The input text is parsed into an AST that is decorated with attributes. For example, the decorated AST of the expression $(3 - 5) + (8 + 2)$ is shown in Figure 1. Each node has an attribute `v`, which is the value of the node and its subtree. The value of the whole expression is then `v` of the root node.

Listing 1 shows the abstract grammar for this language. To define the abstract grammar and attributes, we use the meta-compilation system JastAdd [HM03]. Given a RAG specification, JastAdd generates Java code for the AST classes and the attribute evaluation methods. Thus, the abstract grammar in the example above defines four Java classes to represent the different program elements.

Listing 2: Specification of the attribute v.

```
syn int Expr.v();
eq Add.v() = getLeft().v() + getRight().v();
eq Sub.v() = getLeft().v() - getRight().v();
eq Num.v() = getValue();
```

The classes `Add`, `Sub`, and `Num` are subclasses of the abstract class `Expr`. `Add` and `Sub` have two children, `Left` and `Right`, of type `Expr`. The `Num` class has a token `Value` representing the numerical value.

Listing 2 shows the definition of the attribute `v`. The attribute is declared on the class `Expr`, and each subclass defines an equation for it. Each equation names the attribute on its left-hand side and gives a Java method body without observable side effects on the right-hand side. For each attribute, JastAdd generates a namesake method, which here allows `v` to access the values directly. The equation for a `Num` node simply returns its value. Children and tokens are accessed by methods prefixed with `get`. The equation for an `Add` node accesses `v` on its children and adds them together.

When attribute equations reference other attributes, their evaluation may recurse: for example, if we evaluate the `v` attribute of the `Sub` node in Figure 1, the right-hand side of the equation for `Sub.v` will recurse into the children, both of which will use the equation for `Num.v` and return their literal values (3 and 5, respectively).

This example uses only *synthesized* attributes (indicated by the keyword `syn`), meaning that their defining equations are evaluated in the context of the node to which the attribute belongs, analogously to Java methods. JastAdd supports other kinds of attributes beyond synthesized ones, such as *inherited attributes*, which are evaluated in the context of the parent node and are used for providing nodes with contextual information. This allows passing information downwards in the AST, which IntraJ uses heavily when constructing the control-flow graph. Attributes may transitively depend on their own values, as long as they are explicitly declared to be *circular attributes* [MH07b; Far86; JS86]. *Circular attributes* allow computing fixed points, which is an essential part of dataflow analysis.

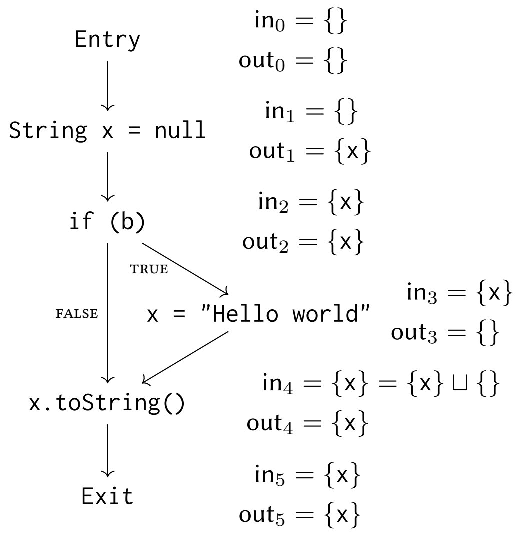
When an attribute is accessed from Java, JastAdd computes it on demand and memoizes the result. Memoization ensures that once an attribute's value is computed, it is stored for future use, preventing redundant calculations and improving efficiency.

2.2 Monotone Frameworks

Many interesting program properties depend on the order in which different parts of the program execute, and on how the contents of the program's variables change over time. To answer questions about about e.g. redundant computations or the variables' contents and liveness, modern production compilers and many software tools rely on *monotone frameworks* [Kil73; KU77], a unifying theoretical approach that enables analyses such as *live variables*, *available expressions*, or *reaching definitions*.

These analyses propagate information along the control-flow graph (CFG), a graph that overapproximates all possible sequences of steps in which a program may execute, with each step represented as a CFG node.

```
void foo(boolean b) {
  String x = null;
  if (b) {
    x = "Hello␣world";
  }
  x.toString();
}
```



Entry
String x = null
if (b)
TRUE
FALSE
x = "Hello world"
x.toString()
Exit
$\mathsf{in}_0 = \{\}$
$\mathsf{out}_0 = \{\}$
$\mathsf{in}_1 = \{\}$
$\mathsf{out}_1 = \{\mathsf{x}\}$
$\mathsf{in}_2 = \{\mathsf{x}\}$
$\mathsf{out}_2 = \{\mathsf{x}\}$
$\mathsf{in}_3 = \{\mathsf{x}\}$
$\mathsf{out}_3 = \{\}$
$\mathsf{in}_4 = \{\mathsf{x}\} = \{\mathsf{x}\} \sqcup \{\}$
$\mathsf{out}_4 = \{\mathsf{x}\}$
$\mathsf{in}_5 = \{\mathsf{x}\}$
$\mathsf{out}_5 = \{\mathsf{x}\}$

Figure 2: Example program (left) and its control-flow graph (right). The CFG is annotated with the `in` and `out` sets containing all the variables that might be `null`.

To express an analysis as a monotone framework, we combine the CFG with two additional components:

- a datatype that defines the information that we want to collect, along with an operation that reconciles possibly conflicting information from different branches (a *semilattice*, formally speaking), and
- a family of *transfer functions* that explain how passing through a CFG node updates this information.

Figure 2 demonstrates this idea with a conservative *null pointer analysis* of a Java program. The program on the left-hand side will throw a `NullPointerException` when the method `toString` is invoked and if the parameter b has the value `false`.

The right-hand side of the same figure shows the program's control flow graph, with control flow edges connecting individual statements in the order of execution. For each of the six control flow nodes, the set in_i contains all variables that might be null before entering the node, and the set out_i contains all variables that might be null afterwards. For the `Entry` node, we assume that no variables may be `null`, so we set $\mathsf{in}_0 = \emptyset$, If f_i is the transfer function for CFG node i, then $\mathsf{out}_i = f_i(\mathsf{in}_i)$. For example,

$$f_1(s) = s \cup \{\mathsf{x}\}$$

since this CFG node assigns `null` to variable `x`. If there is a CFG edge from node i to node k, propagate the out_i to be in_k, as long as i is the only predecessor of k. If node k has multiple predecessors, we must reconcile (or *join*) the incoming information from all predecessors, as for in_4 in our example. Following convention, we write the reconciliation or join operator as $\sqcup$. Since we want to conservatively analyse which variables *might* be `null` on *any* control path, we use the set union, i.e., we set $\sqcup = \cup$. Therefore, we have $\mathsf{in}_4 = \mathsf{out}_2 \sqcup \mathsf{out}_3 = \mathsf{out}_2 \cup \mathsf{out}_3 = \{\mathsf{x}\}$.

The general rule for computing in_i and out_i is thus:

$$\begin{aligned} \mathsf{in}_i &= \bigsqcup_{p \in \mathrm{pred}(i)} \mathsf{out}_p \\ \mathsf{out}_i &= f_i(\mathsf{in}_i) \end{aligned}$$

This kind of analysis is called a *forward analysis* because it propagates information from the `Entry` node to the `Exit` node, but monotone frameworks also support *backward analyses* that traverse the CFG in the opposite direction (e.g., for liveness analysis).

If the source program contains a loop, the CFGs may be circular, which means that the results of out_n for some n may flow back into in_n for the same n, directly or indirectly. In these cases we must compute a fixed point, iterating our computation until none of the in_i or out_i change. Monotone frameworks define sufficient conditions over the transfer functions and the join operator to ensure that such a fixed point always exists, essentially by requiring that we never discard information and that the information for each CFG node reaches a saturation point after a finite number of updates [NNH10].

To express monotone frameworks in RAGs, we can describe transfer functions and the join operator directly as attribute equations over `in` and `out` sets. Since these attributes will generally recursively depend on themselves, we express them as circular attributes, which automatically support efficient on-demand fixed point computation [MH07b; Öqv18]. Section 5 gives an example of what a dataflow analysis expressed as a monotone framework might look like in a RAG.

3 IntraJ Architecture

IntraJ is a framework for the construction of CFGs for Java 8. It is entirely built using RAGs and offers an exception-sensitive control-flow analysis, considering both checked and unchecked exceptions. On top of IntraJ, we have implemented a number of dataflow analyses, such as *Live Variable analysis*, *Reaching Definition analysis*, and a *Null Pointer Dereference analysis.*

IntraJ is an instance of the IntraCFG framework for control-flow analysis, which is a language-independent framework for defining control-flow graphs using RAGs. IntraJ is built on top of the ExtendJ Java compiler, which is also implemented using RAGs.

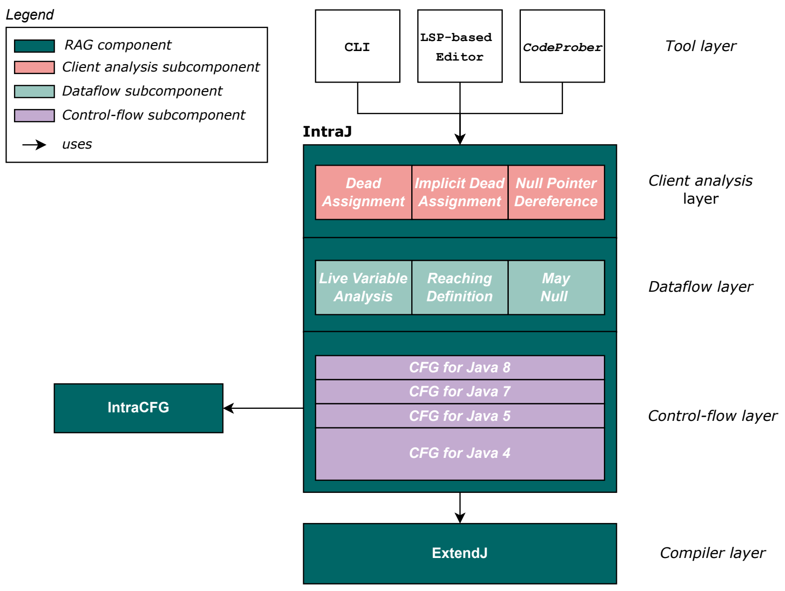


Figure 3: Layered architecture of IntraJ. Each IntraJ layer builds upon the attributes of the underlying layers.

Figure 3 presents the IntraJ architecture as layers of RAG components and subcomponents. Each component and subcomponent defines an attribute API, and uses the APIs of its own and lower layers. We will now discuss each component and subcomponent in more detail.

3.1 The ExtendJ Compiler

ExtendJ is an open-source extensible Java compiler implemented using RAGs. It currently supports Java 4-11, including parsing, compile-time checking and bytecode generation. ExtendJ defines the node types of a Java AST, such as different kinds of declarations, statements, and expressions. It also defines a number of RAG attributes for these node types, like name bindings, types, compile-time errors, and bytecode. ExtendJ offers thousands of predefined attributes available as an API for the programmer to use when building analyses or transformations. As for any RAG-based system, the attributes are computed on demand, and the computation is driven by the attribute API[1].

Once the AST has been constructed, the programmer can call any attribute

[1]The complete ExtendJ API is available at `https://extendj.org/javadoc/`.

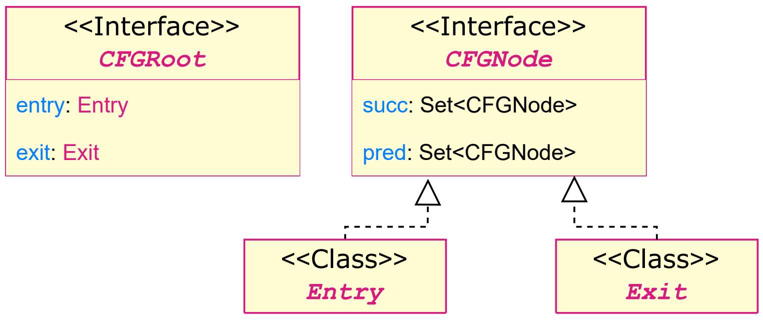


Figure 4: Main API of IntraCFG.

method to get its value. No explicit compilation passes like name binding, type-checking or code generation are needed. To compile a Java program, ExtendJ simply parses the source text into an AST, calls an error attribute to see if there are compile-time errors, and calls bytecode attributes to print the resulting bytecode to suitable class files.

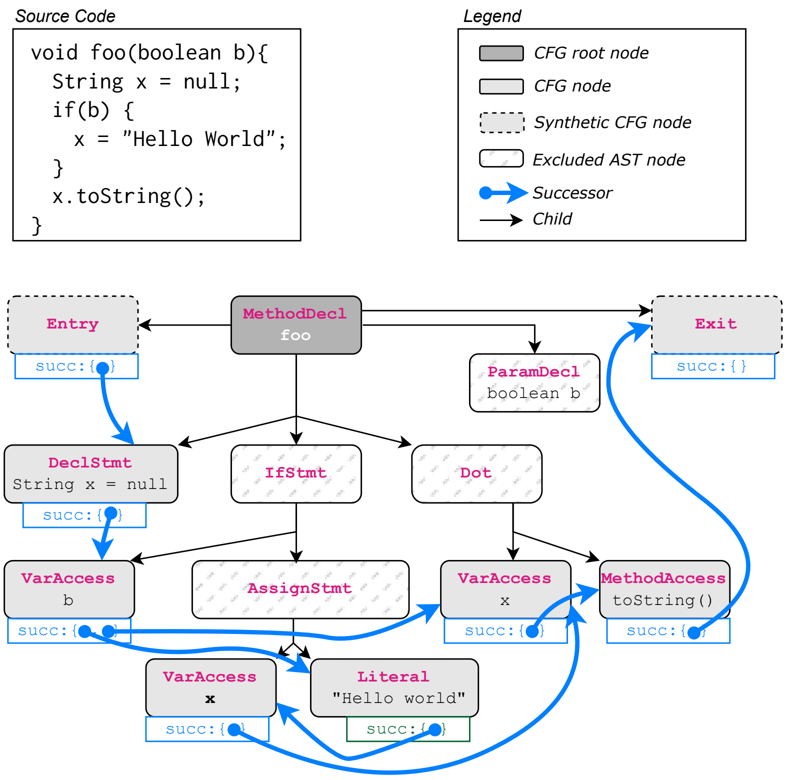
3.2 The IntraCFG component

ExtendJ does not itself provide a CFG for Java. Instead, IntraJ builds on IntraCFG, a language-agnostic RAG component for CFG construction. IntraCFG defines a set of *node type interfaces*, which are analogous to Java interfaces but include overridable default attribute equations.

Figure 4 shows the main API of IntraCFG in the form of interfaces and classes. The `CFGRoot` interface is intended for AST node types that represent subroutines, e.g., method and constructor declarations in Java. Each AST node type that implements the `CFGRoot` interface is automatically extended with two synthetic AST nodes, `Entry` and `Exit`, for the unique entry and exit points of that subroutine's CFG.

The `CFGNode` interface is intended for AST nodes that participate in the CFG as CFG nodes. Each AST node type that implements this interface obtains the attributes `succ` and `pred` through default equations. `succ` returns the control flow successors, and `pred` returns the predecessors of that node, respectively.

Language implementers can define CFGs for their language by adding these interfaces to AST node types of interest and overriding default equations as needed [Rio+21], though there are often multiple possible CFG designs for the same language, balancing precision against complexity.


Source Code

```
void foo(boolean b){
  String x = null;
  if(b) {
    x = "Hello World";
  }
  x.toString();
}
```

Figure 5: AST with superimposed CFG for the `foo` Java method.

3.3 IntraJ Control-Flow Analysis

To define control-flow graphs for Java, IntraJ adds the IntraCFG interfaces to selected ExtendJ node types, and adds and overrides attributes to define the detailed control flow. For IntraJ, we have chosen to implement precision at the expression level, as this approach captures a more detailed control flow, resulting in more precise analyses.

The `CFGRoot` interface is added to the ExtendJ types `MethodDecl` and `ConstructorDecl` so that each method and constructor gets a local CFG. Such additions are done by a simple specification statement in the RAG specification language, e.g.: `MethodDecl implements CFGRoot`.

The CFG is then constructed by adding the `CFGNode` interface to a number of AST node types, to reflect the control flow of statements and expressions. Attribute rules are added to define the detailed control flow so that it complies with the semantics of Java. This way, the CFG is constructed by superimposing it on a subset of the AST nodes. Not all AST nodes need to be included in the CFG, and synthetic AST nodes can be created to capture flow that is implicit, allowing a concise and precise graph to be constructed.

Figure 5 shows a simple example for a Java method `foo`. Here, the `MethodDecl` is a CFG root, and therefore automatically gets synthetic `Entry` and `Exit` nodes. The AST nodes of the types `DeclStmt`, `VarAccess`, `Literal`, and `MethodAccess` nodes are all CFG nodes. We can note that some nodes are excluded from the CFG. For example, the `IfStmt` is not part of the CFG since its control flow is captured by nodes in its subtree. The successor edges are captured by the `succ` attribute that contains references to the successor nodes in the graph. The `pred` attribute (not shown in the figure), represents the predecessor edges and is computed automatically by the IntraCFG component as the reverse of the successors.

The IntraJ CFG specification is structured into subcomponents, as was shown in Figure 3, one for each version of Java[2]. This matches a similar subcomponent structure inside ExtendJ, and allows Java compilers to be built for different language versions. A more detailed description on the construction of the IntraJ CFGs can be found in [Rio+21].

3.4 Dataflow and Client Analyses

IntraJ provides a number of example analyses in its client and dataflow layers. Figure 6 shows the APIs of these analyses, and how they use each other. They also use the APIs in the lower level CFG and compiler layers. The dataflow layer implements various analyses, including Live Variable Analysis, Reaching Definition Analysis, and May Null Analysis. These analyses compute dataflow facts for each node in the CFG. For instance, the `outLVA` attribute computes the set of variables that are live after a given CFG node. This means it determines which variables are used on any path from the current CFG node to the `Exit` node. The Reaching Definition Analysis, like Live Variable Analysis, tracks the definitions that reach each CFG node, considering transitive assignments. Similarly, the May Null Analysis determines the nullness status of reference variables at each node.

The analyses in the client layer make use of the general dataflow analyses to interpret the results in a way suitable for client tools. For example, in the Dead Assignment component, the `Assignment` interface is extended with a boolean attribute `isDeadAssign`. This attribute is true if the CFG node of the assigned variable represents a dead assignment (i.e., the assigned variable is not in the `outLVA` set) and if additional language-specific conditions are met, to capture heuristics.

[2]No subcomponent is needed for Java 6 since that version did not include any new language constructs or semantics that affect control-flow.

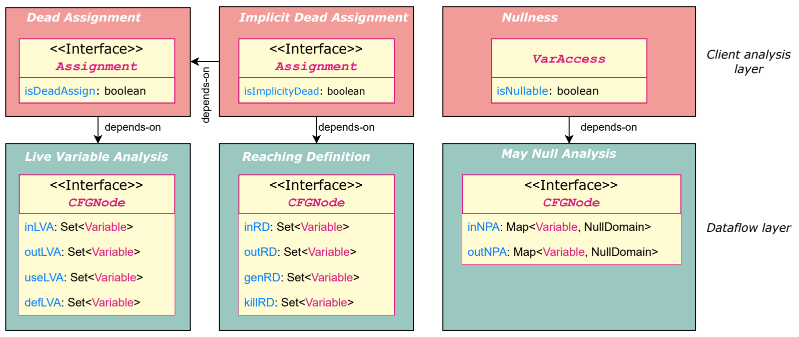


Figure 6: The APIs exposed by the IntraJ's dataflow and client analyses.

As an example of a heuristic condition, consider the declaration of a local variable with an initializing assignment. If the initializer is a usual default value in Java, like null or zero, the developer will typically consider this as good programming practice, even if the assignment is technically dead. Therefore, `isDeadAssign` is defined as false in this case. The client analysis uses the rich API of ExtendJ to easily specify such conditions.

The implicit dead assignment analysis is a client analysis that demands the computation of dead assignment analysis to find assignments that are not dead themselves, but are used in a dead assignment.

Figure 7 showcases examples of bugs detected by the analyses supported by IntraJ. In the first example, the variable `x` is identified as a dead assignment, meaning it is assigned a value which is never used thereafter. Similarly, the variable `y` is an indirect dead assignment because its value is assigned but only used as an operand in the assignment `int z = x + y`, which itself is identified as a dead assignment since the resulting value of `z` is never used.

The second example illustrates a potential null pointer exception. IntraJ detects that there is a potential null pointer exception in the code snippet since the variable `x` is assigned the value `null` and then dereferenced in the statement `x.toString()`.

1
```
void bar(boolean b){
    int x = 10;     // Dead Assignment
    int y = 20;     // Indirect Dead Assignment
    x = 30;         // Indirect Dead Assignment
    int z = x + y; // Dead Assignment
}
```

2
```
void foo(boolean b){
    String x = null;
    if(b){
        x = "Hello World"; }
    x.toString(); //NullPointerException
}
```

Figure 7: Examples of bugs detected by IntraJ. 1. Examples of dead assignments and implicitly dead assignments. 2. Example of null pointer exception.

3.5 Demand-Driven Analyses

As mentioned earlier, the attributes in both ExtendJ and IntraJ are computed on demand. Figure 8 illustrates how this works for the May Null analysis. The example is the same as in Figure 5, but showing only the CFG nodes, and not the full AST. In the example, the dereference of `x` in the statement `x.toString();` may generate a null pointer exception, namely if the boolean parameter `b` is false. The dereference is represented by a `VarAccess` node in the AST. To investigate if it can give a null pointer exception, a tool would query its `isNullable` attribute.

Querying `isNullable` will lead to the recursive evaluation of a number of additional attributes, as shown in Figure 8. In this example, evaluation of `isNullable` will use the `inNPA` attribute of the same node, which is defined using the `outNPA` attributes of the predecessors, which in turn are defined using `inNPA` attributes, and so on, recursively along the predecessors, all the way to the `Entry` node.

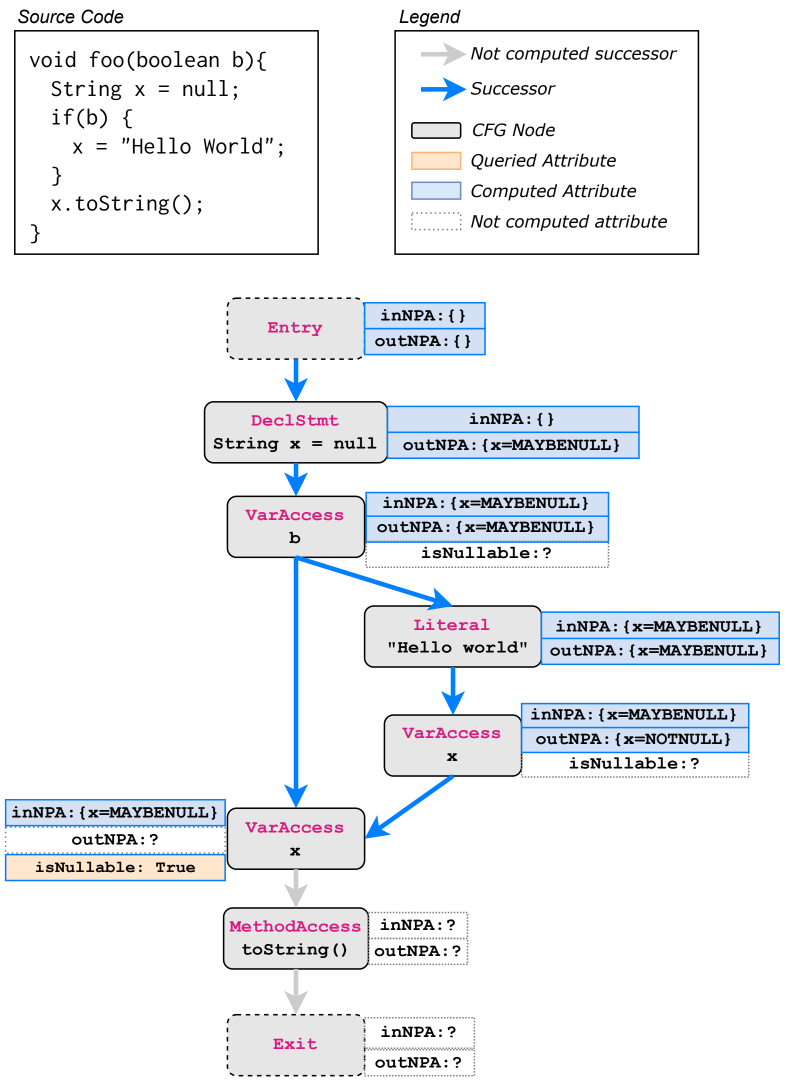
The `inNPA` and `outNPA` attributes contain the nullness status of variables right before and after the CFG node, respectively. For example, consider the the assignment `x = "Hello world"`, represented by the `VarAccess` for `x`. Here `inNPA` shows that `x` is `MAYBENULL` prior to the assignment, and `outNPA` shows that `x` is `NOTNULL` (i.e., it is definitely referring to an object).

To compute the `outNPA` of a node, the `inNPA` is combined with a transfer function that may use attributes defined by ExtendJ, such as the `decl` attribute linking a `VarAccess` to its declaration (not shown in the figure). These attributes are also evaluated on demand. Thus, when a tool queries a specific attribute, only a subset of all available attributes will be evaluated. Usually, this subset is very small, even for the first query when no attributes have yet been memoized.

4 Tool Integration

In this section, we discuss the integration of IntraJ with different development tools. We examine three major applications of IntraJ: command line integration, editor integration based on the Language Server Protocol, and integration into the debugging tool CodeProber. Each of these integrations highlights different ways that IntraJ can be used in tools.

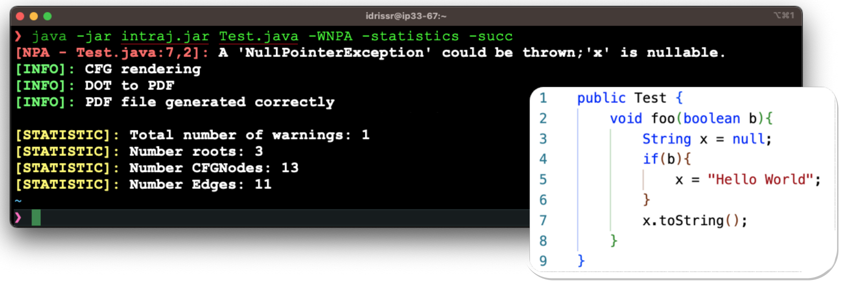
The command line integration allows developers to run the analysis on the entire codebase, providing a comprehensive overview of the issues in a project. The Language Server Protocol (LSP) is a standardized interface that enables editors and IDEs to communicate with language-specific analysis tools. Our LSP integration enables viewing issues in an edited file, running the analysis on save. This approach provides a shorter feedback loop, allowing developers to address issues as they are introduced. Finally, the CodeProber integration shows that IntraJ is fast enough to operate on every keystroke, delivering real-time analysis and feedback for immediate bug detection.



Source Code

```
void foo(boolean b){
  String x = null;
  if(b) {
    x = "Hello World";
  }
  x.toString();
}
```

Figure 8: On-demand evaluation example. Querying `isNullable` on a dereferenced `VarAccess` leads to the computation of only a subset of the NPA attributes. Note that the figure elides ExtendJ attributes, most of which the evaluation does not depend on (and therefore never computes).



```
idrissr@ip33-67:~
❯ java -jar intraj.jar Test.java -WNPA -statistics -succ
[NPA - Test.java:7,2]: A 'NullPointerException' could be thrown;'x' is nullable.
[INFO]: CFG rendering
[INFO]: DOT to PDF
[INFO]: PDF file generated correctly

[STATISTIC]: Total number of warnings: 1
[STATISTIC]: Number roots: 3
[STATISTIC]: Number CFGNodes: 13
[STATISTIC]: Number Edges: 11
~
❯
```

```
1   public Test {
2       void foo(boolean b){
3           String x = null;
4           if(b){
5               x = "Hello World";
6           }
7           x.toString();
8       }
9   }
```

Figure 9: Example of IntraJ running on the command line.

4.1 Command line Integration

Originally, IntraJ was developed as a command line tool, exhibiting competitive performance when compared to existing industrial tools [Rio+21]. The command line integration is suitable for continuous integration pipelines, as it can be easily integrated into existing workflows.

The command-line interface is similar to the javac and ExtendJ compilers, supporting the specification of, for example, classpath, sourcepath, libraries, and files to be analyzed/compiled. Specific IntraJ flags include what analyses to enable for warnings, such as `-WNPA`, `-WDAA` for null pointer analysis and dead assignment analysis, respectively.

Figure 9 illustrates an example of IntraJ running on the command line. The `-statistics` flag is used to summarise the analysis results, displaying the number of warnings and statistics related to the CFGs. When specified, the `-succ` flag generates a PDF file visualising the CFGs of the analysed methods.

4.2 Editor Integration

The demand-driven evaluation in IntraJ makes it very suitable for integration with interactive tooling: analyses can be performed efficiently on individual program elements or files, even if the analysis depends on information in a larger project. To investigate this kind of integration, we used the MagpieBridge framework [LDB19b], which facilitates the integration of static analysers with IDEs that support LSP. MagpieBridge provides an abstraction layer between the LSP protocol and the static analysis tool, allowing for the development of IDE plugins with a very low effort. It reruns the analyses after each save in the editor. The supported LSP functionalities include diagnostics, code actions (e.g., quick fixes), and code-lens.

Figure 10 illustrates the integration of IntraJ with different IDEs via MagpieBridge. ServerAnalysis is a component we developed to handle the communication between IntraJ and the MagpieBridge Server. It is responsible for

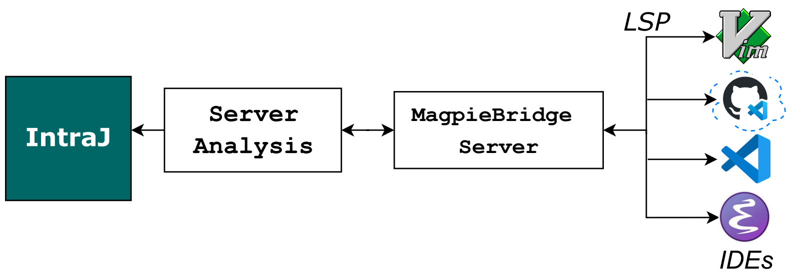


Figure 10: Integration of IntraJ with IDEs through the use of the MagpieBridge framework.

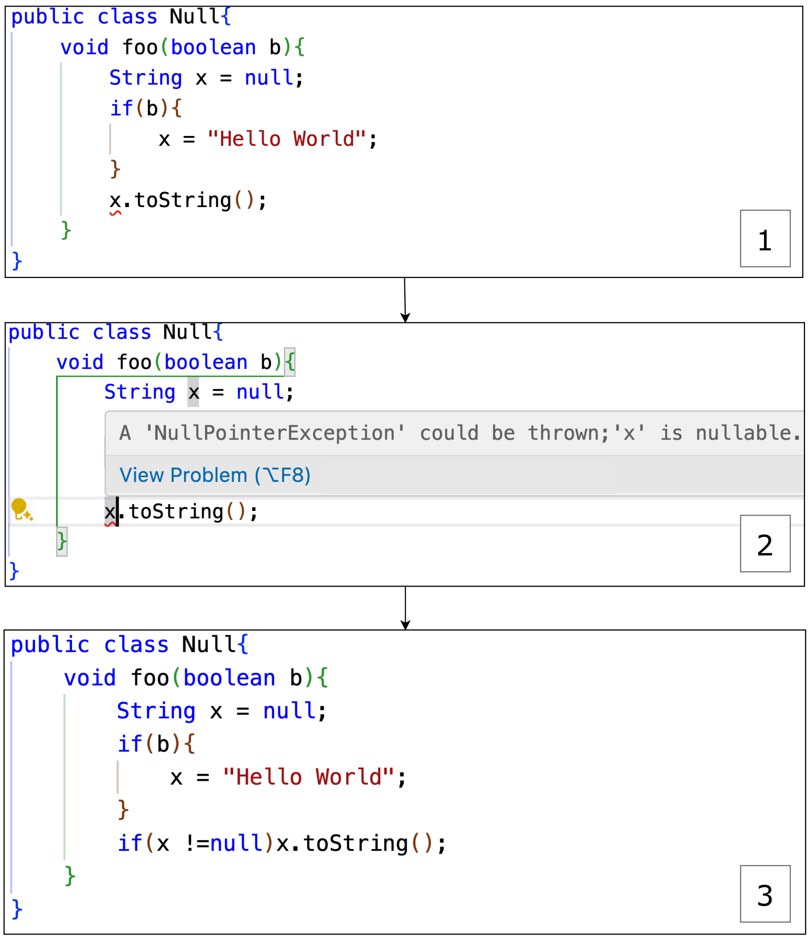
maintaining a record of the active analyses and forwarding events in the editor, such as the save command or opening of a file, to IntraJ. The analysis results are then sent back to MagpieBridge, which subsequently forwards them to the editor, displaying warnings, quick fixes, and explanations to the developer.

Our initial implementation of the client analyses included only the identification of issues, like dead assignments and potential null pointer exceptions. To take advantage of the support from MagpieBridge, we added explanations of the warnings, and also quick fixes for Null Pointer issues.

Figure 11 illustrates an example of interaction between IntraJ and Visual Studio Code. More specifically, it illustrates an instance of a potential `NullPointerException` detected by IntraJ and its representation within the IDE. The lightbulb icon (💡) indicates that a quick fix is available, which can be applied by clicking on the icon.

4.3 CodeProber Integration

To investigate an even tighter integration with the editor, we have integrated IntraJ with CodeProber [Ala+24b], a tool for visualising and exploring the results of compilers and static analysers on an edited program. CodeProber supports exploring partial analysis results such as properties of AST nodes, and is therefore particularly suited for RAG-based tools that use on-demand evaluation. It is browser-based and can support visualizations beyond what is possible via the Language Server Protocol. The visualizations are live and updated as the user edits the analyzed program. Unlike LSP-based tools, CodeProber triggers attribute evaluation on each keystroke, enabling real-time interaction with the analysis results. In contrast, LSP-based tools typically perform analysis less frequently, such as on file save or when a file is opened. As demonstrated in Section 6, IntraJ is fast enough to support the execution of analyses on each keystroke, making it suitable for integration with CodeProber.

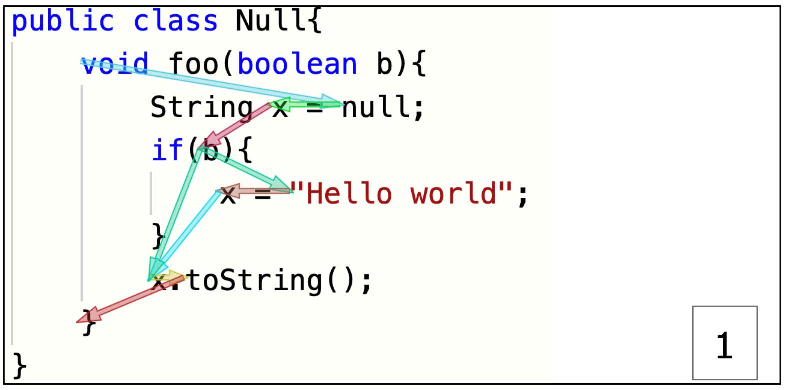


```
public class Null{
    void foo(boolean b){
        String x = null;
        if(b){
            x = "Hello World";
        }
        x.toString();
    }
}
```

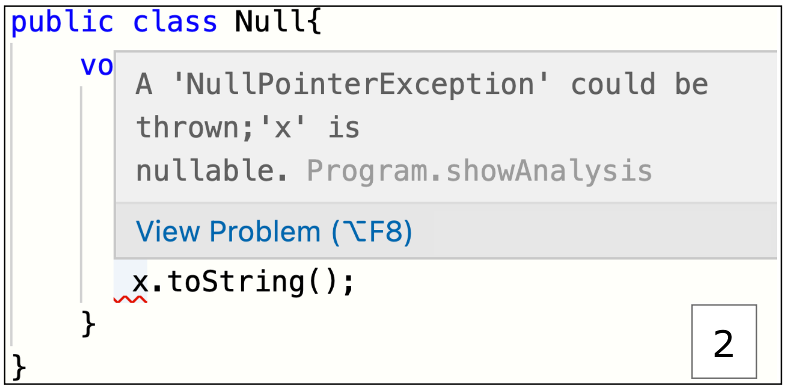
```
public class Null{
    void foo(boolean b){
        String x = null;
        A 'NullPointerException' could be thrown;'x' is nullable.
        View Problem (⌥F8)
        x.toString();
    }
}
```

```
public class Null{
    void foo(boolean b){
        String x = null;
        if(b){
            x = "Hello World";
        }
        if(x !=null)x.toString();
    }
}
```

Figure 11: Bug detection and quick fix in Visual Studio Code using IntraJ. 1. The `NullPointerException` is detected by IntraJ (squiggly line under `x`) with a quick fix available (💡). 2. The user can hover over the warning to see an explanation of the issue. 3. The user can click on the quick fix icon (💡) to apply the fix.



```
public class Null{
    void foo(boolean b){
        String x = null;
        if(b){
            x = "Hello world";
        }
        x.toString();
    }
}
```



```
public class Null{
    vo
        x.toString();
    }
}
```

A 'NullPointerException' could be thrown;'x' is nullable. Program.showAnalysis

Figure 12: Integration of IntraJ in CodeProber. A. The CFG of the method `foo` is superimposed on the source code. (Edge colors are randomly picked.) B. Diagnostics can be accessed by hovering over them, providing explanations of the identified issues.

The example in Figure 12 shows the visual representation of the CFG on top of the source code. The CFG is generated by IntraJ and visualised by CodeProber. It also shows the diagnostics that are displayed when hovering over the squiggly lines. The CFGs and the analysis results are computed automatically at each keystroke, allowing developers to interact with the results in real-time.[3]

5 Extending IntraJ

IntraJ is inherently extensible due to its use of Reference Attribute Grammars [Hed00].

In this section we explore different ways to extend IntraJ and discuss the possibilities for adding new analyses and supporting additional language constructs.

5.1 Extending IntraJ's Functionality

IntraJ can be extended by adding new modules that provide additional attributes and equations to the existing AST. JastAdd provides a modular extension mechanism for RAGs, which allows new modules to be added to IntraJ without modifying the existing codebase. These modules can be independently developed and, if desired, subsequently integrated into IntraJ to provide supplementary functionality. To illustrate this extensibility, we provide examples from various IntraJ submodules: one demonstrating the addition of a new analysis, another showcasing a separate analysis for information flow, and a third example detailing how existing analyses can be extended to support newer Java versions.

5.2 Addition of New Analyses

The existing dataflow and client analyses serve as examples for how to add new dataflow-based analyses. As an example, consider the Implicit Dead Assignment module. It uses results from the ordinary Dead Assignment module and from the Reaching Definitions (RD) module. The key attribute that defines this analysis is the `isImplicitlyDead` attribute, which is shown in Listing 3. This attribute is defined on the `CFGNode` interface, which is implemented by all nodes in the CFG. The attribute definition is written as a method body, but without side-effects, using other attributes as needed (see API Figure 6). It uses the `isDeadAssign` attribute, computed by the ordinary Dead Assignment module. It also uses a local attribute `allUses` (not shown) that in turn is defined using the `inRD` attribute

[3]Courtesy of the CodeProber developer, Anton Risberg Alakküla, a demo version of this integration is available online at `https://github.com/Kevlanche/codeprober-playground`. Enter the code from Figure 12 in the editor. Open a probe for `MethodDecl.showCFG` to get the CFG visualization, and one on `Program.showAnalysis` to get the diagnostics for null-pointer exceptions. Edit the code to see the immediate response.

Listing 3: Definition of the `isImplicitlyDead` attribute, which determines if an assignment is implicilty dead.

```
syn boolean CFGNode.isImplicitlyDead() {
  if (allUses().isEmpty() || this.isDead())
    return false;

  for (CFGNode candidate : allUses()) {
    if (candidate != this && !candidate.isDead() &&
        !candidate.isImplicitlyDead()) {
      return false;
    }
  }
  return true;
}
```

from the Reaching Definition component. Because the analysis is implemented using RAGs, it is automatically evaluated on-demand.

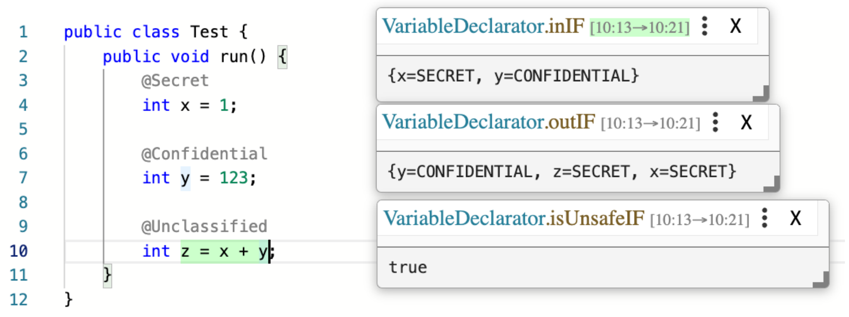
5.3 Information Flow

Another example of an extension to the IntraJ framework is SInfoJ [Sol23], an information flow analysis similar to JFlow [Mye99]. The goal of this analysis is to detect if sensitive data is leaked to untrusted sources. The developer classifies variables according to how sensitive they are. In SInfoJ, the following lattice is used for labeling variables (from lower to higher): `Bottom` $\rightarrow$ `Unclassified` $\rightarrow$ `Confidential` $\rightarrow$ `Secret` $\rightarrow$ `TopSecret`. The analysis can then, for example, identify if a variable labeled as `TopSecret` is leaking information to variables classified with lower security labels.

A simple example is shown in Figure 13, where variables are labelled using Java annotations. The analysis detects that the variable `z` is leaking information, since it is labeled as `Unclassified`, but uses information from variables with security labels `Secret` and `Confidential`.

SInfoJ reuses the CFG from IntraJ but builds its own dataflow analysis as an instance of the monotone framework described in Section 2.2. Variables and their labels are propagated forward in the control flow. Each CFG node has an in and out set, consisting of the variables and their labels before and after the effect of the CFG node, respectively. The transfer function defines the effect, thus, how in is transformed into out.

In the implementation, in and out are defined as the attributes `inIF` and `outIF` (**IF** as in **I**nformation **F**low). Figure 13 shows the value of these attributes for the last variable declaration `z`. In this example, `inIF` is transformed into



```
 1  public class Test {
 2      public void run() {
 3          @Secret
 4          int x = 1;
 5
 6          @Confidential
 7          int y = 123;
 8
 9          @Unclassified
10          int z = x + y;
11      }
12  }
```

```
VariableDeclarator.inIF [10:13→10:21]
{x=SECRET, y=CONFIDENTIAL}

VariableDeclarator.outIF [10:13→10:21]
{y=CONFIDENTIAL, z=SECRET, x=SECRET}

VariableDeclarator.isUnsafeIF [10:13→10:21]
true
```

Figure 13: SINFOJ is an example extension that detects information flow violations (line 10).

`outIF` by adding `z=Secret`, since that is the highest label used in the right-hand side of the assignment. The figure also includes the attribute `isUnsafeIF`, which has the value `true` for this declaration, meaning it is an information flow violation. It may be tedious to annotate all variables with labels, thus some of them can be omitted and automatically derived. For instance, it would be possible to omit the annotation for `z`. Then, the label would be derived to `Secret`, and the declaration would no longer yield a violation.

Part of the definition of the information flow is shown in Listing 4. The `inIF` and `outIF` are implemented like a normal forward analysis, using the attribute `pred` provided by INTRAJ to propagate information forward in the control flow graph. Both attributes compute a value of the type `IFDomain`, which is a mapping from variables to labels. The attribute `inIF` joins all `outIF` of its predecessors by keeping the highest label of each variable. The `outIF` attribute applies the transfer function to `inIF`, e.g., the effect of the CFG node. The transfer function is shown for variable declarations, where the highest label is used from either the annotation or the right-hand side of the declaration (if it has one). The attributes `inIF` and `outIF` are *circular* [MH07b], meaning that they will automatically be solved by a fixed-point iteration algorithm. When defining a circular attribute, a bottom value needs to be given (in this case: `new IFDomain()`).

SINFOJ contains more analysis than shown here, like protecting against conditionals leaking information in control structures. For instance, if a `TopSecret` variable is used in a condition of an `if` statement, then that might leak information to the statements insides the branches. This issue is handled by another analysis also defined using the monotone framework. SINFOJ was implemented by Max Soller as a master thesis. This illustrates that INTRAJ can be extended by students for non-trivial use cases. The Figure 13 is a screenshot of SINFOJ in CODEPROBER, illustrating how attributes easily can be explored when developing analyses.

Listing 4: Part of the definition of information flow in SinfoJ. (Source code from SinfoJ implementation. Complete code available at `https://bitbucket.org/jastadd/infoflow-exjobb-max-soller`)

```
aspect InformationFlowAnalysis {
  syn IFDomain CFGNode.inIF() circular[new IFDomain()] {
    IFDomain res = new IFDomain();
    for (CFGNode p: pred()) {
      res.join(p.outIF());
    }
    return res;
  }

  syn IFDomain CFGNode.outIF() circular[new IFDomain()] {
    return transferFunIF(new IFDomain(inIF()));
  }

  // Transfer function. Default behavior
  syn IFDomain CFGNode.transferFunIF(IFDomain domain) {
    return domain;
  }
  // Transfer function for variable declarations
  eq VariableDeclarator.transferFunIF(IFDomain domain) {
    LabelDomain label = LabelDomain.BOTTOM;
    // Label from annotation
    if (annotatedFlowLabel().isHigherThan(label)) {
      label = annotatedFlowLabel();
    }
    // Label from initializer expression
    if (hasInit()) {
      label = label.returnHighest(getInit().flowLabel());
    }
    // Update label for this variable
    domain.join(this, label);
    return domain;
  }
  ...
}
```

Listing 5: Adding support for ForEach construct to IntraJ.

```
aspect CFG_java5 {
  EnhFor implements CFGNode;
  // Defines the first node that should be traversed when visiting
      an Enhanced For statement

  eq EnhFor.firstNodes() = getExpr().firstNodes();

  // The successor of the collection is the variable declaration
      or the what follows the Enhanced For statement in case the
      collection has been completely traversed.
  eq EnhFor.getExpr().nextNodes() =
      Set.union(getVarDecl().firstNodes(), nextNodes());
  eq EnhFor.getStmt().nextNodes() = getExpr().firstNodes();
  eq EnhFor.getVarDecl().nextNodes() = getStmt().firstNodes();

  eq EnhFor.nextContinue() = getExpr().firstNodes();
}
```

5.4 New Language Constructs

The Java language is constantly evolving, with new language constructs being introduced in almost each new version. As new language constructs are introduced, ExtendJ can be extended to handle them appropriately. IntraJ, being built on top of ExtendJ, can be easily extended as well to support these new features. The possibility of doing this is already demonstrated by the modular support of IntraJ for Java 4 to 8. This guarantees the compatibility of IntraJ with the evolving nature of the Java language.

The code in Listing 5 shows an example of how the CFG is extended to support the *Enhanced For statement* introduced in Java 5. As can be seen from the Listing, the number of lines of code required to support the new language constructs is very small: The first two lines specify that the new construct also serve as CFG nodes. The remaining lines are equations that override default attribute definitions from IntraCFG (`firstNodes`, `nextNodes`, `nextContinue`) in order to define the control flow for the `EnhFor` AST node.

The use of *node type interfaces* further simplifies the process of supporting new language constructs into IntraJ. This approach eliminates the need to modify existing analyses since they rely on the `CFGNode` interface rather than the AST node types. As a result, the addition of new language constructs does not require the modification of existing analyses. For example, by adding CFG support for the `ForEach` construct, all dataflow analyses, including SinfoJ, will automatically handle programs containing `ForEach` statements without additional

Benchmark Name	**LOC**	**Files**	**#Methods**	**Version**	**Application**
`commons-jxpath`	24320	213	2030	1.3	XPath expression processing
`antlr`	36525	192	2070	2.7.2	Parser generator
`jackson-core`	48599	280	3687	commit #`c5b123b`	Core JSON processing library
`pmd`	60749	752	5325	4.2.5	Source code analyzer
`struts`	81394	1111	7023	2.3.22	Web application framework
`joda-time`	86562	330	9324	2.10.13	Date and time library
`jfreechart`	95664	736	6980	1.0.0	Chart library
`fop`	102746	1047	8318	0.95	XSL-FO processing library
`extendj`	147265	396	16025	11.0	Java compiler
`castor`	235745	1711	12643	1.3.3	Data binding framework
`weka`	245719	1223	14952	revision `7806`	Machine learning library
`poi`	329366	2959	23816	3.11	Microsoft Office file processing

Table 1: Evaluated Java benchmarks, including number of lines of code, number of methods, version, and application.

modifications.

6 Evaluation

This section presents the performance evaluation of IntraJ, with a particular focus on its suitability for on-demand analysis in interactive environments. The experiments are designed to simulate the scenario where each compilation unit is analyzed individually, and the analysis results are displayed to the user in real-time.

To assess the performance of IntraJ, we conducted experiments on two different analyses: *Dead Assignment Analysis* and *Null-Pointer Dereference Analysis*. Since the scenario is in an interactive environment, we measure the execution times of the analyses when the JVM is in a steady state. For each compilation unit, the analysis is first computed 25 times to warm up the JVM. Then, we record 25 measurements of the analysis and compute the mean. Between each measurement, the memoized results are explicitly flushed to ensure that each analysis is performed from scratch. The results reported focus exclusively on the analysis time, excluding the time required for program parsing.

The evaluation was performed on a machine with an Intel Core i7-11700K CPU running at 3.60GHz, 128 GB of RAM, and Ubuntu 22.04.3. The benchmarks were executed using the OpenJDK Runtime Environment Zulu 8.50.0.53-CA-linux64, build 1.8.0_275-b01, with the JVM heap size set at 8 GB.

The evaluation of IntraJ uses Java benchmark projects selected from real-world applications, varying in both type and size. These benchmarks included projects from the DaCapo [Bla+06] and Qualitas [Tem+10b] benchmark suites, such as `antlr` and `jfreechart`, alongside additional projects specifically chosen to cover a broad spectrum of applications, e.g., `extendj`. The selected projects ranged from 6,000 to 320,000 lines of code, ensuring a diverse set of benchmarks for the evaluation. A summary of the benchmarks used in the evaluation is presented in Table 1.

6.1 Dead Assignment Analysis

Dead Assignment Analysis is a static analysis technique designed to identify assignments to variables that are never read, indicating potential bugs or unnecessary code.

The results of this analysis are detailed in Table 2.

The analysis times are notably efficient, with the majority of compilation units being processed in under 0.1 seconds. In six out of the twelve benchmarks, IntraJ analyzed all compilation units within this 0.1-second threshold. For the remaining benchmarks, most compilation units that exceeded the 0.1-second mark still completed analysis within 0.1 to 0.2 seconds. No compilation unit required more than 1 second for analysis.

6.2 Null-Pointer Dereference Analysis

Null-pointer Dereference Analysis is a static analysis technique designed to detect potential null-pointer dereference errors within a program. This analysis extends the *May Null* analysis, a flow-sensitive and context-insensitive approach that tracks the potential nullness of variables. To perform the May Null analysis, IntraJ constructs the forward control-flow graph on-demand and computes the predecessor relationship as the inverse of the successor relationship for the relevant variables.

Null-pointer dereference analysis is generally more computationally intensive than dead assignment analysis due to the requirement of constructing control-flow graphs in both directions. Despite the increased computational requirements, the analysis times remain low, with the majority of compilation units being processed in under 0.1 seconds. The results for the *Null-Pointer Dereference Analysis* are presented in Table 4.

For most projects, nearly all files were analyzed in under 0.1 seconds. Specifically, 100% of the files in the commons-jxpath and antlr projects met this threshold. Similarly, over 99% of the files in most other projects were also analyzed within 0.1 seconds.

However, there were a few exceptions. In the extendj benchmark, while 88% of the files were processed in under 0.1 seconds, one particularly large file with a large method (over 6000 lines of code) required approximately 1 second for analysis. Despite this outlier, the overall performance remained efficient.

These results confirm that IntraJ is highly efficient and suitable for use in interactive development environments, even when handling complex and large codebases.

Benchmark	Analysis Time Range	File Count	Mean Analysis Time (s)	Mean File LOC
commons-jxpath	$\leq$ 0.1s	213 (100%)	$0.0015_{\pm 0.0000}$	114
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
antlr	$\leq$ 0.1s	192 (100%)	$0.0027_{\pm 0.0001}$	175
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
jackson-core	$\leq$ 0.1s	280 (100%)	$0.0054_{\pm 0.0001}$	174
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
pmd	$\leq$ 0.1s	751 (99.9%)	$0.0019_{\pm 0.0000}$	69
	0.1s - 0.2s	1 (0.1%)	$0.1634_{\pm 0.0016}$	8913
	0.2s - 1.0s	0	—	—
struts	$\leq$ 0.1s	1111 (100%)	$0.0022_{\pm 0.0000}$	73
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
joda-time	$\leq$ 0.1s	330 (100%)	$0.0118_{\pm 0.0001}$	262
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
jfreechart	$\leq$ 0.1s	735 (99.9%)	$0.0054_{\pm 0.0001}$	129
	0.1s - 0.2s	1 (0.1%)	$0.1358_{\pm 0.0021}$	1132
	0.2s - 1.0s	0	—	—
fop-0.95	$\leq$ 0.1s	1046 (99.9%)	$0.0025_{\pm 0.0000}$	97
	0.1s - 0.2s	1 (0.1%)	$0.1061_{\pm 0.0009}$	1258
	0.2s - 1.0s	0	—	—
extendj	$\leq$ 0.1s	357 (90.2%)	$0.0199_{\pm 0.0002}$	251
	0.1s - 0.2s	21 (5.3%)	$0.1388_{\pm 0.0013}$	860
	0.2s - 1.0s	18 (4.6%)	$0.3393_{\pm 0.0033}$	2202
castor	$\leq$ 0.1s	1711 (100%)	$0.0019_{\pm 0.0000}$	94
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
weka	$\leq$ 0.1s	1216 (99.4%)	$0.0066_{\pm 0.0001}$	190
	0.1s - 0.2s	5 (0.4%)	$0.1203_{\pm 0.0016}$	1635
	0.2s - 1.0s	2 (0.2%)	$0.2959_{\pm 0.0034}$	3488
poi	$\leq$ 0.1s	2954 (99.8%)	$0.0039_{\pm 0.0000}$	108
	0.1s - 0.2s	5 (0.2%)	$0.1519_{\pm 0.0017}$	2031
	0.2s - 1.0s	0	—	—

Table 2: Steady-state performance of IntraJ for *Dead Assignment Analysis* for single compilation units across different Java benchmarks.

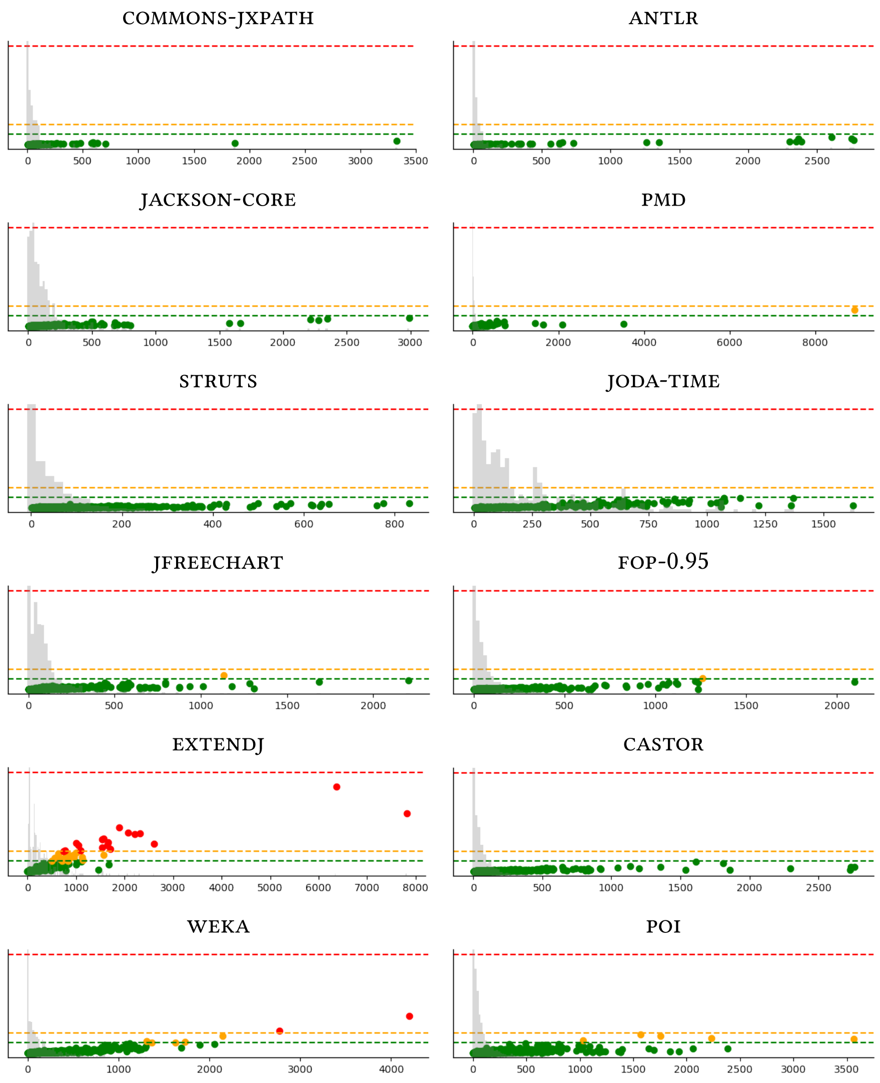


Table 3: Steady-state performance of IntraJ for *Dead Assignment Analysis* for single compilation units across different Java benchmarks. Each plot overlays a histogram and a scatterplot, with the X axis representing LOC for both. The histogram (gray) shows the distribution of compilation unit sizes for each project, with relative frequency on the Y axis. The scatterplot shows the analysis times for each compilation unit on the Y axis, marked green ($\leq$ 0.1 seconds), orange (0.1–0.2 seconds), or red (0.2–1.0 seconds). The dashed lines represent the boundaries at 0.1s (green), 0.2s (orange), and 1.0s (red).

Benchmark	Analysis Time Range	File Count	Mean Analysis Time (s)	Mean File LOC
commons-jxpath	≤ 0.1s	213 (100%)	$0.0021_{\pm 0.0000}$	114
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
antlr	≤ 0.1s	192 (100.00%)	$0.0039_{\pm 0.0000}$	175
	0.1s - 0.2s	0	—	—
	0.2s - 1.0s	0	—	—
jackson-core	≤ 0.1s	279 (99.6%)	$0.0070_{\pm 0.0001}$	163
	0.1s - 0.2s	1 (0.4%)	$0.1033_{\pm 0.0010}$	2990
	0.2s - 1.0s	0	—	—
pmd	≤ 0.1s	751 (99.9%)	$0.0028_{\pm 0.0000}$	69
	0.1s - 0.2s	1 (0.1%)	$0.1249_{\pm 0.0008}$	8913
	0.2s - 1.0s	0	—	—
struts	≤ 0.1s	1110 (99.9%)	$0.0031_{\pm 0.0000}$	73
	0.1s - 0.2s	1 (0.1%)	$0.1115_{\pm 0.0009}$	550
	0.2s - 1.0s	0	—	—
joda-time	≤ 0.1s	327 (99.1%)	$0.0130_{\pm 0.0001}$	255
	0.1s - 0.2s	3 (0.9%)	$0.1078_{\pm 0.0010}$	1097
	0.2s - 1.0s	0	—	—
jfreechart	≤ 0.1s	734 (99.7%)	$0.0064_{\pm 0.0001}$	126
	0.1s - 0.2s	2 (0.3%)	$0.1344_{\pm 0.0013}$	1669
	0.2s - 1.0s	0	—	—
fop	≤ 0.1s	1045 (99.8%)	$0.0033_{\pm 0.0000}$	96
	0.1s - 0.2s	2 (0.2%)	$0.1555_{\pm 0.0019}$	1239
	0.2s - 1.0s	0	—	—
extendj	≤ 0.1s	349 (88.1%)	$0.0253_{\pm 0.0003}$	239
	0.1s - 0.2s	31 (7.8%)	$0.1500_{\pm 0.0013}$	1073
	0.2s - 1.0s	16 (4.0%)	$0.3632_{\pm 0.0035}$	1903
castor	≤ 0.1s	1710 (99.9%)	$0.0025_{\pm 0.0000}$	93
	0.1s - 0.2s	1 (0.1%)	$0.1286_{\pm 0.0013}$	1610
	0.2s - 1.0s	0	—	—
weka	≤ 0.1s	1210 (98.9%)	$0.0085_{\pm 0.0001}$	184
	0.1s - 0.2s	10 (0.8%)	$0.1260_{\pm 0.0012}$	1405
	0.2s - 1.0s	3 (0.3%)	$0.3431_{\pm 0.0035}$	3041
poi	≤ 0.1s	2948 (99.6%)	$0.0048_{\pm 0.0001}$	107
	0.1s - 0.2s	9 (0.3%)	$0.1344_{\pm 0.0013}$	1276
	0.2s - 1.0s	2 (0.1%)	$0.2144_{\pm 0.0019}$	1902

Table 4: Steady-state performance of IntraJ for *Null Pointer Dereference Analysis* for single compilation units across different Java benchmarks.

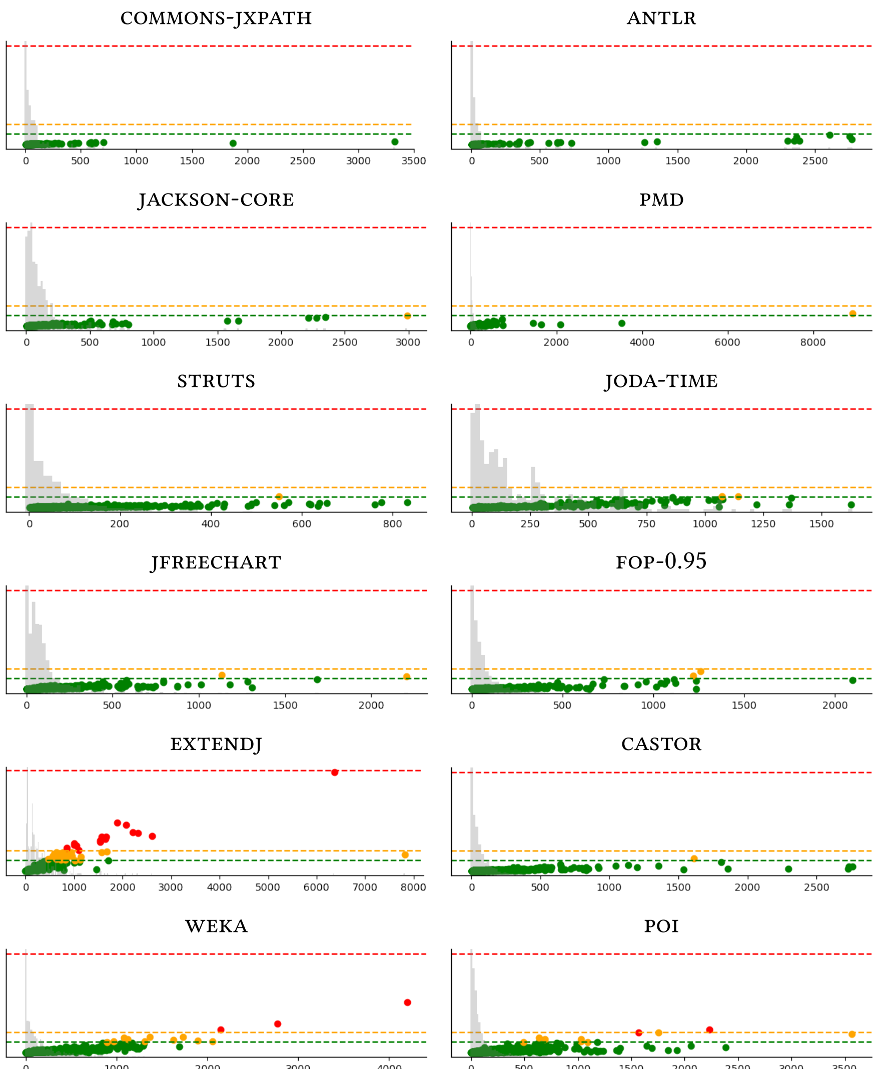


Table 5: Steady-state performance of IntraJ for *Null Pointer Dereference Analysis* for single compilation units across different Java benchmarks. Each plot overlays a histogram and a scatterplot, with the X axis representing LOC for both. The histogram (gray) shows the distribution of compilation unit sizes for each project, with relative frequency on the Y axis. The scatterplot shows the analysis times for each compilation unit on the Y axis, marked green ($\leq$ 0.1 seconds), orange (0.1–0.2 seconds), or red (0.2–1.0 seconds). The dashed lines represent the boundaries at 0.1s (green), 0.2s (orange), and 1.0s (red).

7 Related Work

The challenge of balancing analysis complexity with tool responsiveness is a well-known issue in the field of static analysis. Existing extensible static analysis frameworks like SpotBugs [Spo], Soot [Lam+11], or Infer [CD11] are generally designed for throughput, rather than responsiveness, reflecting their original intended use for batch program analysis. Depending on the internal architecture, increasing responsiveness for interactive use may require nontrivial re-engineering.

For example, Distefano et al. manually re-engineered parts of Infer [Dis+19] to support incremental updates, in order to scale to larger code bases with frequent changes. Arzt et al. exploited properties of the IFDS/IDE analysis framework underlying Soot to incrementalise analysis [AB14], but still needed to make some architectural adjustments.

Prior work has demonstrated strategies that allow analysis frameworks to automatically incrementalise declaratively specified analyses. Dura et al. demonstrate this at the file level [DRS21] for various bug checkers, while Szabo et al. show fine-grained incrementality for a points-to analysis [SEB21]. Both approaches use declarative logic programming to specify their analyses.

IntraJ builds on the RAG framework JastAdd [HM01] that similarly provides declarative interfaces between the attributes computed by different components, but specifies these components in Java, instead of declarative logic programming. This approach implicitly ensures that IntraJ-based analyses are demand-driven.

Söderberg et al. [SH12] proposed a strategy for using dynamic dependency tracking to extend the demand-driven RAG evaluation model to an incremental analysis model that can re-use information from unchanged parts of the program, potentially further increasing reactivity during interactive editing. We expect that this approach would be effective for IntraJ, again without requiring changes to the analyses implemented on top of IntraJ.

8 Conclusions and Future Development

In this paper, we have presented IntraJ, a responsive and extensible framework for intraprocedural control-flow and dataflow analysis for Java source code. By leveraging on-demand evaluation and Reference Attribute Grammars, IntraJ provides real-time analysis results to the programmer, without any noticeable latency in the development environment.

We have demonstrated how IntraJ can be used in different contexts where the programmer can benefit from on-demand analysis, including a command line interface, an editor integration based on LSP, and an integration into the debugging tool CodeProber.

Additionally, we have exemplified how IntraJ can be extended with new on-demand client analyses by writing them as RAG specifications.

In the future, we plan to investigate how RAGs can be used to extend the IntraJ analyses to also support interprocedural analyses. We also aim to expand the range of analyses supported by IntraJ, in particular towards detection of security bugs and vulnerabilities. Furthermore, we see many interesting opportunities for building more interactive exploration tooling for static analysis, based on CodeProber. For example, it would be interesting to generate interactive views of the CFG, perhaps similar to Figure 5 or 8.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

References

[Ala+24b] Anton Risberg Alaküla et al. "Property probes: Live exploration of program analysis results". In: *J. Syst. Softw.* 211 (2024), p. 111980.

[AB14] Steven Arzt and Eric Bodden. "Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 288–298.

[Aye+08b] Nathaniel Ayewah et al. "Using static analysis to find bugs". In: *IEEE software* 25.5 (2008), pp. 22–29.

[Bla+06] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190.

[CD11] Cristiano Calcagno and Dino Distefano. "Infer: An automatic program verifier for memory safety of C programs". In: *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer. 2011, pp. 459–465.

[Cop05] Tom Copeland. *PMD applied*. Vol. 10. Centennial Books Arexandria, Va, USA, 2005.

[Dis+19] Dino Distefano et al. “Scaling static analyses at Facebook”. In: *Commun. ACM* 62.8 (2019), 62–70.

[DRS21] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. “JavaDL: Automatically Incrementalizing Java Bug Pattern Detection”. In: *Proceedings of the ACM on Programming Languages.* Virtual: ACM, 2021.

[EH07b] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada.* Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[Far86] Rodney Farrow. “Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986.* ACM, 1986, pp. 85–98.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000).

[HM01] Görel Hedin and Eva Magnusson. “JastAdd - a Java-based system for implementing front ends”. In: *Electron. Notes Theor. Comput. Sci.* 44.2 (2001), pp. 59–78.

[HM03] Görel Hedin and Eva Magnusson. “JastAdd—an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[JS86] Larry G. Jones and Janos Simon. “Hierarchical VLSI Design Systems Based on Attribute Grammars”. In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986.* ACM Press, 1986, pp. 58–69.

[KU77] John B Kam and Jeffrey D Ullman. “Monotone data flow analysis frameworks”. In: *Acta informatica* 7.3 (1977), pp. 305–317.

[Kil73] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL ’73. Boston, Massachusetts: Association for Computing Machinery, 1973, 194–206.

[Knu68] Donald E Knuth. “Semantics of context-free languages”. In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[Lam+11] Patrick Lam et al. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011).* Vol. 15. 35. 2011.

[LDB19b] Linghui Luo, Julian Dolby, and Eric Bodden. "MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper)". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019).* Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 21:1–21:25.

[MH07b] Eva Magnusson and Görel Hedin. "Circular reference attributed grammars—their evaluation and applications". In: *Science of Computer Programming* 68.1 (2007), pp. 21–37.

[Mye99] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999.* Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 228–241.

[Nad22] Ayman Nadeem. "Human-Centered Approach to Static-Analysis-Driven Developer Tools". In: *Commun. ACM* 65.3 (2022), 38–45.

[Nie94] Jakob Nielsen. *Usability engineering.* Morgan Kaufmann, 1994.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[Öqv18] Jesper Öqvist. "Contributions to Declarative Implementation of Static Program Analysis". PhD thesis. Lund University, Sweden, 2018.

[Rio+21] Idriss Riouak et al. "A Precise Framework for Source-Level Control-Flow Analysis". In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE. 2021, pp. 1–11.

[Sad+15] Caitlin Sadowski et al. "Tricorder: Building a Program Analysis Ecosystem". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1.* Ed. by Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum. IEEE Computer Society, 2015, pp. 598–608.

[Saw99] Kathy Sawyer. "Mystery of Orbiter Crash Solved". In: *Washington Post* (Oct. 1, 1999). Accessed: 2023-01-27, A1.

[SH12] Emma Söderberg and Görel Hedin. “Incremental evaluation of reference attribute grammars using dynamic dependency tracking”. In: (2012). LU-CS-TR:2012-249 (2012).

[Sol23] Max Soller. *SinfoJ: A simple Information Flow Analysis with Reference Attribute Grammars*. Master’s thesis. Available at `http://lup.lub.lu.se/student-papers/record/9149210`. 2023.

[Spo] *SpotBugs*. `https://spotbugs.github.io/`. Accessed: 2023-02-17.

[SEB21] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. “Incremental whole-program analysis in Datalog with lattices”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 1–15.

[Tem+10b] Ewan Tempero et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. In: *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. Dec. 2010, pp. 336–345.

JFeature: Know Your Corpus

Abstract

Software corpora are crucial for evaluating research artifacts and ensuring repeatability of outcomes. Corpora such as DaCapo and Defects4J provide a collection of real-world open-source projects for evaluating the robustness and performance of software tools like static analysers. *However, what do we know about these corpora? What do we know about their composition? Are they really suited for our particular problem?* We developed JFeature, an extensible static analysis tool that extracts syntactic and semantic features from Java programs, to assist developers in answering these questions. We demonstrate the potential of JFeature by applying it to four widely-used corpora in the program analysis area, and we suggest other applications, including longitudinal studies of individual Java projects and the creation of new corpora.

1 Introduction

The impact of our research in computer science is bounded by our ability to demonstrate and communicate how effective our techniques and theories really are. For research on software tools, the dominant methodology for demonstrating

Idriss Riouak, Görel Hedin, Christoph Reichenbach and Niklas Fors.
"JFeature: Know Your Corpus!".
In *22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 236–241.* Limassol, Cyprus, 2022, IEEE.
Paper DOI: 10.1109/SCAM55253.2022.00033.

effectiveness is to apply these tools to "real-life" software development tasks and to measure how well they perform. Blackburn et al. [Bla+08] outline this process in considerable detail, highlighting the need for *appropriate experimental design* (to construct experiments), *relevant workloads* (to obtain relevant data from the experiments), and *rigorous analysis* (to obtain rigorously justified insights from experimental data). The strength of our insights is then bounded by the weakest link in this chain.

Carefully curated, pre-packaged workloads such as the DaCapo Benchmark suite [Bla+06], Defects4J [JJE14], the Qualitas Corpus [Tem+10a], and XCorpus [Die+17] can help ensure that we use relevant workloads. However, no software corpus aims to be representative of all software, and for any given research question there may not be any one corpus designed to answer that question, so we must still validate that the corpus we choose is relevant to what we want to show.

For instance, the DaCapo corpus aims to provide benchmarks with "more complex code, richer object behaviors, and more demanding memory system requirements" [Bla+06] than the corpora that preceded it, and it systematically demonstrates complex interactions between architecture and the Java Run-Time Environment, whereas Defects4J collects "real bugs to enable reproducible studies in software testing research" [JJE14]. Despite DaCapo's focus on run-time performance and Defects4J's focus on software testing, both suites have seen heavy use in research that they were not explicitly intended for, including the authors' own work in static analysis [Rio+21; DRS21] (using Defects4J), and in compilers [EH07a] and dynamic invariant checking [Rei+10] (for DaCapo).

For each of these ostensible mis-uses, the authors selected the corresponding benchmark corpus as the highest-quality corpus they were aware of whose original purpose seemed sufficiently close to the intended experiments. This divergence between research question and corpus purpose required the authors to carefully re-validate the subset of the corpus that they had selected by hand.

In this paper, we argue that there is a need for increased automation and decision support for selecting benchmarks for specific research questions, and present JFeature, a static analysis tool designed to help researchers in this process. JFeature identifies how often a Java project uses key Java features that are significant for different types of software tools. JFeature operates at the source code level, and is capable of identifying not only local syntactic features that may be challenging to encode in regular expression search tools like `grep`, but also complex semantic features that depend on types and libraries. We have implemented JFeature in the JastAdd [HM03] ecosystem as an extension of the ExtendJ [EH07a] Java Compiler. This implementation architecture gives easy access to types and other properties computed by the compiler, and also supports extensibility, allowing researchers to adapt the analysis to fit their specific needs.

We demonstrate JFeature by running it on several widely-used corpora, specifically the DaCapo, Defects4J, Qualitas, and XCorpus corpora.

Our main contributions are:

- JFeature as an example of a tool for extracting information about the features used in Java source code, and
- An overview over JFeature's key insights on the DaCapo, Defects4J, Qualitas, and XCorpus corpora.

The rest of this paper is organised as follows: Section 2 introduces JFeature and discusses the design decisions that underpin the tool. Section 3 shows the results of applying JFeature to the four corpora. Section 4 illustrates how JFeature can be extended to extract new features, taking advantage of properties in the underlying Java compiler. Section 5 outlines future applications of JFeature. Section 6 discusses related work, and Section 7 summarizes our conclusions.

2 JFeature: automatic feature extraction

We have designed JFeature as an extension of the ExtendJ extensible Java compiler. ExtendJ is implemented using Reference Attribute Grammars (RAGs) [Hed00] in the JastAdd metacompilation system. ExtendJ is a full Java compiler, feature-compliant for Java 4 to 7 and close to being feature-compliant for Java 8[1]. In building compilers by means of attribute grammars [Knu68], the abstract syntax tree (AST) is annotated with properties called *attributes* whose values are defined using equations over other attributes in the AST. RAGs extend traditional attribute grammars by supporting that attributes can be links to other AST nodes. ExtendJ annotates the AST with attributes that are used for checking compile-time errors and for generating bytecode. Example attributes include links from variable uses to declarations, links from classes to superclasses, types of expressions, etc. These attributes are exploited by JFeature to easily identify AST nodes that match a particular feature of interest.

2.1 Java version features

There are many different features that could be interesting to investigate in a corpus. As the default for JFeature, we have defined feature sets for different versions of Java, according to the Java Language Specification (JLS). A user can then run JFeature to, e.g., investigate if a corpus is sufficiently new, or select only certain projects in a corpus, based on what features they use. If desired, a user can extend the feature set for a specific purpose.

In recent years there have been several new releases of the Java language. Currently, Java 18 is the latest version available. However, most projects utilise Java 8 or Java 11, both of which are long-term support releases (LTS).

[1]`https://extendj.org/compliance.html`

Feature		Kind	
		Syn	Sem
Java 1.1 - 4, 1997-2002 – [Java; Javb; Javc; Javd]			
Inner Class			✓
java.lang.reflect.*			✓
Strictfp		✓	
Assert Stmt		✓	
Java 5, 2004 – [Javg; Jave]			
Annotated Compilation Unit		✓	
Annotations	Use	✓	
	Decl	✓	
Enum	Use		✓
	Decl	✓	
Generics	Method	✓	
	Constructor	✓	
	Class	✓	
	Interface	✓	
Enhanced For		✓	
Varargs		✓	
Static Import		✓	
java.util.concurrent.*			✓
Java 7, 2011–[Javf]			
Diamond Operator		✓	
String in Switch			✓
Try with Resources		✓	
Multi Catch		✓	
Java 8, 2014– [Javh]			
Lambda Expression		✓	
Constructor Reference			✓
Method Reference			✓
Intersection Cast		✓	
Default Method		✓	

Table 1: Major changes in the Java language up to Java 8.

Table 1 summarises the main features introduced in each Java release after the initial release (JDK 1.0) up to Java 8. We have classified the features into either

- `Syntactic`: can be identified using a context-free grammar, or
- `Semantic`: additionally needs context-dependent information such as nesting structure, name lookup, or types.

While most features are syntactic, there are several features that are semantic, and where the attributes available in the compiler are very useful for identification of the features.

Given any Java 8 project, JFeature collects all the feature usages, grouped by release version. By default, JFeature supports twenty-six features[2], but users may extend the tool and add their own. We have chosen these features by looking at each Java release note [Java; Javb; Javc; Javd; Javg; Jave; Javf; Javh]. We included features that represent the most significant release enhancements, i.e., libraries or native language constructs whose use significantly impacts program semantics. In particular, we included the usage of two libraries, JAVA.UTIL.CONCURRENT.* and JAVA.LANG.REFLECT.*, because their usage may be pertinent for the evaluation of academic static analysis tools.

2.2 Collecting features

To collect features, JFeature uses *collection attributes* [Boy96; MEH07b], also supported by JastAdd. Collection attributes aggregate information by combining contributions that can come from anywhere in the AST. A *contribution clause* is associated with an AST node type, and defines information to be included, possibly conditionally, in a particular collection. Both the information and the condition can be defined by using attributes.

For JFeature, we use a collection attribute, `features`, on the root of the AST. The value of `features` is a set of objects, each defined by a contribution clause somewhere in the AST. The objects are of type `Feature` that models essential information about the extracted feature: the Java version, feature name, and absolute path of the compilation unit where the feature was found.

Figure 1 shows an example with JastAdd code at the top of the figure, and below that, an example program and its attributed AST. The `features` collection is defined on the nonterminal `Program`, which is the root of the AST (line 1). Then two features are defined, STRICTFP and STRING IN SWITCH (lines 3-5 and 7-9).

STRICTFP is a syntactic feature that corresponds to the modifier `strictfp`. In ExtendJ, modifiers are represented by the nonterminal `Modifiers` which contains a list of modifier keywords, e.g., `public`, `static`, `strictfp`, etc. To find out if one of the keywords is `strictfp`, ExtendJ defines a boolean attribute `isStrictfp` for `Modifiers`. To identify the STRICTFP feature, a contribution clause is defined on the nonterminal `Modifiers` (line 3), and the `isStrictfp` attribute is used for conditionally adding the feature to the collection (line 5). The absolute path is computed using other attributes in ExtendJ: `getCU` is a reference to the AST node for the enclosing compilation unit, and `path` is the absolute path name for that compilation unit (line 4).

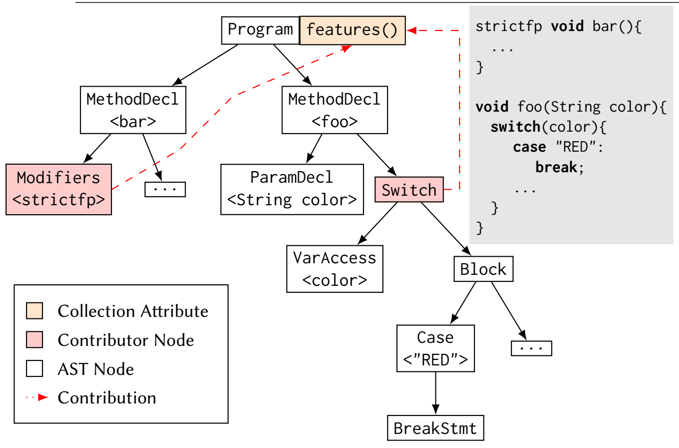
STRING IN SWITCH is a semantic feature in that it depends on the type of the switch expression. It cannot be identified with simple local AST queries or

[2]The complete implementation can be found at `https://github.com/lu-cs-sde/JFeature`.

```
coll HashSet<Feature> Program.features();

Modifiers contributes
  new Feature("JAVA2", "Strictfp", getCU().path())
  when isStrictfp() to Program.features();

Switch contributes
  new Feature("JAVA7", "StringInSwitch", getCU().path())
  when getExpr().type().isString() to Program.features();
```



```
strictfp void bar(){
  ...
}

void foo(String color){
  switch(color){
    case "RED":
      break;
    ...
  }
}
```

Figure 1: Example definitions of features.

regular expressions. Here, the contribution clause is defined on the nonterminal `Switch`, and the feature is conditionally added if the type of the switch expression is a string. ExtendJ attributes used here are `type` which is a reference to the expression's type, and `isString` which is a boolean attribute on types.

3 Corpora Analysis

We used JFeature to analyse four widely used corpora, to investigate to what extent the different Java features from Table 1 are used. We picked the newest available version of each of the corpora.

3.1 Corpora Description

DaCapo Benchmark Suite

Blackburn et al. introduced it in 2006 as a set of general-purpose (i.e., library), freely available, real-world Java applications. They provided performance measurements and workload characteristics, such as object size distributions, allocation rates and live sizes. Even if the primary goal of the DaCapo Benchmark

Table 2: Corpora Analysis. Each entry represents the total number of projects utilising the respective feature.

Corpus (# Projects)	JAVA 1.1 - 4				JAVA 5												
	Inner Class	java.lang.reflect.*	Strictfp	Assert Stmt	Annotated CU	Annot.		Enum		Generics				Enhanced For	VarArgs	Static Import	java.util.concurrent.*
						Use	Decl	Use	Decl	Method	Constructor	Class	Interface				
DaCapo (15)	15	12	2	5	0	8	4	14	8	6	2	7	4	8	7	5	7
Defects4J (16)	16	15	1	8	0	15	7	16	14	13	3	12	10	15	13	14	14
Qualitas (112)	109	100	4	51	9	67	35	109	45	55	7	59	41	68	49	46	50
XCorpus (76)	74	65	4	28	3	39	21	74	32	31	4	35	22	39	28	25	27

Table 3: Corpora Analysis. Each entry represents the total number of projects utilising the respective feature.

Corpus (# Projects)	JAVA 7				JAVA 8				
	Diamond Operator	String in Switch	Try w/ Resources	Multi Catch	Lambda Expression	Constructor Reference	Method Reference	Intersection Cast	Default Method
DaCapo (15)	2	1	3	2	2	0	2	0	0
Defects4J (16)	14	7	13	10	10	5	8	1	1
Qualitas (112)	1	1	1	1	0	0	0	0	0
XCorpus (76)	4	2	3	3	2	1	2	0	0

Suite is intended as a corpus for Java benchmarking, there are several instances of frontend and static analysers evaluation. For evaluation, we used version 9.12-bach-MR1 released in 2018.

Defects4J

introduced by Just et al., is a bug database consisting of 835 real-world bugs from 17 widely-used open-source Java projects. Each bug is provided with a test suite and at least one failing test case that triggers the bug. Defects4J found many uses in the program analysis and repair community. For evaluation, we used version 2.0.0 released in 2020.

Qualitas Corpus

is a set of 112 open-source Java programs, characterised by different sizes and belonging to different application domains. The corpus was specially designed for empirical software engineering research and static analysis. For evaluation, we used the release from 2013 (20130901).

XCorpus

is a corpus of modern real Java programs with an explicit goal of being a target for analysing dynamic proxies. XCorpus provides a set of 76 executable, real-world Java programs, including a subset of 70 programs from the Qualitas Corpus. The corpus was designed to overcome a lack of a sufficiently large and general corpus to validate static and dynamic analysis artefacts. The six additional projects added in the XCorpus make use of dynamic language features, i.e., invocation handler. For evaluation, we used the release from 2017.

3.2 Evaluation

Methodology

To compute complete semantic analysis with JFeature and ExtendJ, all dependent libraries and the classpath are needed for each analysed project. Unfortunately, different projects use different conventions and build systems, making automatic extraction of this information difficult. Therefore, for our study of the full corpora, we decided to extract features depending only on the language constructs and the standard library, but that did not require analysis of the project dependencies. This way, we could run JFeature on these projects without any classpath (except for the default standard library).

Table 2 and Table 3 show an overview of the results of the analysis. For each corpus, we report the number of projects that use a particular feature from Table 1. More detailed results, including the results for all 26 features, and counts for each individual project, are available at `https://github.com/lu-cs-sde/JFeature/blob/main/features.xlsx`.

For standard libraries, like `java.lang.reflect.*` and `java.util.concurrent.*`, we count all variable accesses, variable declarations, and method calls whose type is hosted in the respective package.

While ExtendJ mostly complies to the JLS version 8, its Java 8 type inference support diverges from the specification in several corner cases. As Table 2 and Table 3 show, these limitations did not affect DaCapo, but they did surface in 43 method calls in 9 projects (2 projects in Defects4J that we manually inspected to validate our findings).

Corpora overlap

Corpus	Projects															
	Mock		Asm		Derby		JUnit		Tomcat		Xerces		JRep		JMeter	
	1.1	2.0	3.3	5.2	10.14	10.9	4.10	4.12	6.0	7.0	2.8	2.10	1.1	3.7	2.5	3.1
DaCapo			✓		✓			✓	✓		✓					
Defects4J		✓														
Qualitas						✓	✓			✓		✓		✓	✓	
XCorpus	✓			✓						✓		✓	✓			✓

Table 4: Projects used in the corpora with different versions.

Figure 2 shows the overlap between the four corpora as two Venn diagrams where each number represents a project. In the left diagram, two versions of the same project are counted as two separate projects. In the right diagram, we only consider the project name, disregarding the version. From the left diagram, we can see that Defects4J does not overlap with any other corpus analysed. As expected,

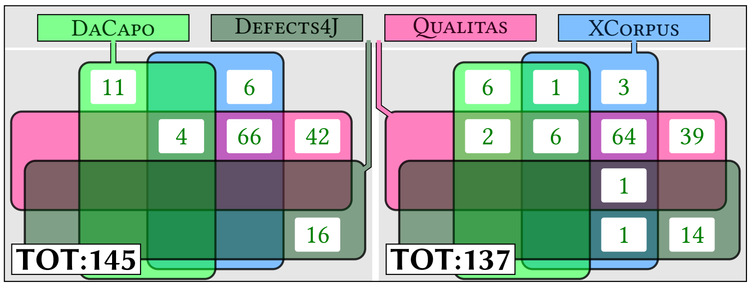


Figure 2: Project overlap. In the left diagram, two projects with the same name but different versions are counted as distinct—the diagram to the right shows overlap when versions are disregarded.

most of the projects are shared across Qualitas and XCorpus as XCorpus was built as an extension of Qualitas. From the diagrams, we can see that eight projects (145-137) are used among the corpora, but with different versions. Table 4 details these projects and versions.

Discussion

Table 2 provides insight into the features utilised by each project. Using Qualitas Corpus as an illustration, we see that `strictfp` is only used in four projects. Similarly, in DaCapo, fewer than fifty percent of the projects use concurrency libraries. With JFeature, we can achieve a fine-grained classification of the properties. We can, for instance, distinguish between uses and declarations of annotations, and when it comes to generics, we can distinguish between the declarations of generic methods, classes, and interfaces, providing the user with a better comprehension of the corpus. It is apparent that most projects utilise only Java 4 and Java 5 features. With the exception of Defects4J, few projects employ Java 7 and Java 8. Indeed, this table reveals that Defects4J is the most modern corpus, as nine of the fourteen assessed applications utilise at least one of the observed Java 8 features.

4 Extensibility

Extensibility is one of the key characteristics of JFeature. Users can create new queries to extract additional features, making use of all attributes available in the ExtendJ compiler. We illustrate this by adding a new feature, Overloading, that measures the number of overloaded methods in the source code. Listing 1 shows the JastAdd code for this: we define a new boolean attribute, `isOverloading`, that checks if a method is overloaded. We then use this attribute to conditionally contribute to the `features` collection, only for overloaded method declarations. The attribute `isOverloading` is defined using several ExtendJ attributes: the attribute `hostType` is a reference to the enclosing type declaration of the method declaration. A type declaration, in turn, has an attribute `methodsNameMap` that holds references to all methods for that type declaration, both local and inherited. If there is more than one method for a certain name, that name is overloaded.

Listing 1: Definition of the Overloading feature

```
MethodDecl contributes
  new Feature("JAVA1", "Overloading", getCU().path())
  when isOverloading() to Program.features();

syn boolean MethodDecl.isOverloading()
  = hostType().methodsNameMap().get(getID()).size() > 1;
```

PROJECTS	~ KLOC	NUMBER of METHODS	OVERLOADED METHODS	%
antlr-2.7.2	34	2081	358	17,2
commons-cli-1.5.0	6	585	76	13
commons-codec-1.16-rc1	24	1812	422	23,3
commons-compress-1.21	71	5359	571	10,7
commons-csv-1.90	8	716	93	13
commons-jxpath-1.13	24	2030	167	8,23
commons-math-3.6.1	100	7229	1779	24,6
fop-0.95	102	8317	666	8,01
gson-2.90	25	2289	125	5,46
jackson-core-2.13.2	48	3687	839	22,8
jackson-dataformat-2.13	15	1122	161	14,3
jfreechart-1.0.0	95	6980	1000	14,3
joda-time-2.10	86	9324	1257	13,5
jsoup-1.14	25	2556	408	16
mockito-4.5.1	19	2054	318	15,5
pmd-4.2.5	60	5324	1021	19,2

Table 5: Results from the OVERLOADING feature.

For the computation to work, it is necessary to supply the classpath, so that ExtendJ can find the classfiles for any direct or indirect supertypes of types in the analysed source code. We analysed 16 distinct projects for which we successfully extracted the classpaths and dependencies required for ExtendJ compilation. The results provided by JFeature for the sixteen projects are summarised in Table 5. As can be seen, each project has overloaded methods. In some cases, such as `commons-codec`, `commons-math`, and `jackson-core`, more than one fifth of the methods are overloaded.

OVERLOADING is a good example of a feature that requires semantic analysis—it can not be computed by a simple pattern match using regular expressions or a context-free grammar.

5 Use cases for JFeature

We already discussed two possible use cases for JFeature: corpus evaluation (Section 3), and extending JFeature to identify specific features (Section 4). In this section, we discuss two additional use cases: longitudinal studies and project mining.

5.1 Longitudinal Study

JFeature can be used to conduct longitudinal studies, i.e., changes occurring over time. As an example, we conducted a study on Mockito and its evolution on the adaption of Java 8 features over time. Mockito is one of the most popular Java mocking frameworks and has an extensive history with over 5,000 commits. Java 6 was utilised by Mockito until version 2.9.x. With version 3.0.0, Java 8 was adopted.

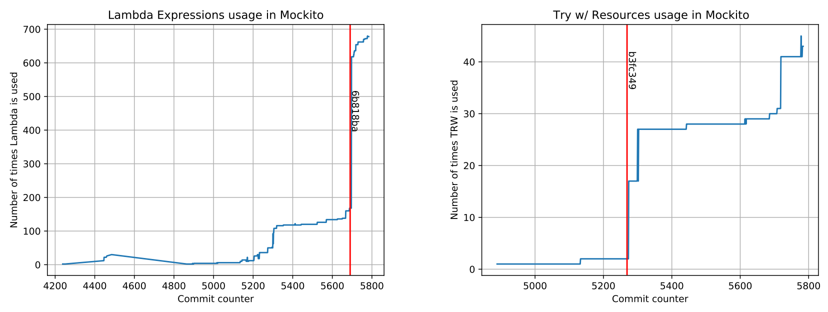


Figure 3: Usage of Lambda Expressions and Try With Resources in Mockito over time.

The evolution of the occurrences of Lambda Expressions and Try With Resources is depicted in Figure 3. As can be seen, at commit number 5269[3], there is a substantial increase in utilisation of try with resources, whereas at commit number 5696[4], there is a significant increase in the use of lambda expressions.

5.2 Project mining

Contemporary revision control hosting services (GitHub[5], GitLab[6], bitbucket[7]) offer uniform interfaces to the source code of millions of software projects. These interfaces enable researchers to "mine" software projects at scale, filtering by certain predefined properties (e.g., the number of users following the project or the main programming language). For example, the *GitHub Java Corpus* [AS13] collects almost 15,000 projects from GitHub, filtered to only include Java projects that have been forked at least once. Combining JFeature with these query mechanisms allows researchers to select projects by more detailed syntactic and semantic features. For instance, a corpus suitable for answering questions

[3]Commit: b3fc349.
[4]Commit: 6b818ba.
[5]`https://github.com`
[6]`https://gitlab.com`
[7]`https://bitbucket.org`

about race detection [Li+14] could select projects that make explicit use of JAVA.UTIL.CONCURRENT.*, while an exploration of functional programming patterns [Cok18] could select projects that use LAMBDA EXPRESSIONS and METHOD REFERENCES.

6 Related work

Existing tools for code metrics are usually focused on code quality metrics, rather than what language features are used, and typically analyse the intermediate representation rather than the source code. One example is the CKJM tool [Spi05] for the Chidamber and Kemerer metrics [CK94]. Another example, that more closely resembles ours, is jCT, an extensible metrics extractor for Java 6 IL-Bytecode, introduced by Lumpe et al. [LMG11], in 2011. Like us, they evaluated their tool on Qualitas Corpus; however, because jCT works only on annotated bytecode and not on source code, the number of features that can be extracted is limited. A significant amount of information is lost during the compilation of Java source code to Java bytecode. For example, enhanced `for` statements, diamond operators and certain annotations, such as `@Override`, are not present in the bytecode. For XCorpus, the authors analysed the language features used, and a summary was presented in their paper [Die+17]. They also analysed the bytecode, which was implemented using the visitor pattern.

A way to improve the user experience would be to integrate JFeature with a visualisation tool like *Explora* [MLN15]. The idea behind *Explora* is to provide to the user a visualisation tool designed for simultaneous analysis of multiple metrics in software corpora. Finally, JFeature may be enhanced by incorporating automated dependency extractors, such as MagpieBridge's *JavaProjectService* [LDB19a], to infer and download libraries automatically. Currently, JavaProjectService infers the dependencies for projects using *Gradle* or *Maven* as build system.

7 Conclusions

We have presented JFeature, a declarative and extensible static analysis tool for the Java programming language that extracts syntactic and semantic features. JFeature comes with twenty-six predefined queries and can be easily extended with new ones.

We ran JFeature on four widely used corpora: the DaCapo Benchmark Suite, Defects4J, Qualitas Corpus, and XCorpus. We have seen that, among the corpora, Java 1-5 features are predominant. This leads us to conclude that some of the corpora may be less suited for the evaluation of tools that address features in Java 7 and 8.

We have illustrated how JFeature can be extended to capture semantically complex features by writing the queries as attribute grammars, extending a full

Java compiler. This allows powerful queries to be written that can make use of all the compile-time properties computed by the compiler.

We discussed several possible use cases for JFeature: evaluation of corpora, mining software collections to create new corpora, and longitudinal studies of how projects have evolved with regard to the use of language features. We also note that for some features to be analysed, the full classpath and dependencies are required. An interesting future direction is therefore to combine JFeature with recent tools that support automatic extraction of such information from projects that follow common build conventions.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[AS13] Miltiadis Allamanis and Charles Sutton. “Mining Source Code Repositories at Massive Scale using Language Modeling”. In: *The 10th Working Conference on Mining Software Repositories*. IEEE. 2013, pp. 207–216.

[Bla+06] S. M. Blackburn et al. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. In: *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190.

[Bla+08] Stephen M Blackburn et al. “Wake up and smell the coffee: Evaluation methodology for the 21st century”. In: *Communications of the ACM* 51.8 (2008), pp. 83–89.

[Boy96] John Tang Boyland. “Descriptional Composition of Compiler Components”. PhD thesis. University of California, Berkeley, 1996.

[CK94] Shyam R Chidamber and Chris F Kemerer. “A metrics suite for object oriented design”. In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[Cok18] David R Cok. “Reasoning about functional programming in Java and C++”. In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 2018, pp. 37–39.

[Die+17] Jens Dietrich et al. “XCorpus - An executable Corpus of Java Programs”. In: *Journal of Object Technology* 16.4 (2017), 1:1–24.

[DRS21] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. “JavaDL: Automatically Incrementalizing Java Bug Pattern Detection”. In: *Proceedings of the ACM on Programming Languages*. Virtual: ACM, 2021.

[EH07a] Torbjörn Ekman and Görel Hedin. “The jastadd extensible Java compiler”. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. 2007, pp. 1–18.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000).

[HM03] Görel Hedin and Eva Magnusson. “JastAdd—an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[Java] *JDK 1.1 New Feature Summary*. Note: Available as file jdk1.1.8/docs/relnotes/features.html in zip file for Java Development Kit (JDK) Documentation 1.1 (jdk118-doc.zip) at `https://www.oracle.com/java/technologies/java-archive-downloads-javase11-downloads.html` (login needed), Last accessed: 2023-02-17.

[Javb] *Java 2 SDK, Standard Edition, version 1.2. Summary of New Features and Enhancements*. Note: Available as file jdk1.2.2.202/docs/relnotes/features.html in zip file for Java 2 SDK, Standard Edition Documentation 1.2.2_006 (jdk-1_2_2_006-doc.zip) at `https://www.oracle.com/java/technologies/java-archive-javase-v12-downloads.html` (login needed), Last accessed: 2023-02-17.

[Javc] *Java 2 SDK, Standard Edition, version 1.3. Summary of New Features and Enhancements*. Note: Available as file docs/relnotes/features.html in zip file for Java 2 SDK, Standard Edition Documentation 1.3.1 (java1.3.zip) at `https://www.oracle.com/java/technologies/java-archive-13docs-downloads.html` (login needed), Last accessed: 2023-02-17.

[Javd] *Java 2 Sdk for Solaris Developer’s Guide*. ISBN: 978-14-005-2241-5, Note: Includes description of New Features and Enhancements for Java 1.4. Sun Microsystems, 2000.

[Jave] *Java SE 6 Features and Enhancements.* `https://www.oracle.com/java/technologies/javase/features.html`. Accessed: 2023-02-17.

[Javf] *Java SE 7 Features and Enhancements.* `https://www.oracle.com/java/technologies/javase/jdk7-relnotes.html`. Accessed: 2023-02-17.

[JJE14] René Just, Darioush Jalali, and Michael D Ernst. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* 2014, pp. 437–440.

[Knu68] Donald E Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[Li+14] Kaituo Li et al. "Residual Investigation: Predictive and Precise Bug Detection". In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (2014), 7:1–7:32.

[LMG11] Markus Lumpe, Samiran Mahmud, and Olga Goloshchapova. "jCT: A Java Code Tomograph". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).* 2011, pp. 616–619.

[LDB19a] Linghui Luo, Julian Dolby, and Eric Bodden. "MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper)". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019).* Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 21:1–21:25.

[MEH07b] Eva Magnusson, Torbjorn Ekman, and Gorel Hedin. "Extending Attribute Grammars with Collection Attributes–Evaluation and Applications". In: *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007).* 2007, pp. 69–80.

[MLN15] Leonel Merino, Mircea Lungu, and Oscar Nierstrasz. "Explora: A visualisation tool for metric analysis of software corpora". In: *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT).* IEEE. 2015, pp. 195–199.

[Javg] *New Features and Enhancements. J2SE 5.0.* `https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html`. Accessed: 2023-02-17.

[Rei+10] Christoph Reichenbach et al. “What can the GC compute efficiently?: A language for heap assertions at GC time”. In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, 2010, pp. 256–269.

[Rio+21] Idriss Riouak et al. “A Precise Framework for Source-Level Control-Flow Analysis”. In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2021, pp. 1–11.

[Spi05] Diomidis Spinellis. “Tool Writing: A Forgotten Art?” In: *IEEE Software* 22.4 (2005), pp. 9–11.

[Tem+10a] Ewan Tempero et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. In: *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. Dec. 2010, pp. 336–345.

[Javh] *What’s New in JDK 8.* `https://www.oracle.com/java/technologies/javase/8-whats-new.html`. Accessed: 2023-02-17.

Efficient Demand Evaluation of Fixed-Point Attributes Using Static Analysis

Abstract

Declarative approaches to program analysis promise a number of practical advantages over imperative approaches, from eliminating manual worklist management to increasing modularity. Reference Attribute Grammars (RAGs) are one such approach. One particular advantage of RAGs is the automatic generation of on-demand implementations, suitable for query-based interactive tooling as well as for client analyses that do not require full evaluation of underlying analyses. While historically aimed at compiler frontend construction, the addition of circular (fixed-point) attributes also makes them suitable for dataflow problems. However, prior algorithms for on-demand circular RAG evaluation can be inefficient or even impractical for dataflow analysis of realistic programming languages like Java. We propose a new demand algorithm for attribute evaluation that addresses these weaknesses, and apply it to a number of real-world case studies. Our algorithm exploits the fact that some attributes can never be circular, and we de-

Idriss Riouak, Niklas Fors, Jesper Öqvist, Görel Hedin and Christoph Reichenbach.
"Efficient Demand Evaluation of Fixed-Point Attributes using Static Analysis".
In *17th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, Pasadena, California, USA, 2024.
Paper DOI: 10.1145/3687997.3695644.

scribe a static meta-analysis that identifies such attributes, and obtains a median steady-state performance speedup of $\sim$2.5x and $\sim$22x for dead-assignment and null-pointer dereference analyses, respectively.

1 Introduction

Static program analysis is a key part of modern software tools, including compilers and static checkers. After first deriving facts from program code, many analyses rely on a fixed-point computation over some lattice to find a solution to a mutually dependent equation system. Typically, this computation is either *data-driven*, exhaustively computing all derivable facts, or *on demand*, computing only the facts necessary to answer a particular query. Demand evaluation can substantially outperform data-driven exhaustive analysis when the analysis client asks for only a subset of the analysis results, e.g., for a dead code elimination that uses constant folding only for branch conditions or for interactive tools that scan only code portions that are visible in the editor.

Reference Attribute Grammars (RAGs) [Hed00] are a high-level declarative formalism for specifying static program analyses in terms of *attributes*, i.e., properties associated with program nodes. These specifications take the form of equations (sometimes called semantic functions) that may introduce dependencies between attributes. RAGs extend Knuth Attribute Grammars (AGs) [Knu68] to allow attribute equations to describe and traverse *references* to other AST nodes, and contemporary RAG frameworks provide facilities to reify additional structures [VSK89a] such as Control Flow Graphs (CFGs), and to compute fixed points with the help of *circular attributes* [MH07c] to solve typical dataflow problems [Söd+13c; Rio+21].

Contemporary RAG compilers [SKV10; Van+10a; HM03] translate these equations into attribute evaluation engines that answer attribute queries by recursively evaluating the equations on demand, memoizing intermediate results. When an attribute has a self-dependency, evaluation iterates until the result no longer changes. This evaluation strategy is practical for many applications; for example, the ExtendJ Java compiler, which is specified using RAGs, executes within $3\times$ the execution time of the handwritten reference compiler `javac` [EH07b].

However, for applications that make heavy use of fixed point computations, efficient evaluation may hinge on identifying strongly-connected components (SCCs) over the dependency graph and evaluating them in topological order [HDT87]. This is non-trivial for RAGs as the dependency graph depends on reference attribute values, and is therefore not known before evaluation. Fixed point support in contemporary RAG compilers uses either a heavy-weight algorithm for all potentially cyclic attributes that can distinguish SCCs separated by non-circular attributes [MH07c], or a light-weight algorithm that operates on a subset of the potentially cyclic attributes but cannot distinguish between different SCCs [ÖH17].

We here propose a novel evaluation algorithm that overcomes the limitations of the earlier algorithms by combining their insights with a technique that statically identifies attributes that are guaranteed to never be on a cycle. We have implemented and validated our algorithm in the JastAdd metaprogramming system [HM03], which compiles RAG specifications to Java code. Our approach constructs a call graph from the generated Java code and maps it back into attribute declaration dependencies, which we use to conservatively overapproximate dynamic evaluation dependency cycles. We then feed this information back into JastAdd to allow it to generate more efficient evaluation code.

While our work builds on RAGs, the algorithms are general in that they can be applied to implement any system that exposes a demand analysis as an observationally pure query API on a graph of nodes (e.g., an abstract syntax tree or an abstract syntax graph). We believe that our algorithms could therefore also be useful for compilers built around other query-based architectures, including Microsoft's Roslyn platform and the `rustc` compiler for Rust.

We start by giving a brief background on RAGs and circular attributes (Section 2). We then present our contributions:

- We introduce our new attribute evaluation algorithm (Section 3).
- We propose a novel approach to conservatively identify dependencies in RAGs based on the call graph of the generated evaluation code and explain how our new algorithm uses this information to speed up evaluation (Section 4).
- We evaluate our approach on a set of real-world case studies. Our evaluation shows that our approach can significantly improve the performance of the generated evaluator (Section 5).

We then discuss related work (Section 6) before concluding the paper (Section 7).

2 Reference Attribute Grammars with Circular Attributes

In this section we introduce Reference Attribute Grammars (RAGs) and circular attributes by defining reachability for a simple state machine language. Consider the example state machine in Figure 1. For each of the machine's three states, we want to compute the set of transitively reachable states.

We demonstrate the analysis in the metacompilation system JastAdd, eliding the concrete syntax definition for brevity. We first parse input state machine

```
state S1;
state S2;
state S3;
trans S1 -> S2;
trans S2 -> S1;
trans S2 -> S3;
```

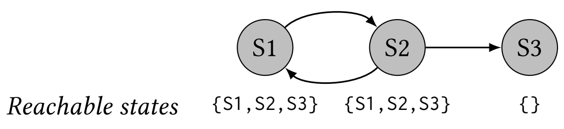


Figure 1: State machine example. Textual syntax (left), visual depiction annotated with reachable states (right).

```
/* - Abstract Syntax Definitions - */
Machine ::= State* Transition*;
State ::= <Label:String>;
Transition ::= <SourceLabel:String> <TargetLabel:String>;

/* - RAG Attribute Definitions - */
syn Set<State> State.successors();
eq State.successors() { ... };

syn Set<State> State.reachable()
circular [new HashSet<State>()];
eq State.reachable() {
       Set<State> result = new HashSet<State>();
       for (State s: successors()) {
              result.add(s);
              result.addAll(s.reachable());
       }
       return result;
}
```

Figure 2: State machine language definition, comprising the abstract grammar and the RAG specification of reachable.

programs such as the one in Figure 1 into an abstract syntax tree (AST). Figure 2 (top) shows the abstract grammar: a state machine (`Machine`) consists of a list of states (`State`) and a list of transitions (`Transition`). Each state has a label, and each transition has a source and a target label.

Figure 2 (bottom) shows the definitions of two attributes, `successors` and `reachable`. Each definition specifies the attribute name on its left-hand side and gives a Java method body on the right-hand side that must have no observable side effects. For example, the attribute `successors` defines the set of all successor states for `State`, though we elide the implementation for brevity. Each non-terminal instance (AST node) has its own set of attribute instances. For a state n, our definition of `reachable` computes the following:

$$n.reachable = \bigcup_{s \in n.successors} \{s\} \cup s.reachable$$

Figure 2 writes the right-hand side of this equation in plain Java code, looping over all successors to construct their union. For each attribute, JastAdd generates a namesake method, which here allows `reachable` to access `successors` directly.

The keyword `syn` marks both attributes as so-called *synthesized* attributes, meaning that we evaluate their defining equations in the context of the node that the attribute belongs to. Other kinds of attributes use other contexts; e.g., *inherited* attributes use the parent node context, though this distinction is inessential for the work that we present here.

Since state machines may contain cycles, an instance of `reachable` may depend on itself. We must thus declare `reachable` as `circular`, which tells JastAdd to evaluate it with a fixed-point iteration algorithm (shown in Section 3). Circular attributes must have explicit *bottom values*: here, we use the empty set (`[new HashSet<State>()]`). Since the set of `States` forms a finite lattice on which set union is monotonic, iteration terminates. JastAdd requires attribute definitions to ensure that there can be no infinite chains of updates, e.g. via finite-height lattices and monotonic updates.

When we access an attribute from Java, JastAdd will now compute it on demand and memoize the result. For example, suppose that we have parsed the program from Figure 1 into an AST and have a reference `s1` to the `S1` state node. By calling `s1.reachable()` we will execute the right-hand side of the `reachable` attribute equation, which in turn will call `s1.successors()`, and then recurse by calling `s.reachable()` for each `s` from `s1.successors()`. If we call `reachable()` on the S3 state node, its `successors` attribute will be the empty set, so `reachable` will return the empty set without recursing. For any attribute evaluation, the exact set of attribute instances that we need to evaluate thus depends on the structure of the AST, the equations, and the values of attributes that we have evaluated so far.

3 Circular Attribute Algorithms

We now describe our on-demand algorithms for attribute evaluation in the presence of circular attributes. We first discuss a general framework that captures commonalities across the algorithms, then detail the constituent subalgorithms.

3.1 Preliminaries

In our approach, each non-terminal X is implemented by a corresponding class `X`, and each attribute *X.attr* by a method `X.attr()`. If `x` is an instance of class `X`, we can thus access the value of an attribute instance $x.attr$ by calling `x.attr()`.

We consider only well-formed RAGs for which each attribute instance will have exactly one defining equation for any possible AST. The defining equation and the attribute may be located in different AST nodes, e.g., if the attribute is inherited rather than synthesized. We abstract away the equation location by introducing a method `X.attr_compute()` for each attribute declaration `X.attr()`. This method will locate the equation in the AST and call a method corresponding to the right-hand side of the equation. In this process, the method will call a number of other attribute instances.

The calls form a dynamic dependency graph where each edge $\langle a,b \rangle$ represents a call from attribute instance a to attribute instance b. When a can transitively reach an attribute instance c along the edges of this dependency graph, we say that c is *downstream* from a, and when a is downstream from a itself, we say that a is *effectively circular*. Since the dynamic dependency graph can depend on dynamically computed reference attributes, we cannot precisely predict whether a given attribute instance is circular in the general case.

3.2 Attribute Declarations and Main Algorithms

To avoid unnecessary fixed-point computation, we require a declaration for each attribute that selects one of three sub-algorithms for evaluating that attribute's instances:

Circular An instance of an attribute declared as Circular is allowed to be effectively circular. A Circular attribute instance, $x.attr$, will be evaluated by a fixed-point computation, and has an explicit bottom value, computed by the method `x.attr_bottom_value()`.

NonCircular An instance of an attribute declared as NonCircular is not allowed to be effectively circular. If it is, evaluating the instance will yield a runtime error.

Agnostic An instance of an attribute declared as Agnostic is allowed to be effectively circular, if there is at least one attribute declared as Circular on each cycle it is part of. An Agnostic attribute does not have any explicit bottom value. Instead, its first approximation will be computed based on the approximations of its downstream attributes. If it is on a cycle without any intervening Circular attribute, attempting to evaluate it will yield a runtime error.

We say that a Circular attribute instance and its downstream attributes, up to any NonCircular attribute, belong to the same *fixed-point component*. Thus, NonCircular attributes stratify different fixed-point components into an acyclic component graph. If evaluation starts in one fixed-point component and flows through a NonCircular attribute into another component, we suspend fixed-point computation for the first component until we have reached a fixed point for the second component (Section 3.5). When two strongly connected components are instead directly connected or separated only by Agnostic attributes, we evaluate them as one single fixed-point component.

We consider three different main algorithms for evaluation: BasicStacked, RelaxedMonolithic, and RelaxedStacked. BasicStacked corresponds to the original algorithm by Magnusson [MH07c] and supports Circular and NonCircular attributes. Our version of BasicStacked is somewhat different from Magnusson's version: We keep track of in which fixed-point iteration each attribute was most recently evaluated. This allows for an important optimization where we avoid evaluating an attribute more than once if its value is used more than once in the same iteration. This also allows us to detect if an attribute that is (erroneously) classified as NonCircular is actually on a cycle at runtime. In the paper by Magnusson, the algorithm did not support such detection, but could instead compute the wrong result in case of a specification error like this. The paper only sketched a fix to this problem, and which relied on keeping track of sets of attribute instances for each fixed-point component, which would have slowed down the evaluation substantially.

A consequence of BasicStacked is that all attributes that may have an effectively circular instance, for some AST, must be declared as Circular. This can be impractical for larger systems, like compilers and program analyzers for real languages. For example, it may be the case that a common attribute, say a type attribute, can have instances that are on cycles only for particular language constructs, e.g., local type inference in lambda expressions. Requiring an attribute to be declared as Circular would then give an efficiency penalty when analyzing all other parts of the program where instances of the attribute are actually not on a cycle. To avoid this problem, Öqvist introduced an alternative algorithm that we call RelaxedMonolithic [ÖH17; Öqv18], which supports Circular and Agnostic attributes. Agnostic attributes can be on a cycle, but if they are not, their evaluation can be more efficient than for Circular attributes.

When using RelaxedMonolithic, all attributes that are not explicitly declared as Circular are assumed to be Agnostic, so there are no NonCircular attributes that can separate strongly connected components. Therefore, when a Circular attribute is evaluated, all its downstream attributes, both Circular and Agnostic, will be evaluated as part of the same monolithic fixed-point component. As our evaluation will show, this can be very inefficient for demand analyses that start with querying a Circular attribute. To get the best of both BasicStacked and RelaxedMonolithic, we therefore propose a new algorithm, RelaxedStacked that supports all three kinds of attributes. In this algorithm, NonCircular attributes can be used to separate the evaluation into smaller fixed-point components. By using a static conservative analysis of the attribute specification, we can identify attributes that are guaranteed to never be on any cycle in any possible AST, and that can therefore safely be classified as NonCircular.

In the RAG specification, Circular attributes are the only attributes requiring an explicit annotation by the user, i.e., `circular`, like the `reachable` attribute in Figure 2. Attributes that are not declared as `circular`, such as `successors`, are referred to as *normal* attributes. Normal attributes are classified as either NonCircular or Agnostic, depending on the main algorithm used, and on results from the static analysis of the specification in the case of the RelaxedStacked algorithm.

3.3 Subalgorithms and Variables

There is one subalgorithm for each of the three attribute kinds: Circular, NonCircular, and Agnostic. We have formulated the subalgorithms so that all three main algorithms can use different combinations of exactly the same subalgorithms. Hence, BasicStacked corresponds to using the two subalgorithms Circular and NonCircular; RelaxedMonolithic corresponds to using Circular and Agnostic; RelaxedStacked corresponds to using all three subalgorithms. The subalgorithms are shown in Listings 1-3 and use the following key global variables.

`IN_CIRCLE` is a boolean global variable that is `true` when evaluation is ongoing inside a fixed-point component, and `false` otherwise. If an attribute is called when `IN_CIRCLE` is `false`, its final value is returned. If it is called when `IN_CIRCLE` is `true`, an approximation of it is returned.

`CHANGE` is a boolean global variable indicating if any value was changed in the current fixed-point iteration.

`CIRCLE_ITER` is a global object uniquely identifying the current fixed-point iteration.

Once an algorithm has computed the final value of some attribute $X.attr$, the algorithm memoizes the result in an instance variable `X.attr_value`. To keep track of whether the attribute is memoized or not, NonCircular attributes use a boolean instance variable `X.attr_computed`. For Circular and Agnostic attributes, the `X.attr_value` instance variable will hold the current approximation of the value. To monitor the status of `X.attr_value`, we use an instance variable X.attr_iter of type Object. Initially, this is set to NOT_INITIALIZED to indicate that no approximation has yet been computed. Then, in each fixed-point iteration when a new approximation is computed, the X.attr_iter is set to the object identifying that iteration. Finally, when it is deduced that the current approximation is the final value of the attribute, this is recorded by setting X.attr_iter to the constant object FINAL_VALUE.

Evaluation starts when the main program calls an attribute of some node of the AST. Since IN_CIRCLE is initially false, this call will return the final memoized value of the attribute. As an effect of this evaluation, other attributes may either be still unevaluated, or have an approximate value, or have their final memoized value. A later call to an attribute with an approximate value will continue its evaluation.

```
// Global variables
boolean IN_CIRCLE = false;
boolean CHANGE = false;
Object CIRCLE_ITER = new Object();

// Global constants
final Object FINAL_VALUE = new Object();
final Object NOT_INITIALIZED = new Object();

class X {
  // Instance variables
  Object attr_iter = NOT_INITIALIZED;
  T attr_value;

  T attr() {
    if (attr_iter == FINAL_VALUE) {
      return attr_value;
    }

    if (attr_iter == NOT_INITIALIZED) {
      attr_value = attr_bottom_value();
    }

    if (!IN_CIRCLE) {
      // DRIVES
      IN_CIRCLE = true;
      do {
        CHANGE = false;
        CIRCLE_ITER = new Object();
        attr_iter = CIRCLE_ITER;
        T v = attr_compute();
        if (!Objects.equals(attr_value, v)) {
          CHANGE = true;
        }
        attr_value = v;
      } while (CHANGE);
      attr_iter = FINAL_VALUE;
      IN_CIRCLE = false;
      return attr_value;
    } else if (attr_iter != CIRCLE_ITER) {
      // FOLLOWS
      attr_iter = CIRCLE_ITER;
      T v = attr_compute();
      if (!Objects.equals(attr_value, v)) {
        CHANGE = true;
      }
      attr_value = v;
      return attr_value;
    } else {
      // ALREADY HANDLED in this iteration
      return attr_value;
    }
  }
}
```

Listing 1: Evaluation of Circular attributes

3.4 The Circular Subalgorithm

Listing 1 shows the subalgorithm for Circular attributes. The first Circular attribute that is called in a fixed-point component will take the role of *driver*, running the loop of fixed-point iterations. Other Circular attributes in the same component will be *followers*. The algorithm distinguishes three different situations when calling a Circular attribute:

`DRIVES` An attribute instance takes the role of driver and starts a fixed-point computation. It runs a loop and in each iteration, it calls its `compute` method to get a new approximation of its value, potentially (transitively) calling other Circular attributes in the component, as well as itself.

`FOLLOWS` An attribute instance other than the driver is called for the first time during the current fixed-point iteration. It computes a new approximation by calling its `compute` method, again potentially (transitively) calling other Circular attributes in the component, as well as itself.

```java
// Global variables
Stack STACK = ...;

class X {
  // Instance variables
  boolean attr_computed = false;
  T attr_value;

  T attr() {
    if (attr_computed) {
      return attr_value;
    }
    if (!IN_CIRCLE) {
      // NORMAL
      attr_value = attr_compute();
      attr_computed = true;
      return attr_value;
    } else {
      // BRIDGE
      push CHANGE, CIRCLE_ITER on STACK;
      IN_CIRCLE = false;
      attr_value = attr_compute();
      IN_CIRCLE = true;
      CHANGE, CIRCLE_ITER = pop from STACK;
      attr_computed = true;
      return attr_value;
    }
  }
}
```

Listing 2: Evaluation of NonCircular attributes

```
class X {
  // Instance variables
  Object attr_iter = NOT_INITIALIZED;
  T attr_value;

  T attr() {
    if (attr_iter == FINAL_VALUE) {
      return attr_value;
    }
    if (!IN_CIRCLE) {
      // NORMAL
      attr_value = attr_compute();
      attr_iter = FINAL_VALUE;
      return attr_value;
    } else {
      if (attr_iter != CIRCLE_ITER) {
        // FOLLOWS
        attr_value = attr_compute();
        attr_iter = CIRCLE_ITER;
        return attr_value;
      } else {
        // ALREADY COMPUTED in this iteration
        return attr_value;
      }
    }
  }
}
```

Listing 3: Evaluation of Agnostic attributes

$$
\begin{aligned}
a &= \{5\} \cup b \\
b &= \{42\} \cup c \cup d \\
c &= b \cup a \\
d &= c
\end{aligned}
$$

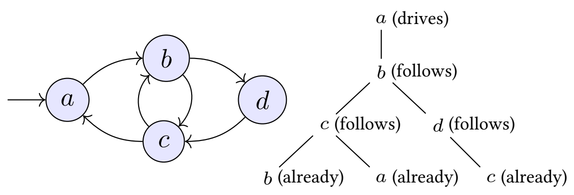


Figure 3: Equation system for Circular attributes (left). Dynamic dependency graph (middle). Call tree for one iteration (right).

ALREADY HANDLED A driver or a follower is called during fixed-point iteration, but has either already been computed in that iteration or is in the process of being computed (i.e., another method invocation for the same attribute is on the call stack). Then it simply returns its current value.

To illustrate how the Circular evaluation works, consider the example in Figure 3 showing an equation system, the dynamic dependency graph, and the tree of method calls for one of the fixed-point iterations, given that a client demands the attribute a by calling `a()`. The attributes a, b, c, and d are all sets of integers, and we assume that they are all declared as Circular. Solving the equation system with a fixed-point iteration, starting out with the empty set as bottom, the solution will be $a = b = c = d = \{5, 42\}$.

Because the evaluation starts with a, this attribute becomes the driver, and will execute the **DRIVES** part of the code, with the fixed-point loop. In each iteration in the loop, it calls its `compute()` method which will in turn call `b()`. The attribute b becomes a follower, and will execute the **FOLLOWS** part of the code which calls its `compute` method that will first call `c()` and then `d()`. Both these attributes also become followers. The attribute c will similarly call `b()` and `a()`, but both these will execute the **ALREADY HANDLED** part of the code, since they are in the process of already being evaluated during the same iteration, and their current approximation is returned directly, without any call to `compute()`, ending the recursion. Similarly, when `d()` calls `c()`, then c has already computed a new approximate value in the same iteration, due to the previous call from `b()` to `c()`, again ending the recursion. We can see from this example that the recursion will terminate, and that each attribute that a depends on will update its value exactly once during an iteration.

Both the driver and the followers update the global variable `CHANGE` to keep track of whether any of the approximations were updated during the current iteration. The driver will loop until there is an iteration where no approximations are updated. The driver and all its followers will then have their final values and can memoize them. For simplicity, the algorithm in Listing 1 only memoizes the driver, i.e., a in the example in Figure 3. Using an optimization called *LastCycle* [MH07c], the driver can memoize all the followers as well (b, c, and d in the

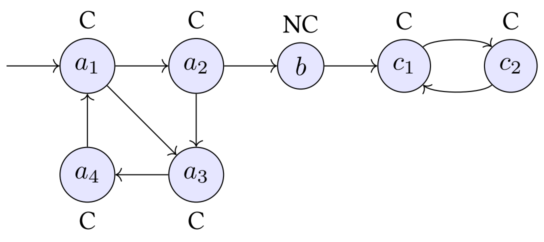


Figure 4: Attribute serving as a bridge between two circular components. C=Circular, NC=NonCircular.

example). This is accomplished by the driver calling its `compute()` method an extra time, with an extra global flag set to signal to followers that they should memoize their values. (The code for this is elided for brevity.) If this optimization is not used, and a previous follower is called at a later point in time, it will become a driver, and find after one iteration that it can be memoized.

3.5 The NonCircular Subalgorithm

NonCircular attributes use the subalgorithm in Listing 2. An instance of a NonCircular attribute is assumed to not be effectively circular. (If it actually is circular, a runtime error will be raised, see Appendix A.) The algorithm distinguishes between two different situations when the NonCircular attribute is called:

`NORMAL` In the normal case, the attribute is called when there is no ongoing fixed-point computation (i.e., `IN_CIRCLE == false`). It can then simply call its `compute()` method and memoize the result. This is so since when `IN_CIRCLE == false`, any attribute called by the `compute()` method will return its final value.

`BRIDGE` If the attribute is called during an ongoing fixed-point computation (i.e., `IN_CIRCLE == true`), any downstream Circular attribute will, by definition, belong to a separate fixed-point component. We say that the NonCircular attribute serves as a *bridge* between the upstream and any downstream components. A downstream component should run its own fixed-point computation for efficiency. This is accomplished by the NonCircular attribute stacking the state of the ongoing component (i.e., the variables `CHANGE` and `CIRCLE_ITER`), and setting `IN_CIRCLE` to `false` before calling its `compute()` method. If a Circular attribute is encountered during the `compute()` call, it will start its own fixed-point computation, and finish this computation before returning its value. After the `compute()` call, the stacked variables are restored, and `IN_CIRCLE` is set to `true` again.

Figure 4 shows an example. Here, the NonCircular attribute b serves as a bridge between the components $\{a_1, a_2, a_3, a_4\}$ and $\{c_1, c_2\}$. Suppose the evaluation starts by a call to a_1, which will become the driver of the $\{a_1, a_2, a_3, a_4\}$ component. When a_2 calls b in the first iteration, the component will be stacked. Then b calls c_1 which becomes the driver of a new fixed-point loop for $\{c_1, c_2\}$. This component will loop until c_1 is finalized and memoized, and then return control to b. Then b will compute and memoize its own value, pop the $\{a_1, a_2, a_3, a_4\}$ component, and return control to a_2. In the second iteration of $\{a_1, a_2, a_3, a_4\}$, calling b will directly return b's memoized value.

If the NonCircular b attribute did not stack the component state and set `IN_CIRCLE` to `false`, its call to `compute()` might yield only an approximation rather than a final value, so it would not be safe to memoize b's value here. Essentially, this would lead to the evaluation of all the attributes $\{a_1, a_2, a_3, a_4, b, c_1, c_2\}$ in a big monolithic fixed-point loop, driven by a_1. The NonCircular algorithm in Listing 2 thus has two advantages over a non-stacking variant of the algorithm: it will separate circular components, and it will avoid evaluating the NonCircular attribute more than once.

3.6 The Agnostic Subalgorithm

Agnostic attributes use the subalgorithm in Listing 3. They may be part of a cycle, but only as followers, never as drivers. We can thus assume that any cycle that contains an Agnostic attribute instance also contains a Circular attribute instance to act as the driver. (If an Agnostic attribute is on a cycle without any Circular attribute, the evaluation algorithm raises an error, cf. Appendix A.)

In the following, we explicitly note several differences to the previous algorithms that we discuss further in Sections 3.6, 3.6, and 3.6.

Note 1: Agnostic attributes have no bottom values. If they are in a cycle, we compute their first approximations from the bottom values of the Circular attributes on the cycle.

The algorithm distinguishes between three situations when an Agnostic attribute is called:

`NORMAL` This case is similar to the corresponding case for NonCircular attributes, i.e., there is no ongoing fixed-point computation (`IN_CIRCLE == false`). The attribute will call its `compute()` method and memoize its result. Note 2: Since the Agnostic attribute may be on a cycle, it may be revisited by another downstream call.

`FOLLOWS` This case is similar to the corresponding case for Circular attributes. It occurs when there is an ongoing cycle (`IN_CIRCLE == true`), and its recorded iteration is different from the current one (`attr_iter != CIRCLE_ITER`). In this case, the attribute's `compute()` method is called to compute a new approximation.

Note 3: Unlike a CIRCULAR attribute, an AGNOSTIC attribute does not compare its current value against the previous one and never updates the CHANGE flag.
Note 4: Unlike a CIRCULAR attribute, an AGNOSTIC attribute updates its `attr_iter` *after* the call to `compute()`.

`ALREADY HANDLED` This case is similar to the corresponding one for CIRCULAR attributes. Here, there is an ongoing cycle (`IN_CIRCLE == true`), and the recorded `iter` is the same as the current one (`attr_iter == CIRCLE_ITER`). In this case, the attribute is already computed in the current iteration, and it simply returns its current approximation without computing a new one, thus ending the recursion.

The four properties that we noted above affect the performance and correctness of the algorithm, as we detail below.

An AGNOSTIC Attribute Has No Explicit Bottom Value

Since AGNOSTIC attributes have no explicit bottom value (see Note 1), they must not end up in the **`ALREADY HANDLED`** case without first having computed an approximate value. We can see that this cannot happen, because `attr_iter == CIRCLE_ITER` can happen only if the evaluation previously has passed through all of the **`FOLLOWS`** code, where `attr_iter` is set to `CIRCLE_ITER`, meaning that the value has been set. For this reason, it is important that `attr_iter` is set to `CIRCLE_ITER` *after* the call to `compute()`, and not before (see Note 4).

An AGNOSTIC Attribute Does Not Set the CHANGE Flag

A new approximate value of an AGNOSTIC attribute cannot depend on itself unless this dependency goes via one or more CIRCULAR attributes. The AGNOSTIC attribute can only get a new approximation if at least one of these CIRCULAR attributes has a new value. But if it has, it will have set the CHANGE flag. Therefore, the AGNOSTIC attribute does not need to set the CHANGE flag (see Note 3). This also means that if there is an iteration where the CHANGE flag was not set, i.e., the fixed-point computation has completed, also the AGNOSTIC attribute will have its final value.

An AGNOSTIC Attribute Executing the NORMAL Case May Be Revisited Downstream

If an AGNOSTIC attribute is on a cycle, but called from outside of circular evaluation, a CIRCULAR attribute on the cycle will drive the fixed-point evaluation. The AGNOSTIC attribute will then start by executing the **`NORMAL`** case, but as a part of this computation, be recursively called (see Note 2). When the fixed-point evaluation has started, the AGNOSTIC attribute will enter the **`FOLLOW`** code

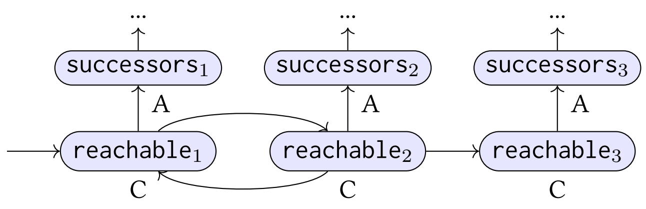


Figure 5: Dynamic dependency graph for the state machine example in Figure 1. C=Circular, A=Agnostic.

in each iteration, and compute a new approximation. When the fixed-point evaluation terminates, the Agnostic attribute will also have been iterated to its final value, as explained in the previous Section 3.6. When the evaluation returns to the Agnostic call executing the **NORMAL** case, it is therefore safe to memoize the attribute.

4 Static Analysis to Identify NonCircular Attributes

For an attribute that is not declared as Circular, we have the option of either declaring it as Agnostic or as NonCircular. Using an Agnostic attribute has the advantage that it is safe to use, even if it is on a cycle (as long as there is some Circular attribute on the cycle).

However, an Agnostic attribute can be quite inefficient if it is called from an upstream cycle, but is not itself on a cycle. In this case, the upstream cycle will have a Circular attribute that drives a fixed-point loop, and the Agnostic attribute will be evaluated once for each iteration of this loop. Since the Agnostic attribute is not on any cycle in this particular case, each of these evaluations will result in the same value, resulting in unnecessary work.

As an example, this situation occurs for our state machine example from Figure 1. Suppose the `successors` attribute, as well as all its downstream attributes that compute the name analysis, are declared as Agnostic. In this case, we get the dependency graph shown in Figure 5. If the evaluation starts in `reachable`$_1$, each of the `successors` attributes, as well as all their downstream attributes, will be re-evaluated once per iteration during each of the n fixed-point iterations of `reachable`$_1$. This can potentially be very inefficient, leading to all downstream attributes being recomputed n times.

Another kind of inefficiency is due to different fixed-point components. If the two components in Figure 4 were separated by an Agnostic rather than by a `NonCircular` attribute, and evaluation starts in the upstream component, then the evaluation could not be done separately for the two components. Instead, all

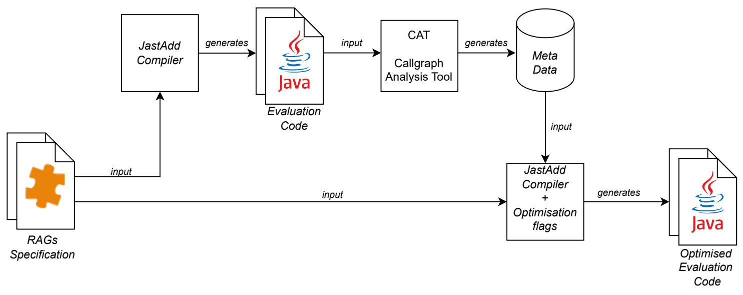


Figure 6: Overview over our approach. The JastAdd metacompiler generates Java code from a RAG specification. CAT determines which attributes may be NonCircular and feeds this information into a second run of JastAdd that can then generate more efficient evaluation code.

the attributes would be evaluated in a big monolithic component, which is less efficient.

Both these inefficiencies would be avoided if we used `NonCircular` attributes instead of Agnostic ones. However, we only want to use `NonCircular` attributes if we are sure that they will never be on any cycle, for any possible AST.

To solve this problem, we have implemented a tool CAT (`https://github.com/idrissrio/cat`), that we use to analyze the static call graph of a RAG. We use this analysis to identify attributes that can safely be declared as `NonCircular`.

4.1 Approach Overview

CAT is a general call graph analysis tool for Java, and we use it to analyze the Java code that is generated from a JastAdd RAG specification. An overview of our approach is shown in Figure 6. Initially, the RAG specification is fed into the JastAdd metacompiler, which generates the corresponding evaluation code in Java, using the subalgorithms described in Section 3. We use the Agnostic code as the default for attributes without annotations. Then, CAT analyses the generated evaluation code and computes the corresponding call graph. In this call graph, method declarations are nodes, and edges represent method calls. The CAT tool uses this call graph to identify what attributes can safely be declared as `NonCircular`, and outputs this information as a meta data file. Then JastAdd is run again, this time with the meta data as additional input and with some optimization flags enabled. JastAdd uses this extra information to generate optimized evaluation code where as many as possible of the unannotated attributes use the NonCircular subalgorithm instead of the Agnostic one.

4.2 Call Graph Construction

A call graph is a directed graph that represents the calling relationships between methods in a program. We say that a call graph is *sound* if it contains all the possible method calls that can occur at runtime. One of the main challenges in constructing a sound call graph is effectively managing *dynamic dispatch*, which is the capability to dynamically choose the method to call based on the runtime type of the receiver object. CAT handles dynamic dispatch by using a technique called Class Hierarchy Analysis (CHA) [DGC95]. CHA is a *context- and flow-insensitive* analysis, meaning that it does not consider the context of the method calls and it does not consider the order of the statements in the program. A key aspect of CHA is that given a method call on a receiver object of a certain type, it considers all the possible subclasses of that type, and includes all the methods in these subclasses in the call graph. This way, CHA ensures that all possible method calls are included in the call graph, even if the exact type of the receiver object is not known at compile time.

4.3 Identifying Non-Circular Attributes

Since we are interested in how attributes call each other, we start by constructing a filtered call graph that only includes methods that correspond to attributes. We first obtain this set of methods from annotations generated by the JastAdd metacompiler, and then project all paths from the original call graph onto the filtered one.

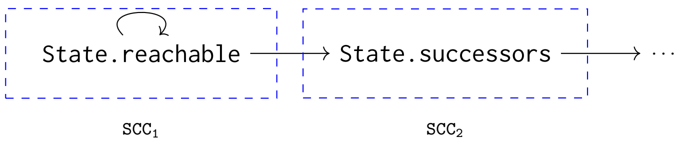
To identify non-circular attributes, we employ *Tarjan's algorithm* [Tar72] on the filtered call graph to discover all strongly connected components (SCCs). A SCC constitutes a set of nodes in a directed graph where each node is reachable from every other node in the set.

Given the SCCs, we can safely mark an attribute n as NonCircular if both of the following conditions hold:

1. n is in an SCC with only a single node, and

2. n does not directly call itself (no self-loop).

These attributes can never be circular for any AST. Attributes not marked as NonCircular by CAT, and not explicitly marked as Circular, are by default considered Agnostic.



```
Set<State> reachable_compute(){
  Set<State> result = new HashSet<State>();
  for (State s : successors()) {
      result.add(s);
      result.addAll(s.reachable());
  }
  return result;
}
```

Figure 7: Call graph between attributes for the state machine language (left), and corresponding `compute` method (right). Dashed rectangles represent strongly connected components (SCC). Green methods are in the filtered call graph. Red methods are other methods in the original call graph.

To illustrate our approach, we revisit the state machine example from Section 2. Figure 7 shows a part of the call graph, and the `compute()` method for the `reachable` attribute that was used to generate it. The SCC analysis of the call graph identifies two distinct SCCs: $\mathtt{SCC}_1$ and $\mathtt{SCC}_2$. We see that the attribute `State.successors` can be declared as NonCircular, as both conditions 1 and 2 are met. Conversely, we see that the attribute `State.reachable` cannot be declared as NonCircular, since there is a self-loop in the graph, violating condition 2. This is expected since `State.reachable` is declared as Circular in the RAG, and instances of it are indeed on a cycle in the example in Figure 5.

4.4 Imprecision and Limitations

CAT is unsound on Java code that uses reflection, native calls, or dynamic class loading. Since CAT only analyzes code that JastAdd generates from RAG specifications, and since existing RAG specifications in JastAdd have not made use of these Java features, we currently expect that the practical significance of this limitation is minimal.

An important imprecision arises from attribute instances that recursively call other instances of the same attribute along the AST structure, but without being cyclic. An example would be when all calls between instances of the same attribute go downwards to children. The static approximation of the call graph will then be cyclic, whereas any dynamic instance of this part of the graph will be acyclic.

Since our approach can be adapted to any type of call graph, we expect that more precise call graphs can mitigate this imprecision, help identify more attributes as NonCircular, and thus further improve performance.

Another limitation of our approach is that the algorithm does not distinguish between different dynamic instances of the same static SCC. Generalizing the algorithm to detect dynamic SCCs at evaluation time, and investigating if this pays off in practice, is an interesting line of future research.

5 Evaluation

In this section, we present the evaluation performance of the three algorithms: BasicStacked, RelaxedMonolithic, and RelaxedStacked. We evaluated the RelaxedStacked algorithm across three distinct case studies: the construction of an LL(1) parser, the ExtendJ Java compiler, and IntraJ, an extension of the ExtendJ frontend for data-flow analysis. This section presents the findings for the LL(1) parser construction and IntraJ case studies, as the results for the ExtendJ case study, detailed in Appendix B, align with expectations and do not offer additional insights. For the IntraJ case study, we evaluate both a forward and a backward analysis. We exclude the BasicStacked algorithm from the second and third case studies, as it requires all attributes on a cycle to be declared Circular, which is impractical for complex applications like ExtendJ and IntraJ. When RelaxedStacked is evaluated, we use the CAT tool to automatically infer what attributes can be declared as NonCircular.

The first case study is included to demonstrate that the RelaxedStacked algorithm does not introduce any performance degradation for this application. On the other hand, with the IntraJ case study we demonstrate the advantages of the RelaxedStacked algorithm for more complex applications and for analyses in on-demand settings.

In all of these case studies, the specification includes a cache configuration, i.e., a specification of which attributes to memoize and which to reevaluate on each access. We used the cache configuration supplied by each respective tool, as the optimization of this aspect is a separate research challenge [ALL96; SH11].

5.1 Evaluation Setup

System Configuration Our experiments were conducted on a machine with an Intel Core i7-11700K CPU running at 3.60GHz and equipped with 128 GB RAM. The machine ran Ubuntu 22.04.3 and the benchmarks were executed using OpenJDK Runtime Environment Zulu 8.50.0.53-CA-linux64, build 1.8.0_275-b01. Additionally, for all evaluations, we fixed the Java Virtual Machine (JVM) heap size to 8 GB.

Benchmark Name	LOC	#Methods	Version
`antlr`	36525	2070	2.7.2
`pmd`	60749	5325	4.2.5
`struts`	81394	7023	2.3.22
`fop`	102746	8318	0.95
`extendj`	147265	16025	11.0
`castor`	235745	12643	1.3.3
`weka`	245719	14952	revision 7806
`poi`	329366	23816	3.11

Table 1: Evaluated Java benchmarks, including number of lines of code, number of methods, and version.

Evaluation Methodology The measurements were conducted separately for start-up performance on a cold JVM, involving a JVM restart for each run, and for steady-state performance, with a single measurement taken after 49 warmup runs. Each benchmark iteration was repeated 25 times, resulting in a total of 1250 runs for steady-state measurements. For steady-state measurements, we introduced a 300-second timeout since RelaxedMonolithic took a long time to run for some benchmarks. If any of the 49 warmup runs exceeded 300 seconds, we terminated the evaluation process for that particular steady-state measurement, disregarding the remaining warmup runs. The reported metrics include the median values and 95% confidence intervals. We checked the correctness of all three case studies by comparing their results to those from the original tools.

Benchmarks Table 1 shows the Java benchmark projects for the IntraJ and ExtendJ case studies. They include projects from the DaCapo [Bla+06] and Qualitas [Tem+10b] suites, e.g., `antlr` and `jfreechart`, and projects that we selected to cover a wide range of applications, including the generated Java source code of ExtendJ itself.

The artifact for running all the experiments is available online [Rio+24].

5.2 Case Study: LL(1) Parser Construction

LL(1) parsers can be generated by computing the *nullable*, *first*, and *follow* sets for a context-free grammar [App04]. Normally, these sets are computed by handwritten fixed-point algorithms. Magnusson et. al. [MH07c] instead formulated the computation as circular attributes. We use the RAG specification from their artifact [MH07a] to evaluate our different algorithms. For comparison, we also ran the original implementation from that artifact (BasicStacked$_{\text{old}}$).

BasicStacked$_{\text{old}}$	BasicStacked	RelaxedMonolithic	RelaxedStacked
4.24± 0.12	2.92± 0.05	8.30± 0.12	2.93± 0.03

Table 2: Startup performance results for the Java 1.2 grammar benchmark. Measurements in milliseconds.

Table 2 shows the startup performance results for computing *nullable*, *first*, and *follow* sets for a Java 1.2 grammar with 155 terminals and 332 productions. We can see that BasicStacked shows a performance improvement of $\frac{4.24}{2.92} = \sim 1.45$x over BasicStacked$_{\text{old}}$, confirming the efficacy of the improvements that we introduced for BasicStacked in Section 3.2. Furthermore, RelaxedStacked performs as well as BasicStacked, and is significantly faster than RelaxedMonolithic, with a speedup of $\frac{8.30}{2.93} = \sim 2.8$x. One reason for this is that RelaxedStacked is able to compute *follow* in a separate fixed-point component than *first* and *nullable*.

5.3 Case Study: IntraJ

IntraJ [Rio+21] is a dataflow analyser for Java built as an extension of the ExtendJ Java compiler. It currently supports detecting two kinds of dataflow bugs: null-pointer dereferences and dead assignments. The analyses implemented in IntraJ are instances of the *Monotone frameworks* [NNH10].

Monotone frameworks are a theoretical approach for reasoning about program dataflow properties. This approach provides a flexible and generic framework for expressing and solving dataflow equations, which can be used to reason about a wide range of dataflow properties, e.g., *reaching definitions* and *available expressions* analyses.

The dataflow information is propagated through the program using the *control-flow graph* (CFG), available with the functions $pred$ (predecessors) and $succ$ (successors). To propagate information from node n to its succeeding nodes (in the CFG) and to represent the effect of passing through a node we use the following equations:

$$\text{in}(n) = \bigsqcup_{p \in \text{pred}(n)} \text{out}(p) \quad (1) \qquad \text{out}(n) = \bigsqcup_{p \in \text{succ}(n)} \text{in}(p) \quad (3)$$

$$\text{out}(n) = f_{\text{tr}}(\text{in}(n), n) \quad (2) \qquad \text{in}(n) = f_{\text{tr}}(\text{out}(n), n) \quad (4)$$

Equations (1) and (2) are used to propagate information from the predecessors of a node n to n itself. Each instantiation or implementation of these equations corresponds to a *forward* analysis. Similarly, the equations (3) and (4) are used to propagate information *backward* in the CFG. The function f_{tr} is called the *transfer function* of the analysis and captures the effect of passing through a node in the CFG.

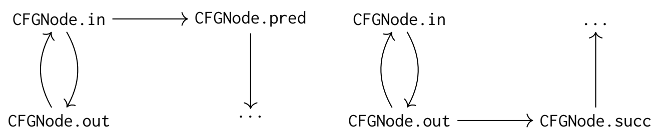


Figure 8: Static call graph for forward (left) and backward (right) analysis.

Benchmark	Start up			Steady State		
	RelaxedMonolithic	RelaxedStacked		RelaxedMonolithic	RelaxedStacked	
	Time (s)	Time (s)	Speedup	Time (s)	Time (s)	Speedup
`antlr`	3.16±0.07	1.83±0.03	× 1.73 ↑	1.42±0.01	0.51±0.01	× 2.78 ↑
`pmd`	6.49±0.12	3.48±0.05	× 1.86 ↑	3.61±0.02	1.39±0.02	× 2.60 ↑
`struts`	9.32±0.18	5.31±0.09	× 1.75 ↑	5.18±0.07	2.17±0.06	× 2.38 ↑
`fop`	8.38±0.09	4.74±0.05	× 1.77 ↑	5.57±0.03	2.08±0.09	× 2.68 ↑
`extendj`	40.88±0.74	16.11±0.21	× 2.54 ↑	37.16±0.66	12.94±0.35	× 2.87 ↑
`castor`	11.10±0.25	6.89±0.14	× 1.61 ↑	6.96±0.04	3.16±0.15	× 2.21 ↑
`weka`	28.12±0.09	12.93±0.12	× 2.17 ↑	23.50±0.20	9.31±0.19	× 2.52 ↑
`poi`	34.63±0.23	16.17±0.12	× 2.14 ↑	27.85±0.25	11.14±0.08	× 2.50 ↑

Table 3: Performance of *dead assignment analysis*, comparing the RelaxedMonolithic and RelaxedStacked algorithms in startup and steady state.

Benchmark	Start up			Steady State		
	RelaxedMonolithic	RelaxedStacked		RelaxedMonolithic	RelaxedStacked	
	Time (s)	Time (s)	Speedup	Time (s)	Time (s)	Speedup
`antlr`	28.09±0.28	2.48±0.06	× 11.34 ↑	26.74±0.09	0.73±0.01	× 36.65 ↑
`pmd`	32.36±0.20	4.46±0.09	× 7.26 ↑	28.14±0.10	1.73±0.01	× 16.24 ↑
`struts`	66.97±0.85	6.42±0.10	× 10.43 ↑	61.74±0.74	3.54±0.14	× 17.45 ↑
`fop`	83.52±0.26	6.04±0.09	× 13.84 ↑	79.12±0.06	2.99±0.08	× 26.48 ↑
`extendj`	1510.75±5.99	13.04±0.08	× 115.87 ↑	≥ 300.00 ⏱	9.55±0.08	≥ 31.42 ↑
`castor`	143.99±4.06	8.54±0.27	× 16.85 ↑	137.35±1.85	4.59±0.13	× 29.93 ↑
`weka`	475.89±2.66	17.21±0.45	× 27.65 ↑	≥ 300.00 ⏱	11.88±0.32	≥ 25.25 ↑
`poi`	571.56±4.47	21.14±0.16	× 27.03 ↑	≥ 300.00 ⏱	15.32±0.08	≥ 19.59 ↑

Table 4: Performance of *Null-Pointer Dereference Analysis*, comparing the RelaxedMonolithic and RelaxedStacked algorithms in startup and steady state. The ⏱ symbol indicates that the analysis timed out.

In IntraJ, in, out, $succ$, and $pred$ are represented by attributes. Figure 8 shows the static call graphs for a forward and a backward analysis. In both graphs, the `CFGNode` class represents a node in the CFG. The attributes `CFGNode.in` and `CFGNode.out` are circular and require a fixed-point computation to compute their values. Our tool CAT will detect that both `CFGNode.pred` and `CFGNode.succ` can never be on a cycle and can thus be declared as NonCircular.

Performance For IntraJ we conducted the evaluation on two dataflow analyses, namely the *null-pointer dereference* and the *dead assignment* analyses. The null-pointer dereference analysis detects expressions that may cause a null-pointer dereference. The dead assignment analysis detects assignments that are never used. Both the analyses are monotone frameworks, with the difference that the null-pointer dereference analysis is a forward analysis (see equations (1) and (2)), while the dead assignment analysis is a backward analysis (see equations (3) and (4)).

Each analysis is done by querying an attribute in IntraJ that collects all warnings in the benchmark program. This attribute will in turn demand the dataflow `in`/`out` attributes, which in turn demand the `pred`/`succ` attributes. These attributes may in turn demand name- and type analysis attributes as defined by the underlying compiler ExtendJ. Thus, in these analyses, many attributes will be demanded downstream from the circular dataflow attributes. It is therefore expected that RelaxedStacked will perform better than RelaxedMonolithic.

Table 3 and Table 4 show the performance of the RelaxedMonolithic and RelaxedStacked algorithms for both the dead assignment and the null-pointer dereference analyses. The start up measurements include both parsing and analysis and the steady state measurements include only analysis. The results show significant performance improvements for the RelaxedStacked algorithm compared to the RelaxedMonolithic algorithm. For dead assignment analysis, the startup speedup of RelaxedStacked ranges from $\sim$1.7x to $\sim$2.5x, with a median speedup of around $\sim$1.8x. In steady-state, the speedup becomes even more significant, ranging from $\sim$2.2x to $\sim$2.8x, with a median speedup of around $\sim$2.5x. One reason for the speedup is that RelaxedMonolithic will compute the control-flow graph (`succ` and its downstream attributes) in each fixed-point iteration, whereas for RelaxedStacked `succ` will be classified as NonCircular, and will only be computed once.

For null-pointer dereference analysis, the results show an even more significant improvement for RelaxedStacked, with a speedup between $\sim$7x to $\sim$115x for startup performance, with a median of $\sim$15.3x. For steady state performance, the speedup was between $\sim$16x to $\sim$35x, with a median of $\sim$22x, disregarding 3 measurements that timed out for RelaxedMonolithic. The reason for the larger difference and variation is that this is a forward analysis which uses the `pred()` attribute which is defined as the reverse of the successor, leading to even more downstream attributes being unnecessarily reevaluated

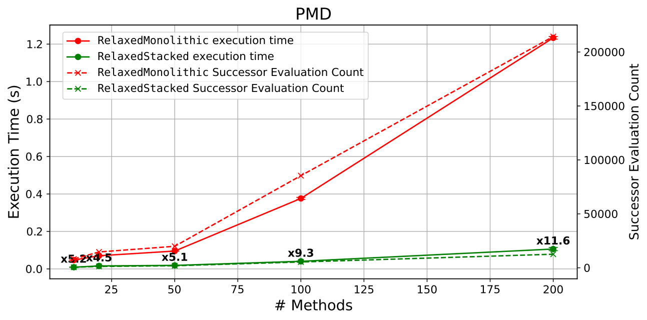


Figure 9: Steady-state performance of null-pointer dereference analysis for randomly selected sets of methods of the pmd benchmark. Solid lines represent execution time (left axis, seconds). Dashed lines represent successor attribute evaluations (right axis, count).

for the RelaxedMonolithic algorithm. We can observe an extremely high speedup for the `extendj` benchmark, where the RelaxedStacked algorithm is approximately 115 times faster than the RelaxedMonolithic algorithm. Upon closer inspection, we discovered that the `extendj` benchmark has a very large generated method for parsing Java source code, consisting of 6844 lines of code, and where the problems of RelaxedMonolithic become particularly pronounced.

The experiments in Table 3 and Table 4 analyze complete benchmark programs. To further demonstrate the on-demand nature of the algorithms, we ran the analyses on sets of randomly selected methods, querying an attribute summarizing the results for each of the selected methods. For each benchmark, we randomly selected 10, 20, 50, 100, and 200 methods to run the experiment, and report the steady-state performance of the analyses. We present the results exclusively for pmd as findings across other projects are similar. Figure 9 shows the results for the null-pointer dereference analysis. We report both the execution time and the number of times a `succ` attribute was evaluated, and it can be observed that these metrics correlate closely. We can also note that the speedups for RelaxedStacked are consistent with the earlier results in Table 3 and Table 4 running on the whole benchmark, approaching similar numbers as the number of methods increases. This experiment demonstrates the on-demand nature of the algorithms, resulting in very short response times when only a subset of the results are demanded, and with similar performance profiles as for the complete programs.

6 Related Work

Knuth's original attribute grammars [Knu68] disallowed cyclic dependencies. Farrow [Far86] and Jones et al. [JS86] independently introduced circular but well-defined attribute grammars. Farrow's approach statically analyzes dependencies, while Jones' relies on a dynamic dependency graph to identify strongly connected components and supports incremental evaluation. Sasaki and Sassa extend attribute grammars with remote links [SS03], a restricted form of reference attributes, and describe exhaustive circular evaluation over them. In contrast to our work, none of these approaches support demand evaluation, nor general reference attributes. Sasaki and Sassa's remote links must be set before evaluation, i.e., may not be computed by attributes.

Boyland describes demand-driven evaluation for circular attributes in the presence of so-called remote attributes (similar to reference attributes), but gives no explicit evaluation algorithm [Boy96]. Hesamian recently added statically scheduled support for circular attributes to Boyland's remote attribute system APS [Hes23]. However, this implementation is exhaustive (not demand-driven) and the experimental results are limited to comparatively small grammars and synthetic input. The largest grammar in this work is the nullable-first-follow grammar from Magnusson [MH07c] that we discuss in Section 5.2.

Previous demand-driven algorithms for RAGs with circular attributes include BasicStacked$_{\text{old}}$ by Magnusson et al. [MH07c] and RelaxedMonolithic by Öqvist et al. [ÖH17; Öqv18]. Our algorithm generalizes both. Öqvist further presents a concurrent lock-free attribute evaluation algorithm based on RelaxedMonolithic, while Söderberg et al. [SH15] present an extension of BasicStacked to handle circular higher-order attributes. Both contributions are orthogonal to the ones presented here.

Kiama [SKV10] and Silver [Van+10a] are two RAG systems that could benefit from using the RelaxedStacked algorithm. Kiama already supports circular attributes by implementing one of the basic algorithms described by Magnusson et al. [MH07c].

Logic programming, especially in Datalog, is the basis for other declarative approaches to program analysis. Datalog-based analysis frameworks include Doop [BS09], which supports points-to analysis of Java bytecode, and the commercial .QL system [DM+07]. These generally follow a two-phase process that first extracts program facts into a database and then evaluates logical rules until it reaches a fixed point, though Dura et al. describe how both phases can be integrated into a single declarative framework [DRS21]. While these approaches are often limited to boolean lattices, Madsen et al. [MYL16a] demonstrate a Datalog variant with support for general complete finite-height lattices. Unlike our work, Datalog frameworks generally use exhaustive evaluation, though some Datalog-based tools use on-demand evaluation strategies based on logical rule rewriting to use so-called Magic Sets [Ban+85], or hybrid strategies, as in the Clog framework [DR24], which combines exhaustive evaluation with on-demand queries to a compiler frontend.

Stein et al. present a general approach to demand-driven abstract interpretation over a pre-computed CFG, with cyclic computations over infinite-height domains [SCS21], though their experimental results are limited to synthetic workloads. Other demand-driven approaches range from frameworks for distributive interprocedural dataflow analysis [DGS95; HRS95] to points-to analysis for full languages like Java [Sri+05; Spä+16]. It is an area of future work to investigate how the RAG approach can be applied to similar problems.

7 Conclusion

We have presented a new formulation of demand-driven evaluation of Reference Attribute Grammars with circular (fixed-point evaluated) attributes. Our approach integrates three attribute kinds, Circular, Agnostic, and NonCircular, in our new RelaxedStacked algorithm, improving upon previous algorithms that only supported combining Circular with either NonCircular or Agnostic attributes.

Our experiments show that effective use of NonCircular attributes is crucial to efficient evaluation. Since manually selecting NonCircular attributes is challenging and error-prone, we perform a call graph analysis on the RAG to automatically identify NonCircular attributes, ensuring correctness and efficiency.

We have evaluated the new algorithm on LL(1) parser construction, on a Java compiler, and on two intraprocedural dataflow analyses for Java. In the parser case study, RelaxedStacked matched the performance of BasicStacked and was 2.8x faster than RelaxedMonolithic. For the Java compiler, RelaxedStacked matched the performance of RelaxedMonolithic, which was expected since this application contains few circular attributes. For the dataflow analyses, we observed substantial speedups for RelaxedStacked over RelaxedMonolithic: a 1.8x median improvement in startup and 2.5x in steady-state for dead assignment analysis, and 15.3x and 22x, respectively, for null-pointer dereference analysis. We also conducted experiments by sampling results from the benchmark programs, demonstrating that our algorithm works efficiently even when only a subset of the program's results are demanded.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

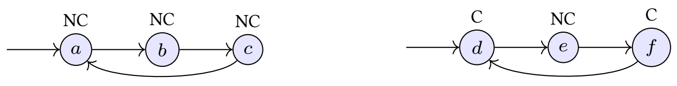


Figure 10: Attributes a, b, c and e are incorrectly specified as `NonCircular`.

Appendices

A Safe Evaluation of Incorrectly Specified RAGs

This appendix details how a runtime error is raised if an Agnostic or NonCircular attribute is incorrectly specified.

Safe Evaluation of Incorrectly Specified NonCircular Attributes

It is important that the algorithms are safe in that they do not compute the wrong result even if the developer incorrectly declares an attribute as NonCircular, while for some AST, it turns out to be effectively circular. Rather, a runtime error should be raised in this case. In our algorithm, a NonCircular attribute on a cycle will lead to endless recursion, and therefore raise a stack overflow error. As an alternative solution, it would be straightforward to extend the algorithm to use one additional flag per NonCircular attribute instance to track and report such circular dependencies. (The code for this solution is elided for brevity.)

To see that the algorithm is safe in this respect, we can consider two cases, as shown in Figure 10. In the left example, all the attributes a, b, c, are (incorrectly) declared as NonCircular, although they are on a cycle. Suppose that evaluation starts by calling a. All the attributes will take the **`NORMAL`** branch in the algorithm, and just continue calling each other in an endless recursion, eventually leading to stack overflow.

In the example to the right, the attribute e is (incorrectly) declared as NonCircular and the others (correctly) as Circular. Suppose that evaluation starts in one of the Circular attributes, say d. It will become a driver, start a fixed-point evaluation, and start an iteration with a unique id, say 1. This id is saved in its `d_iter` instance variable. When the evaluation reaches e, it takes the **`BRIDGE`** branch, and stacks the current circular evaluation. The evaluation then reaches f which will become the driver of a new fixed-point evaluation, with a new unique iteration id, say 2. When the evaluation reaches d again, it will become a follower since circular evaluation is ongoing. It will call its compute method since its stored iteration id (1) differs from the current one (2). When the evaluation again reaches e, it again takes the **`BRIDGE`** branch, and stacks the current evaluation. The evaluation continues this way, stacking cyclic evaluation for every visit to the f attribute, leading to endless recursion and eventually stack overflow.

If the evaluation instead starts in the NonCircular e, it will first take the **`NORMAL`** branch, but at the next visit, it will take the **`BRIDGE`** branch, and lead to the same kind of endless recursion.

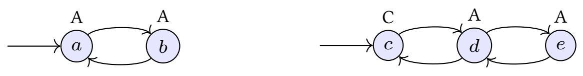


Figure 11: Attributes a, b, d and e are incorrectly specified as Agnostic.

Safe Evaluation of Incorrectly Specified Agnostic Attributes

If an Agnostic attribute is on a cycle without any Circular attribute, the algorithm must be safe in that it does not return an incorrect value, but instead raises a runtime error. As for the NonCircular attributes, we will use endless recursion, i.e., stack overflow, to identify such an error. (As for NonCircular attributes, an alternative solution using a boolean flag could be used, but elided here for brevity.)

To see that the algorithm is safe in this respect, we consider two cases, as shown in Figure 11. In the left example, an incorrectly specified Agnostic attribute a is called from outside any cyclic evaluation. It thus enters the **`NORMAL`** code, and calls b, which also enters its **`NORMAL`** code. Then b calls a which again enters the **`NORMAL`** code. We see that this leads to endless recursion and eventually stack overflow.

In the right example, an incorrectly specified Agnostic attribute d is called from inside a cyclic evaluation. Here, the evaluation starts with the Circular attribute c which becomes the driver. When d is reached, it will execute the **`FOLLOWS`** code, and call e. The e attribute is also Agnostic, and also executes the **`FOLLOWS`** code, and calls d again. Since the value of `attr_iter` for d is unchanged, and thus still different from `CIRCLE_ITER`, the d attribute will again execute the **`FOLLOWS`** code, and we have endless recursion, eventually leading to stack overflow. It is important that the `attr_iter` is not set until after the call to `compute` to get this behavior (relating to Note 4 in Section 3.6).

B Case Study: ExtendJ

ExtendJ [EH07b] is a Java compiler supporting Java 11, built using the meta-compilation system JastAdd [HM03]. ExtendJ uses the RelaxedMonolithic algorithm introduced by Öqvist [ÖH17; Öqv18]. It cannot be run with the BasicStacked algorithm because it includes a number of attributes that are effectively circular only on rare occasions, and that are not declared as Circular. It can be run with our new RelaxedStacked algorithm, but we do not expect big performance differences. The reason is that ExtendJ has relatively few circular attributes, and these are typically downstream from the error checking and code generation attributes that drive the attribute evaluation.

In Table 5 we present performance results for the ExtendJ compiler, showing both startup and steady-state performance for both RelaxedMonolithic and RelaxedStacked. As expected, the results for the two algorithms are very similar.

Benchmark	Start up			Steady State		
	Relaxed-Monolithic	Relaxed-Stacked		Relaxed-Monolithic	Relaxed-Stacked	
	Time (s)	Time (s)	Speedup	Time (s)	Time (s)	Speedup
`antlr`	$1.75_{\pm 0.03}$	$1.76_{\pm 0.05}$	$\approx$	$0.42_{\pm 0.00}$	$0.42_{\pm 0.00}$	$\approx$
`pmd`	$4.03_{\pm 0.07}$	$4.03_{\pm 0.08}$	$\approx$	$1.31_{\pm 0.02}$	$1.30_{\pm 0.01}$	$\approx$
`struts`	$5.32_{\pm 0.16}$	$5.14_{\pm 0.16}$	$\approx$	$1.76_{\pm 0.05}$	$1.74_{\pm 0.04}$	$\approx$
`fop`	$4.99_{\pm 0.19}$	$4.96_{\pm 0.15}$	$\approx$	$1.72_{\pm 0.03}$	$1.71_{\pm 0.06}$	$\approx$
`extendj`	$6.77_{\pm 0.14}$	$6.84_{\pm 0.12}$	$\approx$	$3.92_{\pm 0.10}$	$3.90_{\pm 0.04}$	$\approx$
`castor`	$8.17_{\pm 0.29}$	$7.98_{\pm 0.18}$	$\approx$	$3.31_{\pm 0.09}$	$3.20_{\pm 0.03}$	$\approx$
`weka`	$9.88_{\pm 0.16}$	$9.63_{\pm 0.17}$	$\approx$	$4.77_{\pm 0.34}$	$4.63_{\pm 0.09}$	$\approx$
`poi`	$14.18_{\pm 0.40}$	$14.54_{\pm 0.53}$	$\approx$	$8.00_{\pm 0.10}$	$7.86_{\pm 0.06}$	$\approx$

Table 5: Performance of ExtendJ compilation of the benchmark, comparing the RelaxedMonolithic and RelaxedStacked algorithms in startup and steady state. Times in seconds. Speedup is the ratio between RelaxedStacked and RelaxedMonolithic. Results are considered the same if the confidence intervals overlap.

References

[ALL96] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. "Analysis and caching of dependencies". In: *SIGPLAN Not.* 31.6 (1996), 83–91.

[App04] Andrew W Appel. *Modern compiler implementation in C.* Cambridge university press, 2004.

[Ban+85] Francois Bancilhon et al. "Magic sets and other strange ways to implement logic programs". In: *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems.* 1985, pp. 1–15.

[Bla+06] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications.* Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190.

[Boy96] John Tang Boyland. "Descriptional Composition of Compiler Components". PhD thesis. University of California, Berkeley, 1996.

[BS09] Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses". In: *Proceedings of OOPSLA '09.* Orlando, Florida, USA: ACM, 2009, pp. 243–262.

[DM+07] Oege De Moor et al. ".QL: Object-oriented queries made easy". In: *International Summer School on Generative and Transformational Techniques in Software Engineering.* Springer. 2007, pp. 78–133.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis”. In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP ’95. Berlin, Heidelberg: Springer-Verlag, 1995, 77–101.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. “Demand-driven Computation of Interprocedural Data Flow”. In: *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 37–48.

[DR24] Alexandru Dura and Christoph Reichenbach. “Clog: A Declarative Language for C Static Code Checkers”. In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. , Edinburgh, United Kingdom: Association for Computing Machinery, 2024, 186–197.

[DRS21] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. “JavaDL: Automatically Incrementalizing Java Bug Pattern Detection”. In: *Proceedings of the ACM on Programming Languages*. Virtual: ACM, 2021.

[EH07b] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[Far86] Rodney Farrow. “Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*. ACM, 1986, pp. 85–98.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000).

[HM03] Görel Hedin and Eva Magnusson. “JastAdd—an aspect-oriented compiler construction system”. In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[Hes23] Seyedamirhossein Hesamian. “Statically Scheduling Circular Remote Attribute Grammars”. Theses and Dissertations. 3383. PhD thesis. University of Wisconsin-Milwaukee, 2023.

[HDT87] Susan Horwitz, Alan J. Demers, and Tim Teitelbaum. “An Efficient General Iterative Algorithm for Dataflow Analysis”. In: *Acta Informatica* 24.6 (1987), pp. 679–694.

[HRS95] Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. “Demand Interprocedural Dataflow Analysis”. In: *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1995, Washington, DC, USA, October 10-13, 1995.* Ed. by Gail E. Kaiser. ACM, 1995, pp. 104–115.

[JS86] Larry G. Jones and Janos Simon. “Hierarchical VLSI Design Systems Based on Attribute Grammars”. In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986.* ACM Press, 1986, pp. 58–69.

[Knu68] Donald E Knuth. “Semantics of context-free languages”. In: *Mathematical systems theory* 2.2 (1968), pp. 127–145.

[MYL16a] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. “From Datalog to Flix: A Declarative Language for Fixed Points on Lattices”. In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 194–208.

[MH07a] Eva Magnusson and Görel Hedin. *CRAG artifact.* `https://bitbucket.org/jastadd/crag-artifact`. Accessed: 2024-04-01. 2007.

[MH07c] Eva Magnusson and Görel Hedin. “Circular reference attributed grammars — their evaluation and applications”. In: *Science of Computer Programming* 68.1 (2007). Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA ’03), pp. 21–37.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[Öqv18] Jesper Öqvist. “Contributions to Declarative Implementation of Static Program Analysis”. PhD thesis. Lund University, Sweden, 2018.

[ÖH17] Jesper Öqvist and Görel Hedin. “Concurrent circular reference attribute grammars”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017.* Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, 2017, pp. 151–162.

[Rio+24] I. Riouak et al. *Efficient Demand Evaluation of Fixed-Point Attributes Using Static Analysis (Artifact).* 2024.

[Rio+21] Idriss Riouak et al. “A Precise Framework for Source-Level Control-Flow Analysis”. In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE. 2021, pp. 1–11.

[SS03] Akira Sasaki and Masataka Sassa. “Circular Attribute Grammars with Remote Attribute References and their Evaluators”. In: *New Generation Computing* 22.1 (2003), pp. 37–60.

[SKV10] Anthony M. Sloane, Lennart C.L. Kats, and Eelco Visser. “A Pure Object-Oriented Embedding of Attribute Grammars”. In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010). Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 205–219.

[SH11] Emma Söderberg and Görel Hedin. “Automated Selective Caching for Reference Attribute Grammars”. In: *Software Language Engineering*. Ed. by Brian Malloy, Steffen Staab, and Mark van den Brand. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 2–21.

[SH15] Emma Söderberg and Görel Hedin. “Declarative rewriting through circular nonterminal attributes”. In: *Computer Languages, Systems & Structures* 44 (2015), pp. 3–23.

[Söd+13c] Emma Söderberg et al. “Extensible intraprocedural flow analysis at the abstract syntax tree level”. In: *Sci. Comput. Program.* 78.10 (2013), pp. 1809–1827.

[Spä+16] Johannes Späth et al. “Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java”. In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy.* Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 22:1–22:26.

[Sri+05] Manu Sridharan et al. “Demand-driven points-to analysis for Java”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA.* Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, 2005, pp. 59–76.

[SCS21] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. “Demanded abstract interpretation”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 282–295.

[Tar72] Robert Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM Journal on Computing* 1.2 (June 1972), pp. 146–160.

[Tem+10b] Ewan Tempero et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. In: *2010 Asia Pacific Software Engineering Conference (APSEC2010).* Dec. 2010, pp. 336–345.

[Van+10a] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1 (2010). Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA ’06 and ’07), pp. 39–54.

[VSK89a] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. “Higher Order Attribute Grammars”. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation.* PLDI ’89. Portland, Oregon, USA: Association for Computing Machinery, 1989, 131–145.

Popular Science Summary in English

Software plays a crucial role in our modern world, driving everything from smartphones to medical devices. However, as our dependency on software increases, so do the risks associated with software bugs—errors in the code that can lead to serious consequences. As an example, imagine an architect designing a skyscraper in a busy city. Every aspect of the building, from its foundation to the top floor, must be flawless. A minor flaw in the structure, such as a miscalculation in the steel framework, could compromise the entire building, leading to dangerous cracks or even a catastrophic collapse. Similarly, software bugs are like hidden structural weaknesses; they might go unnoticed during development, but if not detected and corrected, they can lead to severe outcomes.

Well known examples of such bugs include the Mars Climate Orbiter crash in 1999, where a software error caused the spacecraft to disintegrate in the Martian atmosphere, and the Therac-25 radiation therapy machine, which delivered lethal doses of radiation to patients due to a software malfunction.

To reduce these risks, developers employ a technique known as *static analysis*. Static analysis involves examining the code for potential errors without actually executing the program. While static analysis is a powerful tool, it can be time-consuming. Typically, developers write their code and then run the static analysis tool to identify issues. During this process, developers must wait for feedback, which can interrupt their workflow and reduce overall productivity. This is similar to inspecting a skyscraper's structure only once a day; if flaws are discovered, builders must backtrack, undo their work disrupting the construction process. Imagine if inspectors could continuously monitor the construction, checking each new addition as it is made without any noticeable overhead. This would allow them to catch and address issues in real time, preventing small problems from becoming major defects.

This thesis explores a new method to enhance the efficiency and precision

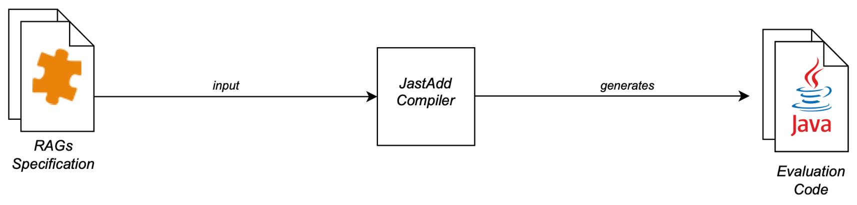


Figure 1: The process of generating code using a declarative approach.

of static analysis, especially when integrated into Integrated Development Environments (IDEs), by using a declarative approach.The goal is to deliver almost instant feedback, enabling developers to correct issues as they write code, rather than waiting hours for results.

Declarative Approach

Declarative programming, in contrast to imperative programming, is a paradigm where developers focus on defining the desired outcomes or properties of the code, rather than detailing the exact steps to achieve them. In other words, the developer specifies *what* should be computed, not *how* it should be computed.

To further clarify, let us revisit the analogy of constructing a building. In a traditional, imperative approach, you would provide detailed instructions for each step of the construction process—laying the foundation, erecting the walls, and installing the roof. In contrast, with a declarative approach, you would simply describe the finished building—its height, materials, and layout—and let a specialized team figure out the most efficient way to build it according to those specifications.

In software development, programming using a declarative approach typically involves writing code in a high-level language that abstracts away the implementation details. Another program, known as a compiler, then takes this high-level specification and generates the lower-level code that can be executed by a computer. This separation allows developers to focus on the *what*—the logic and goals of the program—while leaving the *how* to the underlying system. There exists a variety of declarative programming languages and compilers. In our work we used a formalism called *Reference Attribute Grammars* (RAGs). We implemented our system using a tool called *JastAdd*, which is Java based compiler for RAGs: given a high-level specification of a program, JastAdd generates the corresponding Java code. The process of generating code using a declarative approach is illustrated in Figure 1.

Programs developed using a declarative style are structured into smaller components known as attributes. These attributes can be individually defined and adjusted, simplifying the maintenance and extension of the overall program. In our work, these attributes are combined to create a more complex system, specifically a static analyzer. The advantage of this approach is that it provides a clear and modular structure, making it easier to understand and maintain the code.

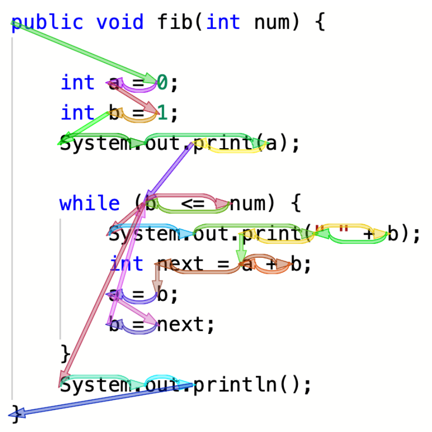
Main Contributions

The main contributions of this thesis are centered around improving the efficiency and responsiveness of static analysis tools using a declarative approach.

The first major contribution is the development of IntraCFG, a framework designed to create control-flow graphs (CFGs). A control-flow graph is a program representation that shows how the execution flows from one instruction to another. A CFG can be viewed as a map of the program's structure, illustrating the possible paths the code can take during execution. These graphs are essential for static analysis tools, as they help to reason about the program's behavior and identify potential issues. Figure 2 shows an example of a CFG for a simple Java program.

Traditional methods for building these graphs can be slow and less precise because they rely on intermediate representations like bytecode. In contrast, IntraCFG builds these graphs directly from the program's structure, known as the Abstract Syntax Tree (AST).

Building on this framework, we created IntraJ, a tool specifically designed to analyze Java code using the IntraCFG framework. IntraJ was tested against SonarQube, a widely-used tool in the industry for static analysis. The results showed that IntraJ was significantly faster than SonarQube while maintaining the same level of accuracy. This means that developers using IntraJ can get quicker feedback on potential issues in their code, allowing them to address problems much sooner.



```
public void fib(int num) {

    int a = 0;
    int b = 1;
    System.out.print(a);

    while (b <= num) {
        System.out.print(" " + b);
        int next = a + b;
        a = b;
        b = next;
    }
    System.out.println();
}
```

Figure 2: Example of CFG on the program's source code for the Fibonacci function. Colors are randomly assigned to make it easier to distinguish individual arrows.

To further enhance the speed of our analysis, we improved the attribute evaluation algorithms used for RAGs with a new algorithm for attribute evaluation called RelaxedStacked. Some attribute values in a program can depend, directly or indirectly, on their own value. If not handled properly, this can lead to repeated calculations that slow down the analysis process. RelaxedStacked identifies which attributes are not recursive (does not depend on itself) by analysing the code generated by *JastAdd*. Once these attributes are identified, the algorithm will generate new code that avoids unnecessary recomputation, by storing the results of the value of each non-recursive attribute immediately. We applied this new algorithm to IntraJ, and the results showed a significant improvement in the analysis speed, making it almost instantaneous (less than 0.1 seconds for analysing a single file).

In conclusion, this research addresses the need for faster and more integrated tools to ensure the quality and safety of software. By making static analysis more responsive, it not only helps developers write better code but it also reduces costs and risks associated with software bugs.