Declarative Specification of Intraprocedural Control-flow and Dataflow Analysis

Idriss Riouak

Licentiate Thesis, 2023

Department of Computer Science
Lund University

ISBN 978-91-8039-586-1 (electronic version)
ISBN 978-91-8039-587-8 (print version)
ISSN 1652-4691
Licentiate Thesis 1, 2023

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, 2023

Abstract

Static program analysis plays a crucial role in ensuring the quality and security of software applications by detecting and fixing bugs, and potential security vulnerabilities in the code. The use of declarative paradigms in dataflow analysis as part of static program analysis has become increasingly popular in recent years. This is due to its enhanced expressivity and modularity, allowing for a higher-level programming approach, resulting in easy and efficient development.

The aim of this thesis is to explore the design and implementation of *control-flow* and *dataflow* analyses using the declarative *Reference Attribute Grammars* formalism. Specifically, we focus on the construction of analyses directly on the source code rather than on an intermediate representation.

The main result of this thesis is our language-agnostic framework, called IntraCFG. IntraCFG enables efficient and effective dataflow analysis by allowing the construction of precise and source-level control-flow graphs. The framework superimposes control-flow graphs on top of the abstract syntax tree of the program. The effectiveness of IntraCFG is demonstrated 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 an educational language, Teal.

IntraJ has proven to be faster than and as precise as well-known industrial tools. The combination of precision, performance, and on-demand evaluation in IntraJ leads to low latency in querying the analysis results. This makes IntraJ a suitable tool for use in interactive tools. Preliminary experiments have also been conducted to demonstrate how IntraJ can be used to support interactive bug detection and fixing.

Additionally, this thesis presents JFeature, a tool for automatically extracting and summarising the features of a Java corpus, including the use of different Java features (e.g., use of *Lambda Expressions*) across different Java versions. JFeature provides researchers and developers with a deeper understanding of the characteristics of corpora, enabling them to identify suitable benchmarks for the evaluation of their tools and methodologies.

Contribution Statement

The following papers are included in this dissertation:

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. DOI: 10.1109/SCAM52516.2021.00009.

An artifact, including the source code and evaluation results, was submitted for the paper at the ROSE (**R**ecognizing and Rewarding **O**pen Science in **SE**) festival and awarded with the badges "*Open Research Objects*" and "*Research Objects Reviewed*".
DOI: 10.5281/zenodo.5296618.

Paper II 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. 236–241s.* Limassol, Cyprus, 2022. DOI: 10.1109/SCAM55253.2022.00033.

An artifact, including the source code and evaluation results, was submitted for the paper at the ROSE (**R**ecognizing and Rewarding **O**pen Science in **SE**) festival and awarded with the badges "*Open Research Objects*".
DOI: 10.5281/zenodo.7053768.

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

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

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 minor contributor to the work

◑ 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

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, the artifact is *"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."*

Sources: `https://icsme2021.github.io/cfp/AEandROSETrack.html`, and `https://cyprusconferences.org/icsme2022/call-for-joint-artifact-evaluation-track-and-rose-festival-track/`.

Acknowledgements

I want to take this opportunity to express my gratitude to the many people who have supported me during this journey.

Firstly, I would like to thank my main supervisor, Görel Hedin. Her expertise, guidance and support throughout this journey have been invaluable. I am deeply grateful for the time, patience, and effort she invested in shaping my research.

I would also like to thank my co-supervisors, Christoph Reichenbach and Niklas Fors, for their contributions and advice. Their feedback, guidance and experience were essential in my research progress.

I would also like to extend my heartfelt thanks to my friends and colleagues, Momina Rizwan, Matthias Mayr, Anton Risberg Alaküla, and everyone in the Computer Science Department at Lund University. Their kindness, humour, and friendship were the bright spots in this challenging process.

Lastly, I want to express my deepest gratitude and love to my wife, Marta Rotari. Her unwavering support, love and encouragement have been invaluable.

To all of you, thank you from the bottom of my heart for all your support and help in completing this thesis.

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).

Tack så mycket!

Thank you very much!

Grazie mille!

Contents

Abstract **iii**

Contribution Statement **v**

Acknowledgements **vii**

I Introduction **1**

1 Introduction 1
2 Background 4
2.1 Automatic Program Analysis 4
2.2 Control-flow analysis 6
2.3 Dataflow Analysis 8
2.4 Attribute Grammars 11
2.5 Reference Attribute Grammars 15
2.6 The JastAdd Metacompiler 15
Interfaces 20
3 IntraCFG: Intraprocedural Framework for Source-Level Control-Flow Analysis 21
3.1 Overall Architecture 21
3.2 The IntraCFG Framework 22
3.3 IntraJ: IntraCFG for Java 24
3.4 Performance and Precision 27
IntraJ Artifact Evaluation 30
4 IntraTeal: IntraCFG for Teal 31
5 IDE Integration 36
5.1 LSP support via MagpieBridge: warnings, quick-fixes and bug explanations 36
5.2 Visualisation via CodeProber 37

6 JFeature: Java Feature Extractor 39
7 Conclusions and Future Work 41
References 43

Included Papers 49

I A Precise Framework for Source-Level Control-Flow Analysis 51
1 Abstract 51
2 Introduction 52
3 RAGs and the IntraCFG framework 53
3.1 RAG frameworks for control flow 54
3.2 The IntraCFG framework 55
3.3 Computing the successor attributes 57
3.4 Computing predecessors 58
4 IntraJ: IntraCFG implementation for Java 7 60
4.1 Statements and Expressions 60
4.2 Static and Instance Initialisers 62
4.3 Exceptions Modelling 63
5 Client Analysis 66
5.1 Null Pointer Exception Analysis 66
5.2 Live Variable Analysis 68
5.3 Dead Assignment Analysis 68
6 Evaluation and Results 69
6.1 Precision 69
6.2 Performance 71
7 Related Work 74
8 Conclusions 75
References 76

II JFeature: Know Your Corpus 81
1 Abstract 81
2 Introduction 81
3 JFeature: automatic feature extraction 83
3.1 Java version features 83
3.2 Collecting features 85
4 Corpora Analysis 86
4.1 Corpora Description 86
4.2 Evaluation 88
5 Extensibility 90
6 Use cases for JFeature 91
6.1 Longitudinal Study 92
6.2 Project mining 92

7 Related work 93
8 Conclusions 93
References 94

Introduction

1 Introduction

In the past few decades, software has become increasingly important in all systems. As our reliance on software increases, the repercussions of bugs can become more severe, resulting in substantial financial losses and even loss of life. Well-known examples of software bugs include the *Therac-25* radiation therapy machine [LT93], the *Mars Climate Orbiter* crash [Saw99], resulting in a 327 million dollars loss, and the *Toyota unintended acceleration* [Kan+10]. Not only can software bugs result in financial losses and harm people, but they can also negatively impact the environment, as seen in the *Deepwater Horizon* oil spill [SL10].

Program analysis is a branch of computer science that aims to study the behaviour and properties of computer programs. Program analysis plays a crucial role in software development and maintenance, as it helps to ensure the expected functioning of software systems and to identify potential bugs or security vulnerabilities such as the ones mentioned above. In this thesis, we focus on *static program analysis*. Static program analysis (*static analysis* for short) is *"the art of reasoning about a program's behaviour without executing it"* [MS18]. It is an essential technique for improving the quality and reliability of software systems and has been widely used in various applications such as safety [Cou+05; Bla+02] and security [PKB21; Arz+14; Aye+08; Say+22; FD12], performance optimisation [Aho+07; App04], and software maintenance [GDMH12]. Static analysis aims to identify potential errors, bugs, or vulnerabilities in a program before it is executed. By examining the source code of a program, static analysis can provide a detailed and precise understanding of its behaviour, including its control flow [All70], dataflow [KU77], and potential interactions with other system components.

One of the fundamental techniques used in static analysis is *dataflow analysis*, which focuses on the flow of data through a program. Dataflow analysis applications are used to identify potential sources of errors, such as uninitialised variables or null dereference [KSS17], and to optimise the program's performance by identifying opportunities for parallelisation or other forms of optimisation [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. However, more recently, there has been a growing interest in using declarative paradigms for dataflow analysis, 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. There are several declarative languages for specifying dataflow analysis, such as *FlowSpec* [SWV20], a domain-specific language, or the functional language *Flix* [MYL16b].

In this thesis, 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 syntax and semantics of languages, and as such, they are widely used in the development of compilers and static analysis tools. Our implementation is built upon the ExtendJ [EH07a] Java compiler, which is written in JastAdd [HM01]. The JastAdd system supports RAGs and implements demand-driven evaluation algorithms, ensuring that properties are evaluated only when necessary, reducing the overall evaluation time by avoiding redundant computations.

The primary goal of this thesis is to explore the definition of static analysis frameworks using RAGs. Our intention is to exploit the declarative nature of RAGs and demand-driven evaluation to explore the possibility of using static analysis directly in the development process, i.e., in the Integrated Development Environment (IDE). Our attention is mainly directed towards intraprocedural analyses, including control-flow and dataflow analysis, which involve examining the behaviour of a method or function in isolation (i.e., without considering the interactions with other methods or functions).

We initially employed the older RAG-based framework, JastAddJ-intraflow [Söd+13], but we encountered some limitations, including the construction of imprecise control-flow graphs (CFGs). A CFG is a directed graph that represents all possible paths in a program, and represents the evaluation order of the program. An imprecise CFG can lead to incorrect results in subsequent analyses, such as dataflow analysis.

Therefore, we developed a more general framework called IntraCFG. We evaluated the precision of IntraCFG with two instances of the framework, IntraJ and IntraTeal. IntraJ is an instance of IntraCFG for the Java programming language built upon the ExtendJ compiler. We compared IntraJ's performance and precision with the industrial tool SonarQube [Son]. Our results show that IntraJ is more efficient and as precise as SonarQube. Additionally, we implemented IntraTeal, an instance of our framework for the teaching language Teal, which has been a supportive tool for teaching control-flow and dataflow analyses. Finally, we initiated an exploratory study on integrating IntraJ and IntraTeal in interactive tools, e.g., IDEs.

While evaluating IntraJ, we found it challenging to locate suitable benchmark suites. To overcome this issue, we designed and implemented JFeature, a static analysis tool for automatically extracting features from a Java codebase. JFeature allows researchers and developers to explore the characteristics of a codebase, including the use of different Java features (e.g., use of Lambda Expressions) across different Java versions, facilitating the identification of suitable corpora for evaluating their tools and methodologies.

The next section of this thesis provides background on the underlying concepts and techniques used in this work. We then present our main contributions:

- IntraCFG: a language-agnostic framework for intraprocedural analysis. An overview of this contribution can be found in Section 3, and more details are presented in Paper 1.
- IntraJ: an instance of IntraCFG for Java. We evaluate IntraJ compared to an industrial tool, showcasing its performance and precision. An overview of this contribution can be found in Section 3.3, with more details provided in Paper 1.
- IntraTeal: an instance of IntraCFG for teaching control-flow and dataflow analysis using the Teal language. This contribution is described in Section 4.
- Integration into IDEs: we have conducted exploratory experiments on integrating IntraJ into IDEs. The preliminary results of this contribution are described in Section 5.
- JFeature: a tool for automatically extracting and summarising key features of a Java codebase. An overview of this contribution can be found in Section 6, with more details presented in Paper 2.

To conclude, we will present our conclusions and possible future work in Section 7.

2 Background

This section provides an overview of the research that underlies the main results of this thesis, with a focus on *program analysis*. Specifically, we will delve into the concepts of dataflow analysis [Aho+07; NNH10], control-flow analysis [All70], (Reference) attribute grammars [Knu68; Hed00], and their implementation through the JastAdd metacompiler.

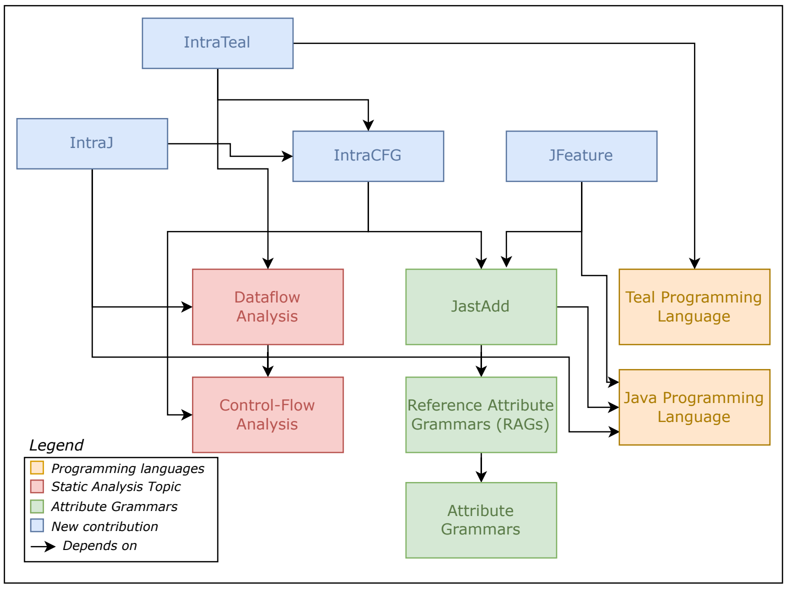


Figure 1: Dependency graph of the concepts discussed in this Section.

The dependency graph in Figure 1 shows the relationship between the concepts and the contributions of this thesis.

2.1 Automatic Program Analysis

Automatic Program Analysis is a branch of computer science that aims to automatically analyse and evaluate programs' properties, e.g., *correctness*, *liveness* and *safety*. We can distinguish two main approaches to program analysis: *static* and *dynamic* analysis.

Dynamic analysis examines the behaviour of a program by executing it. Precise information about a single program's execution are gathered and used to determine the properties of the program. This approach is effective in detecting runtime errors, such as memory leaks [Lab03], performance bottlenecks [Int], and security vulnerabilities [LZZ18]. However, its limitations come from the requirement for complete and accurate input data for a single run, and the difficulty in accounting for all possible execution paths.

In contrast, static analysis performs the analysis without executing the program, relying on information gathered from the source code. This approach has the advantage of being more exhaustive, as it can analyse the entire program. One limitation of static analysis is that it can result in *false positive* and *false negative* results. False positive results occur when the analysis incorrectly reports a problem in the code, whereas false negative results occur when the analysis fails to report an actual problem in the code. The *soundness* and *completeness* of the analysis determine the accuracy of the results. Soundness refers to the property that if the analysis reports a problem, then there is indeed a problem in the code. Completeness refers to the property that if there is a problem in the code, the analysis will eventually report it. In practice, it is often difficult to achieve both soundness and completeness, which leads to the trade-off between false positive and false negative results in static analysis.

Important kinds of static analysis include *Type* and *Effect analyses* [NN99]. Type Analysis aims to determine the type of variables and expressions in a program, which can be used to identify type mismatches, type errors, and improve code readability. Effect analysis focuses on the study of side effects that a program can produce, such as the mutation of data. This information can be used to identify potential bugs and improve program understandability.

In this thesis, we focus on intraprocedural control-flow and dataflow analysis, that are crucial for implementing many type and effect analyses. Intraprocedural control-flow analysis focuses on determining the order in which statements and expressions within a single method in a program will be executed, without considering any function or method calls that may occur. Intraprocedural dataflow analysis uses the control-flow information to determine properties on the flow of data within a single method. Our implementations are instances of the Monotone Framework, a mathematical framework that provides a foundation for dataflow analysis. In the future, we aim to improve the precision of the analysis by considering the implementation of interprocedural analysis, which involves analysing the flow of data across multiple procedures in a program.

Precision is a crucial aspect of dataflow analysis. Depending on the desired outcome, various levels of precision can be achieved, but it is important to weigh the trade-off between precision and performance. A higher level of precision often requires increased computational resources, leading to a decrease in performance, and vice versa. Hence, finding the right balance between precision and performance is crucial in achieving effective and efficient dataflow analysis results.

In our research, we have chosen to compute control-sensitive and exception-sensitive control-flow analyses. Control-sensitivity refers to the ability of the analysis to distinguish between true and false branches of conditional statements, such as if-statements, and taking into account the implicit side effects of the conditional expressions. Exception-sensitivity refers to the ability of the analysis to distinguish between the normal and exceptional execution paths of a program. This approach simplifies the specification of dataflow analyses which are built upon the control-flow analysis.

In terms of implementation strategies, there are various approaches that can be used, including Datalog [DRS21], functional programming [MYL16a], or ad-hoc implementation. However, we choose to implement the analysis using Reference Attribute Grammars [Hed00]. This approach allows us to exploit the benefits of modularity, high-level programming, and on-demand evaluation, which results in a flexible and efficient implementation.

We recognise that the results of our analyses, while effective, may not be sound nor complete and that the analysis may not be able to identify all the bugs or vulnerabilities in a program [Liv+15].

2.2 Control-flow analysis

Control-flow analysis 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: on the source level and the intermediate representation (IR). The source-level approach involves analysing the source code of a program and constructing the CFG directly from the source code on top of the abstract syntax tree. The IR approach involves first converting the source code into an IR, e.g., bytecode, and then constructing the CFG from the IR.

The construction of the CFG at the source level presents several advantages. One of the main benefits is the ability to map the analysis results directly back to the source code and present it to the user in the context of the original program. On the other hand, if the CFG is constructed at the IR level, there would be a complex translation step required to find the corresponding source level constructs. In some cases it would not even be possible due to the loss of information during the translation process, e.g., `source-file retention` policy for Java annotations. For this reason, constructing the CFG at the source level is particularly useful for debugging and program understanding tasks, as it provides a clear and direct representation of the program. Additionally, this approach enables faster and more efficient analysis as it eliminates the overhead of IR generation and can also handle semantically and syntactically invalid code, making it useful for analysing programs with errors or incomplete code. Furthermore, in situations where IR generation must occur in real-time, such as when the analysis is performed in an IDE, the overhead of code generation and optimisation may cause latency in the IDE and frustration for the developer [Pis+22]. In these cases, constructing the CFG at the source level may be a more efficient option.

However, there are also some disadvantages to constructing the CFG on the source level. One of the main limitations is that it can be more difficult to accurately capture the control-flow of a program since the source code may contain unsugared constructs, such as macros and preprocessor directives, that can complicate the analysis specification. In comparison to the engineering effort required by the IR approach, the source-level approach presents complex implementation challenges. Despite the advantage of IRs being generally smaller than the source language, thus reducing the number of constructs to be handled in constructing the CFG and potentially reducing the size or complexity of the analysis code, the source-level approach requires a higher level of complexity in its implementation. In addition, the source code may be written in a variety of languages with different syntax and semantics, making it challenging to design a single analysis that works across all languages.

The examples in Figures 2 and 3 show the source-level and bytecode control-flow graphs of a simple method `foo`. Adding *Entry* and *Exit* nodes to a CFG is a common practice to simplify the implementation of dataflow analyses. These nodes represent the unique entry and exit points of the method, respectively. The Entry node serves as a starting point for the analysis, allowing for a proper initialisation of the parameters. The Exit node is particularly useful in backward dataflow analyses (see Section 2.3), as it is used as unique starting point for the analysis.

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

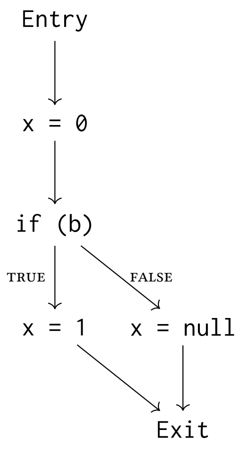


Figure 2: Source level control-flow graph of the `foo` method, showing the branching behaviour of the `if`-statement.

```
1 :0 : iconst_0
2 :1 : invokestatic #7
3 :4 : astore_2
4 :5 : iload_1
5 :6 : ifeq 17
6 :9 : iconst_1
7 :10: invokestatic #7
8 :13: astore_2
9 :14: goto 19
10:17: aconst_null
11:18: astore_2
12:19: return
```

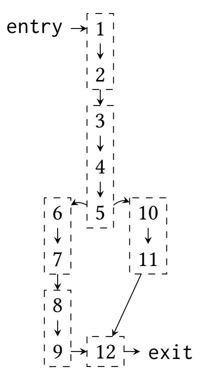


Figure 3: Bytecode control-flow graph of the foo method. Each dashed box represents a basic block.

2.3 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, 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].

[1]For simplicity, just for this example, we assume that the language allows only assignments of

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.

- If n is not an assignment node, no information is updated.

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 $\perp$, and a unique greatest element commonly denoted as $\top$. These elements satisfy the properties that for any element x in the lattice,

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.

$\bot \leq x$ 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*.

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 $f(x) \leq f(y)$.

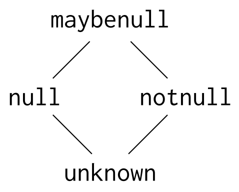


Figure 4: Diagram of the partial order in the example in Section 2.3, showing the order relation between `maybenull` ($\top$), `unknown` ($\bot$), `null`, and `notnull`.

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, `undefined` 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:

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

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

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:

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

where pred(n) is the set of predecessor 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., $\text{succ}(n) = \{s \mid (n,s) \in E\}$. 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 fixpoint computation is used to resolve this circular dependency. A fixpoint is a mathematical technique that finds a stable state, in a system. In the context of the Monotone frameworks, this refers to finding a state where the in and out sets of all nodes in the CFG have reached a stable value, and no further changes will occur. The authors of [NNH10] present several techniques for computing fixpoints. They demonstrate that there exists a fixpoint [KT29] and that the resulting fixpoint is the unique and the minimum among all fixed points of the transfer function f.

2.4 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 behaviour 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 use the notation introduced by Fors et al. in [FSH20], where attribute names are preceded by a symbol (e.g.,$\uparrow$,$\downarrow$) that indicates the kind of the attribute (e.g., 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 instance, the type of the expression "3 + 4", is determined to be integer based on the types of the operands.

Synthesized attributes are composed by a declaration and an equation:

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

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 $\mathsf{A1} <: \mathsf{A}$ and $\mathsf{A2}\ <: \mathsf{A}$, different equations can be given for $\mathsf{A2}$ and $\mathsf{A1}$. If all subtypes will use the same equation, the declaration and the equation can be combined into a single line, as follows:

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

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.

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

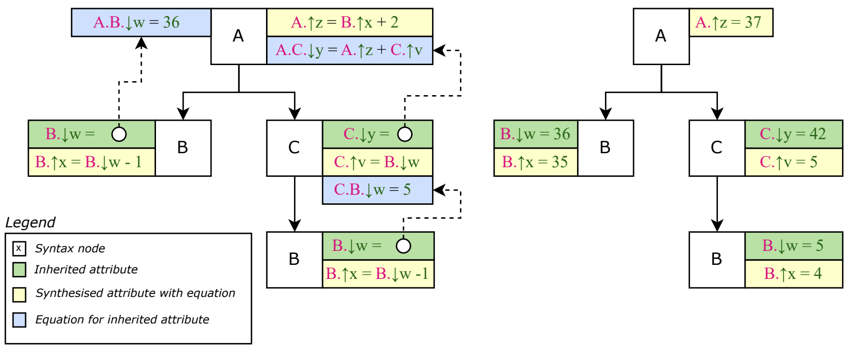


Figure 5: Attribute grammar example. Left: Abstract syntax tree with attribute equation system. Right: solution of the equation system. Dashed arrows indicates the location of the equation for an inherited attribute.

where A and B are node types and the down-arrow symbol (↓) 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.

To better explain the concept of AGs, we present a very simple example. We start by defining the following abstract grammar:

$$
\begin{aligned}
&A ::= B \quad C\\
&B\\
&C ::= B
\end{aligned}
$$

and the following attribute declarations and equations:

Synthesized:	*Inherited:*
int A.↑z = B.↑x + 2	int C.↓y
int B.↑x = B.↓w − 1	int B.↓w
int C.↑v = B.↓w	A.B.↓w = 36
	C.B.↓w = 5
	A.C.↓y = A.↑z + C.↑v

It should be noted that there are two equations for the inherited attribute B.↓w, due to the presence of B as a child node in both A and C. Figure 5 depicts the described example. The attributes and equations are instantiated for the AST, and then form an equation system which, in this case, can be solved by substitution. If we instantiate the equations for the AST in Figure 5, we get the following equation system where all variables (attribute instances) are uniquely named from the root of the tree:

$$\begin{aligned}
&\mathsf{A}.\uparrow\mathrm{z} = \mathsf{A}.\mathsf{B}.\uparrow\mathrm{x} + 2\\
&\mathsf{A}.\mathsf{B}.\uparrow\mathrm{x} = \mathsf{A}.\mathsf{B}.\downarrow\mathrm{w} - 1\\
&\mathsf{A}.\mathsf{C}.\mathsf{B}.\uparrow\mathrm{x} = \mathsf{A}.\mathsf{C}.\mathsf{B}.\downarrow\mathrm{w} - 1\\
&\mathsf{A}.\mathsf{C}.\uparrow\mathrm{v} = \mathsf{A}.\mathsf{C}.\mathsf{B}.\downarrow\mathrm{w}\\
&\mathsf{A}.\mathsf{B}.\downarrow\mathrm{w} = 36\\
&\mathsf{A}.\mathsf{C}.\mathsf{B}.\downarrow\mathrm{w} = 5\\
&\mathsf{A}.\mathsf{C}.\downarrow\mathrm{y} = \mathsf{A}.\uparrow\mathrm{z} + \mathsf{A}.\mathsf{C}.\uparrow\mathrm{v}
\end{aligned}$$

Solving the equation system for the attribute $\mathsf{A}.\uparrow\mathrm{z}$ gives:

$$\begin{aligned}
\mathsf{A}.\uparrow\mathrm{z} =& \mathsf{A}.\mathsf{B}.\uparrow\mathrm{x} + 2\\
=& (\mathsf{A}.\mathsf{B}.\downarrow\mathrm{w} - 1) + 2\\
=& (36 - 1) + 2\\
=& 37
\end{aligned}$$

Solving for $\mathsf{C}.\downarrow\mathrm{y}$, reusing the solved value for $\mathsf{A}.\uparrow\mathrm{z}$, gives:

$$\begin{aligned}
\mathsf{A}.\mathsf{C}.\downarrow\mathrm{y} =& \mathsf{A}.\uparrow\mathrm{z} + \mathsf{A}.\mathsf{C}.\uparrow\mathrm{v}\\
=& 37 + \mathsf{A}.\mathsf{C}.\uparrow\mathrm{v}\\
=& 37 + \mathsf{A}.\mathsf{C}.\mathsf{B}.\downarrow\mathrm{w}\\
=& 37 + 5\\
=& 42
\end{aligned}$$

The attribute value of a node B depends on the context in which it is evaluated, resulting in potentially different values. Specifically, the attribute value of $\mathsf{B}.\downarrow\mathrm{w}$ can be either 36 or 5, depending on the context of the node B. When we solve for $\mathsf{C}.\downarrow\mathrm{y}$, the value of $\mathsf{B}.\downarrow\mathrm{w}$ is defined by node C with value 5.

The previous example presents a straightforward instance of AGs and does not display its full expressiveness. In real-life scenarios, AGs tend to be more complex. Features such as node subtyping and declaration of nodes with multiple children of the same type, play an essential role in expressing the semantics of a programming language.

For example, in the following grammar:

$$\begin{aligned}
&A ::= B_0 \quad B_1\\
&B
\end{aligned}$$

the children of A can be distinguished by the index of the child, i.e., A has two children B with indexes 0 and 1.

2.5 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) were introduced 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 analysis: checks that all names are well-defined and used correctly. A relation between the name and the declaration of the name is constructed,
- Type analysis: checks that all expressions have a valid type. A relation between the expression and its type is constructed,
- Graph of a class hierarchy: a graph where nodes are classes and edges are inheritance relations,
- Control flow graph: a graph where nodes are statements or expressions, and edges are control flow relations, and,
- Call graph: a graph where nodes are methods and edges are method calls.

2.6 The JastAdd Metacompiler

The JastAdd metacompiler [HM01] is a Java-based tool that generates Java code from a RAG specification. The generated code can be used by an analysis tool to construct an AST and to perform analysis on the AST. Another important aspect of JastAdd is its support for on-demand attribute evaluation. Attribute evaluation is performed only when the corresponding attribute is required and triggered by the analysis tool. This approach enables JastAdd to avoid performing unnecessary computations, which can improve the run-time performance of the tool.

The JastAdd metacompiler is based on the following components:

- The JastAdd language: a language for the definition of RAGs. The JastAdd language is used to specify the abstract grammar of a language and, with a Java-like syntax, the attributes of the RAG.
- The JastAdd compiler: that is a compiler that generates Java code from a RAG.

In JastAdd, synthesized attributes are defined using the `syn` keyword followed by the type and the name of the attribute. Similarly, inherited attributes are defined using the `inh` keyword.

Let us reconsider the example depicted in Figure 5. The abstract grammar is defined in a "`.ast`" file with the following syntax:

```
A ::= B C;
B;
C ::= B;
```

where each line defines an AST node type. The attributes are defined in a “`.jrag`” file with the following syntax:

```
1   syn int A.z() = getB().x() + 2;
2   syn int B.x() = w() - 1;
3   syn int C.v() = getB().w();
4   inh int C.y();
5   inh int B.y();
6   eq A.getC().y() = z() + getC.v();
7   eq C.getB().w() = 5;
8   eq A.getB().w() = 36;
```

An equation is defined in the context of a node type, and children are accessed using getters. For example, the equation on line 6 defines the value of the inherited attribute y of an A node’s child C. It uses the A node’s synthesized attribute z and the C child’s synthesized attribute v to compute the value of the inherited attribute y.

An additional key feature of JastAdd is its support of AspectJ-style [Kic+97] intertype declarations for the definition of attributes. Attribute declarations and equations can be written in aspects, and JastAdd creates corresponding Java methods that are woven into the classes defined by the abstract grammar.

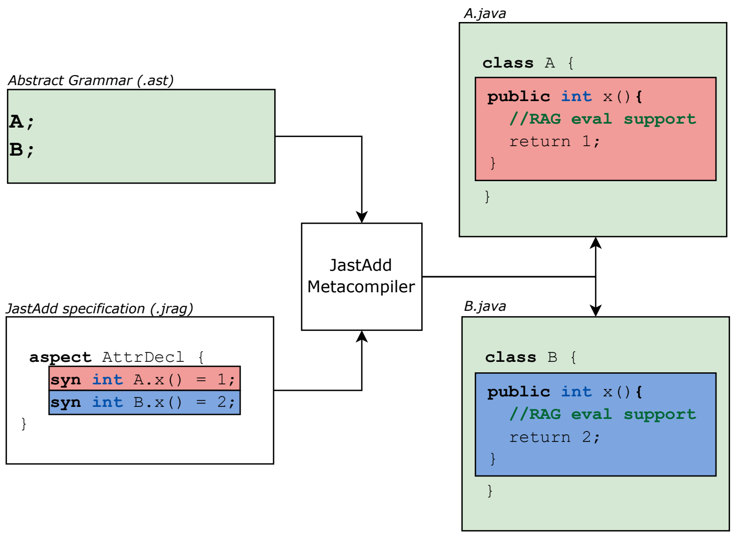


Figure 6: JastAdd generates Java classes from the abstract grammar, and weaves in code corresponding to the intertype declarations in the aspect into the generated classes A and B. Additional RAG code for on-demand evaluation support is elided.

The example in Figure 6 shows an intertype declaration of an attribute. The attributes A.↑x and B.↑x are defined in the aspect AttrDecl and are woven into the classes A and B, respectively.

JastAdd supports not only synthesized and inherited attributes, but also: *Parametrised Attributes* [Hed00], *Higher-Order Attributes* [VSK89], *Circular Attributes* [Far86; JS86], and *Collection Attributes* [Boy96].

Parametrised Attributes

Parametrised Attributes are attributes which value might depend not only on the AST node itself, but also on the value of the arguments supplied to it. Attributes of this kind are widely used, especially in the definition of type-checking rules. For example:

```
syn boolean Type.compatibleType(Type t) = this == t;
```

A more advanced set of rules, such as those desigend to handle subtyping, would be expected for a real language.

Higher-Order Attribute (HOA)

Higher-Order Attribute, also known as *Non-Terminal Attributes (NTA)*, are attributes which value is a freshly new subtree. They are called *Higher-Order Attributes* because they are attributes and, at the same time, non-terminal; therefore, they can be attributed. The subtree computed by an HOA behaves like a normal AST node, i.e., it can be attributed and used in the definition of other attributes. In JastAdd, HOAs are defined using the `nta` keyword. HOAs are widely used to reify information that is not explicit in the source code and, therefore, not present in the AST. For example, we use HOAs to reify a method's entry and exit point. In CFGs, it is common to have a unique entry point and exit point for each method.

```
syn nta Entry FunDecl.entry() = new Entry();
syn nta Exit FunDecl.exit() = new Exit();
```

In our examples, in this thesis, we will often use the right-arrow symbol ($\rightarrow$) to clarify that an attribute is an HOA. For example, FunDecl.$\rightarrow$entry and FunDecl.$\rightarrow$exit.

Circular Attributes

Circular Attributes are attributes which definition might depend directly or indirectly on themselves. In JastAdd, circular attributes are expressed using the `circular` keyword. To guarantee termination, circular attributes are given a bottom starting value, and are evaluated in a fixed-point computation, i.e., the attribute is evaluated until the value of the attribute does not change. The requirements, which are not checked by JastAdd, to guarantee termination are:

- All the possible values computed by the attribute must be placed in a lattice of finite height.
- The intermediate results of the fix-point algorithm must increase or decrease monotonically[2].

We use the symbol A.$\circlearrowleft\uparrow$x to denote a circular synthesized attribute. An example of a circular attribute is the following:

```
syn int A.x() circular[0] = Math.min(3, x()+1);
```

In JastAdd, circular attributes are defined using the `circular` keyword. The circular synthesized attribute A.$\circlearrowleft\uparrow$x begins with a value of 0 and a fix-point computation is performed until the attribute reaches stability. In this example, the attribute evaluates to 3 after the third iteration as the value of the attribute stops changing.

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

Circular attributes can be used to directly encode mathematical recursive equations, such as *in* and *out* in the monotone framework. For example, the equations for the nullness analysis, defined in Section 2.3, can be written as follows:

```
 1  syn Gamma CFGNode.in() circular[new Gamma()] {
 2    Gamma res = new Gamma();
 3    for (CFGNode e : pred())
 4      res.join(e.out());
 5    return res;
 6  }
 7
 8  eq Entry.in() = new Gamma(allVars());
 9
10  syn Gamma CFGNode.out() circular[new Gamma()] {
11    Gamma res = new Gamma(in());
12    res = trFun(res);
13    return res;
14  }
```

The attribute declarations and definitions at Lines 1 and 10, define the in and out attributes, respectively, for all the CFG nodes. The attribute equation ad line 8 define the boundary condition for the entry point of the CFG, i.e., all the variables are mapped to $\perp$.

Collection Attributes

Collection Attributes are not defined using equations but instead using the so-called contributions. The result of a collection attribute is the aggregation of contributions that can come from anywhere in the AST. A contribution clause is associated with an AST node type and describes the information to be included in a collection attribute, possibly under certain conditions. Collection attributes are especially useful, for example, in compiler construction to collect all the semantic errors in a program from anywhere in the AST. In JastAdd, collection attributes are defined using the `coll` keyword.

An example of a collection attribute is the following:

```
coll Set<Errors> Program.errors();
Expr contributes this when !type.compatible(expectedType()) to
    Program.errors();
```

The collection attribute, `Program.errors()`, denoted with Program.□errors, is used to collect all the semantic errors in the program. The contribution clause states that a reference to the `Expr` node (i.e., `this`) is contributed to the collection when the expression type is incompatible with the expected type. There can be any number of contribution clauses for a particular collection attribute, so new kinds of errors can be added simply by adding new contribution clauses.

Other examples of using collections include finding reverse relations, for example, computing the predecessors in a CFG, given the set of successors.

Interfaces

JastAdd supports a mechanism for defining and extending AST classes with interfaces. Interfaces in JastAdd provide a way to extend the behaviour of AST classes. By declaring an interface for an AST class, the class can inherit equations and attributes defined in the interface. This enables developers to modularise the behaviour of AST classes, making it easier to reuse and maintain code. Interfaces can be declared using the `interface` keyword, and, similarly to Java, they can extend other interfaces. By using the `implements` keyword, it is possible to state that an AST class implements an interface. For example, if we want to abstract a common operation for all the CFG nodes, we can define an interface `CFGNode` and then specify that all the CFG nodes implement it. For example:

```
public interface CFGNode;

AddExpr implements CFGNode;
AssignStmt implements CFGNode;
//...

syn Gamma CFGNode.in() circular[new Gamma()] {
  Gamma res = new Gamma();
  for (CFGNode e : pred())
    res.join(e.out());
  return res;
}
```

In this case, both `AddExpr` and `AssignStmt` implement the interface `CFGNode`. We then define the `CFGNode`.↻↑in for the `CFGNode` interface. This attribute is then inherited by all the classes that implement the interface, i.e., `AddExpr` and `AssignStmt`, allowing them to share the common behaviour specified only for `CFGNode`.

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

The techniques of CFG construction have seen significant advancements in recent years, with various frameworks being proposed to aid in the construction of precise intraprocedural CFGs [SWV20; Söd+13]. We contribute to the state-of-the-art by introducing IntraCFG, a declarative, RAG-based, and language-independent framework for constructing precise intraprocedural CFGs. In this section we give an overview of IntraCFG, and more details are available in Paper 1.

Unlike most other frameworks, which build CFGs on the IR level, e.g., bytecode, IntraCFG's approach superimposes the CFGs on the AST. This allows for accurate client analysis, as the CFGs are constructed directly on the source code level rather than an intermediate representation. Differently from the existing RAG-based framework [Söd+13], IntraCFG enables the construction of AST-Unrestricted CFGs, which are CFGs whose shape is not restricted to the AST structure.

3.1 Overall Architecture

The overall architecture of how to use our proposed framework, IntraCFG, is shown in Figure 7. IntraCFG provides the skeleton and default behaviour for constructing CFGs, which can be instantiated for specific languages[3], e.g., Java or Teal. The *Control-flow analysis* (CFA) module is specifically tailored for a particular programming language and implements the interfaces defined in IntraCFG. This module is responsible for constructing the CFG for the program. Once the CFA module has generated the CFG, other client analyses can be added, for example, different dataflow analyses. The compiler module plays a crucial role as it defines the AST nodes on which the CFA module heavily relies. The compiler also provides access to several APIs that can be used by the CFA module and client analyses, such as the type of a variable or the constant value of an expression, to perform advanced optimizations.

[3] **IntraX** in the diagram.

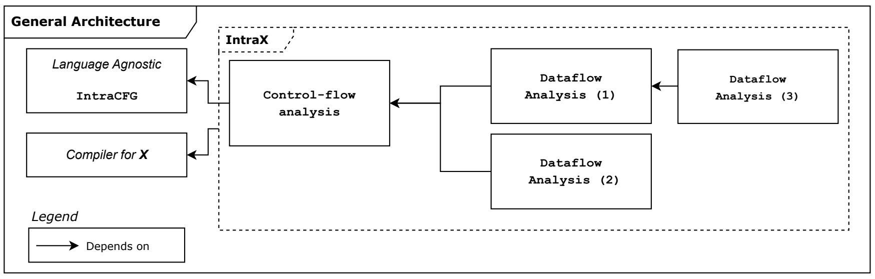


Figure 7: Overall architecture of instantiating IntraCFG for a language **X**.

IntraCFG consists of several key components, including interfaces, attribute equations that define the default behaviour, and APIs. The interfaces provide the structure for the CFG, and the attribute equations define the default behaviour 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.

3.2 The IntraCFG Framework

In this section, we describe the main components of IntraCFG and how they are used to construct CFGs. A more detailed description of the framework can be found in Paper 1.

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 HOAs: `CFGRoot`.$\rightarrow$entry and `CFGRoot`.$\rightarrow$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`.$\uparrow$succ and `CFGNode`.$\uparrow$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 CFGNodes in their subtrees. Indeed, all CFGNodes are CFGSupport nodes, but CFGSupport nodes that are not CFGNodes 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 CFGNodes 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.

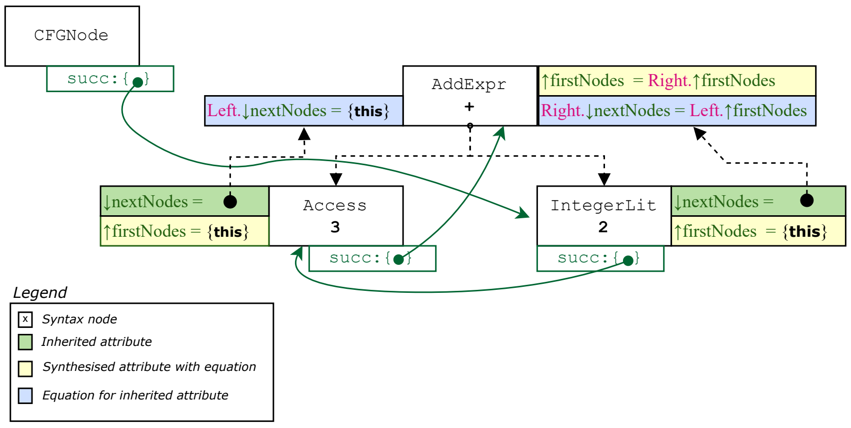


Figure 8: Example of definition of ↓nextNodes and ↑firstNodes for the Teal AddExpr using right-to-left operand evaluation. All the nodes are CFGNodes. Dashed arrows indicates the location of the equation for an inherited attribute.

The example in Figure 8 shows how the ↓nextNodes and ↑firstNodes attributes can be defined to achieve the desired CFG. This example assumes that operands are evaluated right-to-left in order to illustrate the versatility of the approach. The AddExpr and its operands all implement the CFGNode interface. For any AddExpr, the ↑firstNodes attribute is defined to be equal to the ↑firstNodes of its right operand. The AddExpr node also defines the ↓nextNodes attribute of both its operands: for the right operand, it is equal to the ↑firstNodes of the left operand, while for the left operand, it is equal to a singleton set containing the AddExpr node itself. The ↑succ attribute is not overridden by any node, and therefore, it is defined to be equal to the node's ↓nextNodes attribute.

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.

All `CFGNodes` have immediate access to the Entry and Exit nodes of the CFG, through the inherited `CFGNode`.↓entry and `CFGNode`.↓exit attributes declared in `CFGNode` and defined by the nearest `CFGRoot` ancestor.

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 [EH07a].

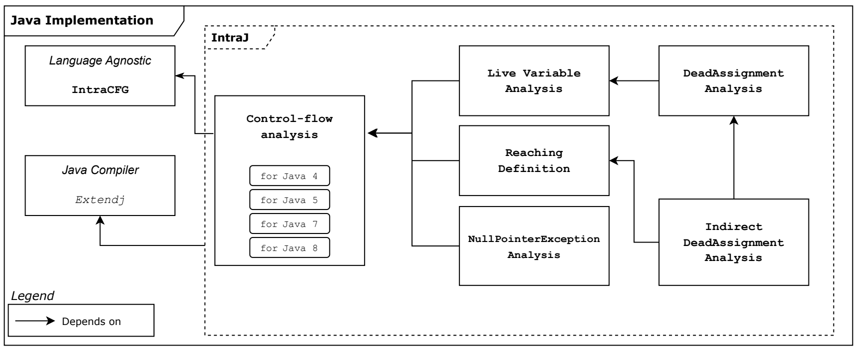
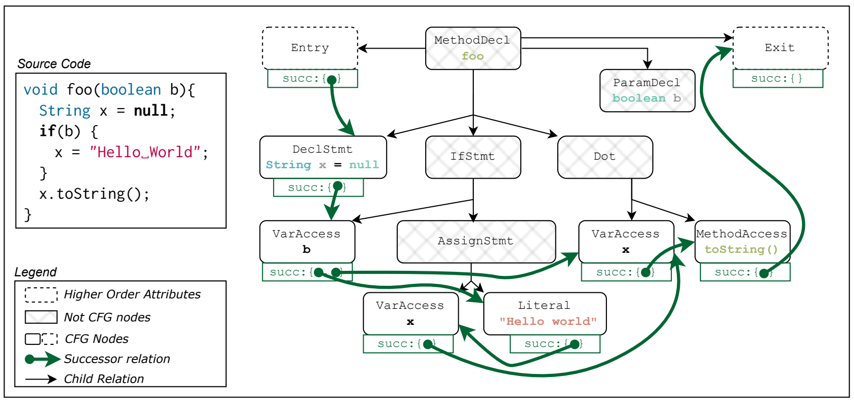


Figure 9: Overall architecture of instantiating IntraCFG for the Java language.

As seen in Figure 9, 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.


Source Code

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

Figure 10: CFG of the `foo` Java method.

The example in Figure 10 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 are live at each point in the program.
- *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 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 2.3). 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 only on the attributes defined by IntraCFG, there are language dependencies in the transfer function, which is modelled as a parametrised attribute. The transfer function attribute is defined for each AST node in the CFG to capture the semantics of passing through that node, and for different AST node types, the transfer function is defined differently. For example, in the case of NullPointerException analysis, the general and language-independent definition of the transfer function is defined as follows:

```
syn CFGNode.trFun(Gamma gamma) = gamma;
```

While the transfer function for AssignStmt is defined as follows:

```
eq AssignStmt.trFun(Gamma gamma) {
  Gamma newGamma = new Gamma(gamma);
  Variable decl = getLHSDecl();
  if (nullRHS()){
    newGamma.put(decl, NULL);
  }
  else{
    newGamma.put(decl, NOTNULL);
  }
  return newGamma;
}
```

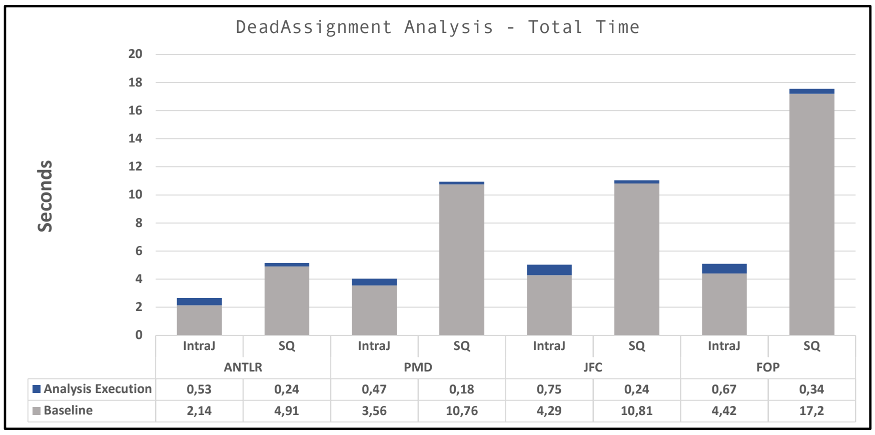
The transfer function for AssignStmt in this example receives the mapping of variables to their nullable status, denoted as Γ (see Section 2.3). The equation begins by retrieving the declaration of the left-hand side of the assignment statement (computed by the compiler), and checks whether the right-hand side is null or not. Depending on the result, the equation updates the status of the left-hand side variable in Γ. Note that in this example, the attribute trFun for the AST

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

node `AssignStmt` has been defined with a body (surrounded by { and }) to specify its behaviour, whereas we have previously declared attributes in a single line without the body. This is because the attribute in this case is more complex, and requires a more detailed definition to capture its functionality accurately. One requirement, to ensure the correct evaluation of results, for defining attributes in JastAdd is that the code within the body must be observationally pure, which means that it should not produce any externally visible side-effects.

3.4 Performance and Precision

We evaluated the performance and precision of IntraJ on a set of 4 well-known open-source Java projects, including: the ANTLR parser generator, the PMD static code analyser, the JFreeChart charting library and the FOP PDF formatter. 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 30K for ANTLR to 100K for FOP. We compared the precision and performance of IntraJ with the RAG-based framework JastAddJ-Intraflow [Söd+13], and the SonarQube static code analyser [Son]. In this section, we provide an overview of the evaluation compared to SonarQube, while the evaluation compared to JastAddJ-Intraflow can be found in Paper 1. Specifically, we compared IntraJ with SonarQube on two distinct analyses: the `DeadAssignment` and `NullPointerException` analysis.

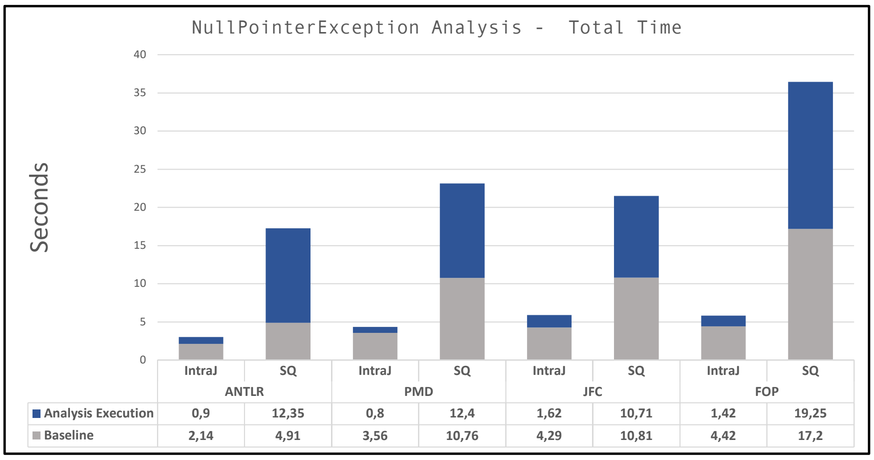


	ANTLR		PMD		JFC		FOP	
	IntraJ	SQ	IntraJ	SQ	IntraJ	SQ	IntraJ	SQ
■ Analysis Execution	0,53	0,24	0,47	0,18	0,75	0,24	0,67	0,34
■ Baseline	2,14	4,91	3,56	10,76	4,29	10,81	4,42	17,2

Figure 11: Total time for the `DeadAssignment` Analysis. This includes time for parsing, checking the absence of errors, running the analysis.

Figure 11 shows the evaluation results for the `DeadAssignment` analysis. The figure reports two distinct measurements: the total time to load each benchmark without executing the analysis, i.e., *Baseline*, and the total time to compute the analysis, i.e., *Analysis Execution*.

Our analysis shows that IntraJ outperforms SonarQube in all cases since the *Baseline* in SonarQube is significantly higher than in IntraJ. However, we can observe, that SonarQube's `DeadAssignment` analysis is faster than IntraJ's. We believe that this is due to the fact that SonarQube computes the CFGs during the *Baseline* phase.

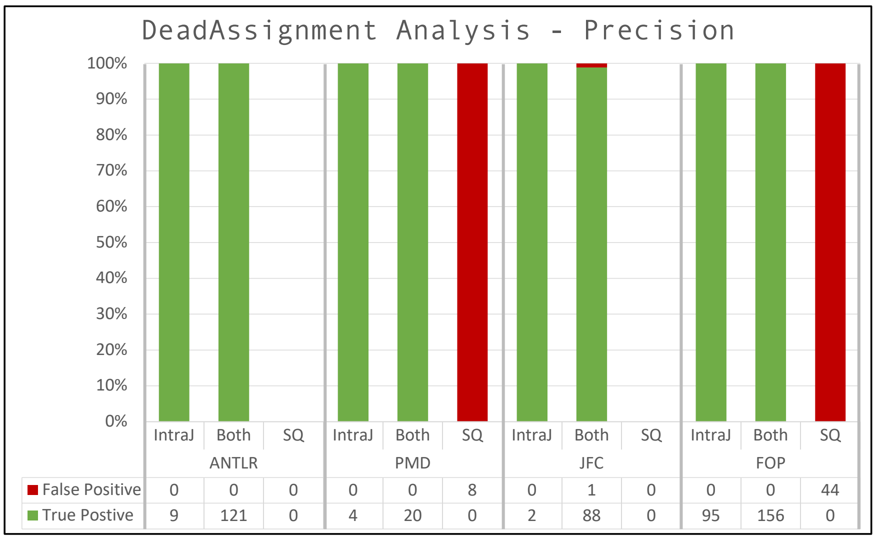


	IntraJ	SQ	IntraJ	SQ	IntraJ	SQ	IntraJ	SQ
	ANTLR		PMD		JFC		FOP	
■ Analysis Execution	0,9	12,35	0,8	12,4	1,62	10,71	1,42	19,25
■ Baseline	2,14	4,91	3,56	10,76	4,29	10,81	4,42	17,2

Figure 12: Total time for the `NullPointerException`. This includes time for parsing, checking the absence of errors, running the analysis.

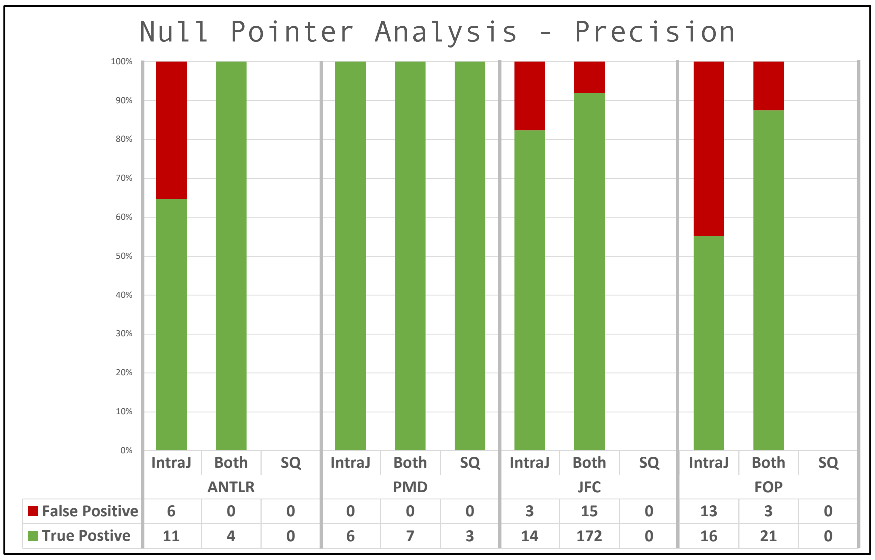
The evaluation results for the `NullPointerException` analysis are presented in Figure 12. Similarly to the `DeadAssignment` analysis, we compared both the total time to process each benchmark and the total time to run the analysis. The results indicate that, in all cases, IntraJ outperforms SonarQube. It can be noticed that, even when excluding the *Baseline* measurements, IntraJ still achieves faster execution times than SonarQube for all benchmarks.

We compared the precision of IntraJ with SonarQube for the `DeadAssignment` and `NullPointerException` analyses. The precision of an analysis is measured as the ratio of true positives and false positives to the total number of reports.



	IntraJ	Both	SQ	IntraJ	Both	SQ	IntraJ	Both	SQ	IntraJ	Both	SQ
		ANTLR			PMD			JFC			FOP	
🟥 False Positive	0	0	0	0	0	8	0	1	0	0	0	44
🟩 True Postive	9	121	0	4	20	0	2	88	0	95	156	0

Figure 13: Precision of the `DeadAssignment` Analysis. We report the ratio of true positives and false positives to the total number of reports.



	IntraJ	Both	SQ	IntraJ	Both	SQ	IntraJ	Both	SQ	IntraJ	Both	SQ
		ANTLR			PMD			JFC			FOP	
🟥 False Positive	6	0	0	0	0	0	3	15	0	13	3	0
🟩 True Postive	11	4	0	6	7	3	14	172	0	16	21	0

Figure 14: Precision of the `NullPointerException` Analysis. We report the ratio of true positives and false positives to the total number of reports.

Figure 13 shows the results for the `DeadAssignment` analysis. It is evident that

IntraJ outperforms SonarQube in terms of precision for all benchmarks. IntraJ is able to identify additional true positives that are not reported by SonarQube. Conversely, SonarQube reports a significant number of false positives, especially in FOP, where it reports 44 false positives.

The results for `NullPointerException` analysis are shown in Figure 14. For this 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.

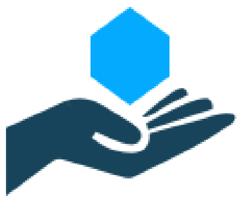
Overall, the results suggest that IntraJ enables practical dataflow analyses, with run-times and precision comparable to state-of-the-art tools.

IntraJ Artifact Evaluation

Artifact evaluation is an essential part of the scientific process as it enables researchers to demonstrate the reproducibility and practicality of their work [Kri13]. It also allows other researchers to build upon and reuse the work in their studies, helping to advance the field as a whole.

In this context, our artifact, IntraJ, was presented at the ROSE[5] Festival 2021, co-located with ICSME2021 and SCAM2021. The artifact was awarded the *Open Research Objects* (ORO) and *Research Objects Reviewed* (ROR) badges. The ORO badge indicates that the artifact is publicly accessible and has been *"placed in an archival repository, with a unique identifier and a link provided"*. The ROR badge highlights that the artifact has been *"very carefully documented and structured, making it suitable for reuse and repurposing in accordance with research community norms and standards"*. This recognition demonstrates the high quality and usefulness of the artifact.

[5]**R**ecognizing and Rewarding **O**pen Science in **SE**, `https://icsme2021.github.io/cfp/AEandROSETrack.html`


(a) Open Research Objects (ORO) badge

(b) Research Objects Reviewed (ROR) badge

Figure 15: Badges awarded to the artifact.

4 IntraTeal: IntraCFG for Teal

As a further demonstration[6] of the applicability of IntraCFG, we developed IntraTeal[7], 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[8] 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 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. The concrete and abstract grammar of Teal-0 are shown in Figure 16. The concrete grammar is used for parsing. The abstract grammar is given using JastAdd syntax, and will be used in examples throughout this section to describe analyses in Teal.

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

```
Program   ::=  ⟨Decl⟩⋆

Decl      ::=  ⟨VarDecl⟩
           |   fun id ( ⟨formals⟩? )
               ⟨optTyped⟩ = ⟨stmt⟩

VarDecl   ::=  var id ⟨optTyped⟩
           |   var id ⟨optTyped⟩ := ⟨Expr⟩;

formals   ::=  id ⟨optTyped⟩
           |   id ⟨optTyped⟩ , ⟨formals⟩

optTyped  ::=  : ⟨Type⟩
           |   ε

Type      ::=  int | string | any
           |   array [ ⟨Type⟩ ]

Block     ::=  { ⟨Stmt⟩ ⋆ }

Expr      ::=  ⟨Expr⟩ ⟨binop⟩ ⟨Expr⟩
           |   not ⟨Expr⟩
           |   ( ⟨Expr⟩ ⟨optTyped⟩ )
           |   ⟨Expr⟩ [ ⟨Expr⟩ ]
           |   id ( ⟨actuals⟩? )
           |   [ ⟨actuals⟩? ]
           |   new ⟨Type⟩ ( ⟨Expr⟩ )
           |   int | string | null
           |   id

actuals   ::=  ⟨Expr⟩
           |   ⟨Expr⟩, ⟨actuals⟩

binop     ::=  + | - | * | / | %
           |   == | !=
           |   < | <= | >= | >
           |   or | and

Stmt      ::=  ⟨VarDecl⟩
           |   ⟨Expr⟩ ;
           |   ⟨Expr⟩ := ⟨Expr⟩ ;
           |   ⟨Block⟩
           |   if ⟨expr⟩ ⟨block⟩ else ⟨block⟩
           |   if ⟨expr⟩ ⟨block⟩
           |   while ⟨expr⟩ ⟨block⟩
           |   return ⟨expr⟩ ;
```

```
Program ::= Decl*;

abstract Decl;
VarDecl : Decl ::= IdDecl [DeclTy:Type]
    [Initializer:Expr];
FunDecl : Decl ::= IdDecl [DeclRetTy:Type]
    Formal:VarDecl* [Body:Stmt];

abstract Expr;
abstract BinExpr : Expr ::=
     Left:Expr Right:Expr;
AddExpr : BinExpr;
SubExpr : BinExpr;
MulExpr : BinExpr;
DivExpr : BinExpr;
ModExpr : BinExpr;
EQExpr : BinExpr;
NEQExpr : BinExpr;
LTExpr : BinExpr;
GTExpr : BinExpr;
LEQExpr : BinExpr;
GEQExpr : BinExpr;
OrExpr : BinExpr;
AndExpr : BinExpr;
CallExpr : Expr ::= IdUse Actual:Expr*;
Null : Expr;
ArrayLiteralExpr : Expr ::= Expr*;
IndexExpr : Expr ::= Base:Expr Index:Expr;
NotExpr : Expr ::= Expr;
TypedExpr : Expr ::= Expr DeclType:Type;
NewExpr : Expr ::= Type Actual:Expr*;
Access : Expr ::= IdUse ;

abstract Constant : Expr;
IntConstant : Constant ::= <Value:Long>;
StringConstant : Constant ::= <Value:String>;

abstract Stmt;
VarDeclStmt : Stmt ::= VarDecl;
ExprStmt : Stmt ::= Expr;
AssignStmt : Stmt ::= LValue:Expr RValue:Expr;
Block : Stmt ::= Stmt*;
IfStmt : Stmt ::= Cond:Expr Then:Stmt Else:Stmt;
WhileStmt : Stmt ::= Cond:Expr Body:Stmt;
ReturnStmt : Stmt ::= Expr;
SkipStmt : Stmt;

IdUse ::= <Identifier>;
IdDecl ::= <Identifier>;
```

Figure 16: The concrete parsing grammar and parts of the abstract JastAdd grammar of the Teal language. Colour coding highlights declarations (**orange**), expressions (**blue**), attribute names (**grey**), and statements (**magenta**).

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 17.

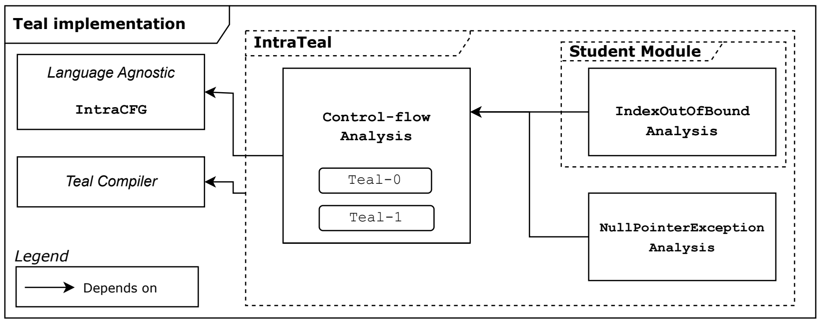


Figure 17: Overall architecture of IntraCFG instantiated for the Teal language.

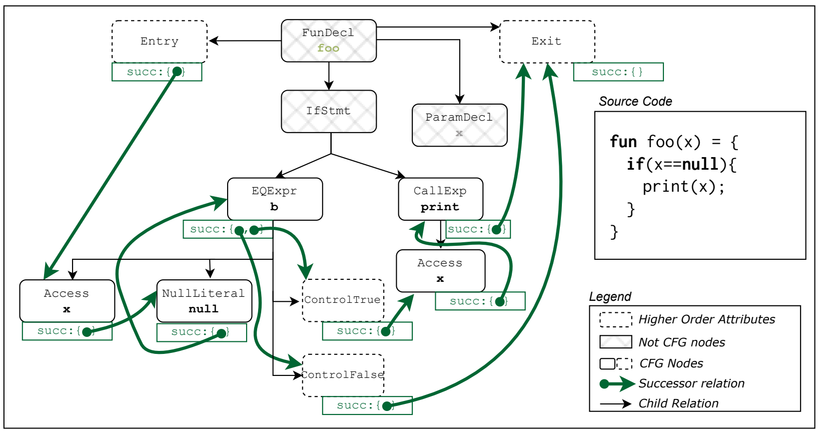
As part of the course, the complete source code of the IntraTeal control-flow analysis was provided to students, along with instructions and guidelines for utilizing the API to implement their analyses. We also made available a reference implementation of the `NullPointerException` analysis. To extend 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.

Another key aspect of IntraTeal and IntraJ implementations is that they construct enhanced control sensitive CFGs, i.e., it understands that the following code, dereferencing `x` inside the body of the if-statement, is safe:

```
if (x != null){
  x.f = 1
}
```

Control sensitive CFGs are a more precise representation of the control flow of a program, as they take into account the different behaviour 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 18 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



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

Figure 18: Example of control sensitivity in IntraTeal.

```
eq ControlTrue.nullnessTransfer(NullDomain lattice) = new
    NullDomain(lattice).join(getCond().getImplicitAssignment());
```

Listing 1: Transfer function for `ControlTrue` HOA.

or false, respectively. The `NullPointerException` analysis can benefit from this more refined CFG. In Listing 1, we show how this enhanced CFG is used to improve the precision of the analysis. The Listing shows the equation for the transfer function of the `ControlTrue` HOA. Specifically, the transfer function for the `ControlTrue` HOA records the effect of the implicit assignment[9] when the condition of the `IfStmt` evaluates to true. In this case, the variable `x` is assigned the value `null` and the `ControlTrue`'s transfer function records this information.

Listing 2: Control sensitivity to improve null pointer analysis.

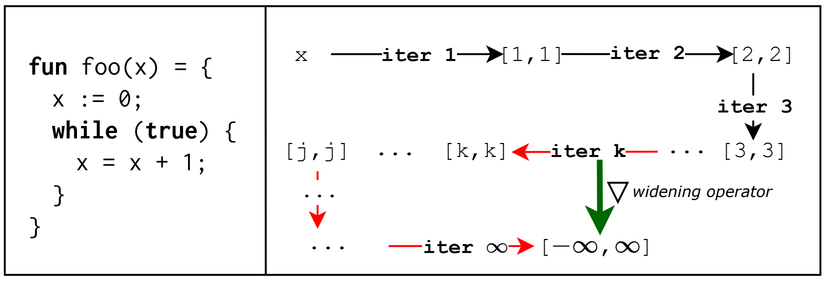
```
if(x!=null){
//x is not null here
}
```

Listing 3: Control sensitivity to improve interval analysis.

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

For the example in Listing 2 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.

[9]Modelled as assertions in [MS18]



```
fun foo(x) = {
  x := 0;
  while (true) {
    x = x + 1;
  }
}
```

Figure 19: Trivial example of non-converging analysis. The red path represents the non-converging evaluation sequence. The analysis keeps changing the value of the variable x without ever reaching a fix-point. The green path represents a sequence converging to a fix-point by using the widening operator.

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 3 the `ControlTrue` and `ControlFalse` are used to keep track of the information that the object x is in the interval [5,6].

The task assigned to the students was to implement an `IndexOutOfBound` analysis on the interval domain. The interval domain, being an infinite domain, poses a significant challenge for dataflow analysis. In particular, the iterative nature of dataflow analysis algorithms, which rely on the computation of a sequence of approximations, may not terminate when applied to an infinite domain. Even if the computation terminates, it can result in an excessive consumption of computational resources. The example in Figure 19 illustrates a non-converging program.

To ensure the termination of the analysis, the students were required to define their own widening operator [BHZ03]. However, JASTADD circular attributes, which are used to implement dataflow analysis, do not natively support widening operators, as they are designed to work only on finite lattices. To overcome this limitation, an ad-hoc solution solution was implemented to trigger the widening operator after a certain number of steps.

This experience highlights the need for native support for widening and narrowing operators in JASTADD, to allow static analysis developers to effectively deal with infinite lattices. In the future, we plan to investigate the implementation of this feature in JASTADD to facilitate the development of static analysis tools.

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 [LDB19] framework. We then describe the integration of IntraJ and IntraTeal with CodeProber [RA+22], 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[10] workshop, held in conjunction with ECOOP 2022, with the title *"Source-Level Dataflow-Based Fixes: Experiences From Using IntraJ and MagpieBridge"*.

This integration process and evaluation are 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 analyses 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 20 illustrates the integration of IntraJ with different IDEs. Server-Analysis is a component that we developed to handle the communication be-

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

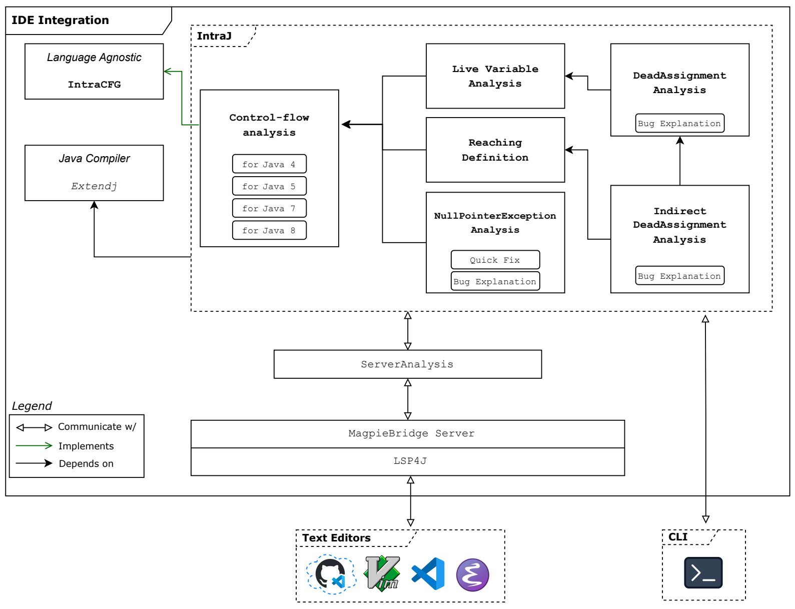
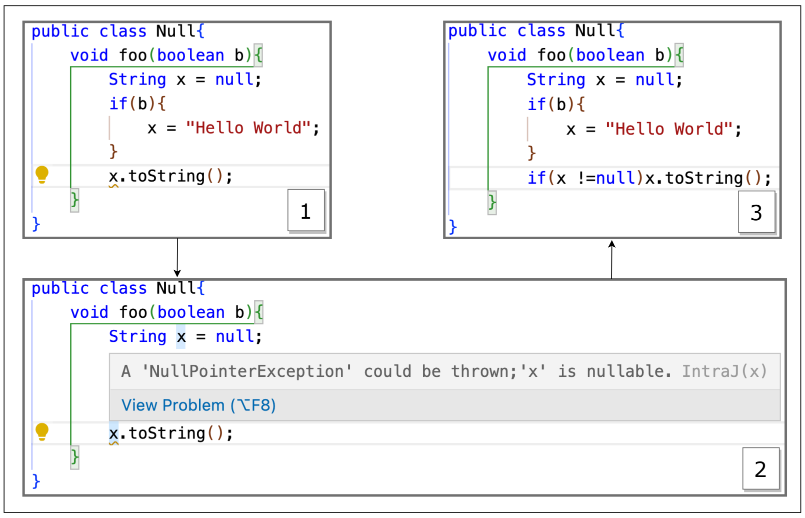


Figure 20: Integration of IntraJ with IDEs through the use of the MagpieBridge framework.

tween 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 21 illustrates an example of interaction between IntraJ and Visual Studio Code. More specifically, 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



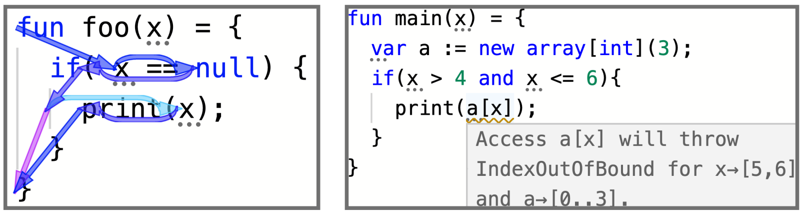
```
public class Null{
    void foo(boolean b){
        String x = null;
        if(b){
            x = "Hello World";
        }
        x.toString();
    }
}
```

```
public class Null{
    void foo(boolean b){
        String x = null;
        A 'NullPointerException' could be thrown;'x' is nullable. IntraJ(x)
        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 21: 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.

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.



```
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 22: Interaction of INTRATEAL in CODEPROBER.

The example in Figure 22 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 queries. In this Section, we give an overview of this work, and more details are available in Paper 2.

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 23, we show the architecture of JFeature.

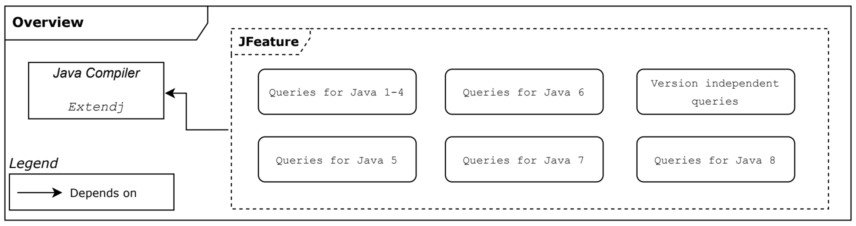


Figure 23: JFeature architecture.

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+10], 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 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 2, 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 Conclusions and Future Work

In this thesis, we have explored the use of RAGs declarative paradigms for intraprocedural control-flow and dataflow analysis in static program analysis. Our main contributions is the development of a new framework for precise construction of source-level control-flow graphs, called IntraCFG. IntraCFG is language-agnostic and is a flexible framework that can be used to construct precise control-flow graphs for a wide range of programming languages. We have demonstrated the effectiveness of IntraCFG through two case studies, namely IntraJ and IntraTeal. IntraJ is an instance of IntraCFG for Java, built upon the ExtendJ compiler. We have demonstrated the potential of IntraJ as a precise tool for detecting complex dataflow analysis, such as NullPointerExceptions and DeadAssignements. We compared its performance and precision with the industrial tool SonarQube. The results showed that IntraJ is more efficient and as precise as SonarQube. Additionally, we have shown the use of IntraTeal for educational purposes, in combination with the visualisation tool CodeProber, resulting in an effective tool for students to learn about dataflow analysis. 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 the future, we plan to investigate using RAGs to support interprocedural analysis in IntraCFG. Interprocedural analysis refers to the process of analysing the interactions and dependencies between different procedures or functions within a program. One of the main challenges in this area is the computation of precise *call graphs*, which are graphs representing the dependencies between procedures and functions. Call graphs are crucial for interprocedural dataflow analysis, as they allow to determine the flow of data between different procedures and functions. Static analysis techniques can be used to compute call graphs, including *Rapid Type Analysis* [BS96] which is fast but imprecise, or *call graph* analysis [VR+10a] in combination with *points-to* [Ste96] analysis which is computationally expensive but more precise. Points-to analysis involves determining the memory locations that a variable may point to. In order to achieve accurate and efficient program analysis within IDEs, we aim to explore a combination of static analysis techniques, including Rapid Type Analysis and Points-to Analysis. Performing a precise analysis typically requires starting from the main entry point of the program. However, in order to run the analysis within IDEs, we plan to start with the methods that are currently being edited.

This presents a significant challenge for the analysis, as it must be able to accurately track data flow from these partial starting points.

Furthermore, we plan to push even further the use of the on-demand evaluation feature in IntraJ. Currently, quick-fixes are computed when the analyses are triggered. We plan to investigate the feasibility of computing quick-fixes on-demand, i.e., when they are requested by the user. The proposed change could help optimize the use of IntraJ in IDEs. Finally, we plan to add native support for widening and narrowing operators in JastAdd's circular attributes, which would enable an easier implementation of analysis over infinite lattices.

References

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

[All70] Frances E Allen. “Control flow analysis”. In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19.

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

[BS96] David F Bacon and Peter F Sweeney. “Fast static analysis of C++ virtual function calls”. In: *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* 1996, pp. 324–341.

[BHZ03] Roberto Bagnara, Patricia Hill, and Enea Zaffanella. “Widening Operators for Powerset Domains”. In: Dec. 2003, pp. 403–433.

[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 Programing, 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.

[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.

[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.

[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, 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: *ACM SIGPLAN Notices* 21.7 (1986), pp. 85–98.

[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.

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

[GDMH12] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. "Reconciling manual and automatic refactoring". In: *2012 34th International Conference on Software Engineering (ICSE).* 2012, pp. 211–221.

[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: *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 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.

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

[Kan+10] Sean Kane et al. “Toyota sudden unintended acceleration”. In: *Safety Research & Strategies* (2010).

[KSS17] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice.* CRC Press, 2017.

[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.

[KT29] Bronislaw Knaster and Alfred Tarski. “A fixed point theorem for monotone mappings”. In: *Fundamenta Mathematicae* 14 (1929), pp. 124–134.

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

[Kri13] Shriram Krishnamurthi. “Artifact Evaluation for Software Conferences”. In: *SIGSOFT Softw. Eng. Notes* 38.3 (2013), 7–10.

[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.

[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.

[LDB19] 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.

[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.

[MYL16b] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. “Programming a Dataflow Analysis in Flix”. In: *Tools for Automatic Program Analysis (TAPAS)* (2016).

[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.

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

[NN99] Flemming Nielson and Hanne Riis Nielson. “Type and effect systems”. In: *Correct System Design: Recent Insights and Advances* (1999), pp. 114–136.

[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.

[RA+22] Anton Risberg Alaküla et al. “Property Probes: Source Code Based Exploration of Program Analysis Results”. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. 2022, pp. 148–160.

[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).

[SL10] Don Shafer and Phillip A. Laplante. “The BP Oil Spill: Could Software be a Culprit?” In: *IT Professional* 12.5 (2010), pp. 6–9.

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

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

[Ste96] Bjarne Steensgaard. “Points-to analysis in almost linear time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 1996, pp. 32–41.

[Tem+10] 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+10a] 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, 214–224.

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

[VSK89] 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

1 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. DOI: 10.1109/SCAM52516.2021.00009.
An artifact, including the source code and evaluation results, was submitted for the paper at the ROSE (**R**ecognizing and Rewarding **O**pen Science in **SE**) festival and awarded with the badges "*Open Research Objects*" and "*Research Objects Reviewed*".
DOI: 10.5281/zenodo.5296618.

2 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+13] 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 [EH07b]. 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.

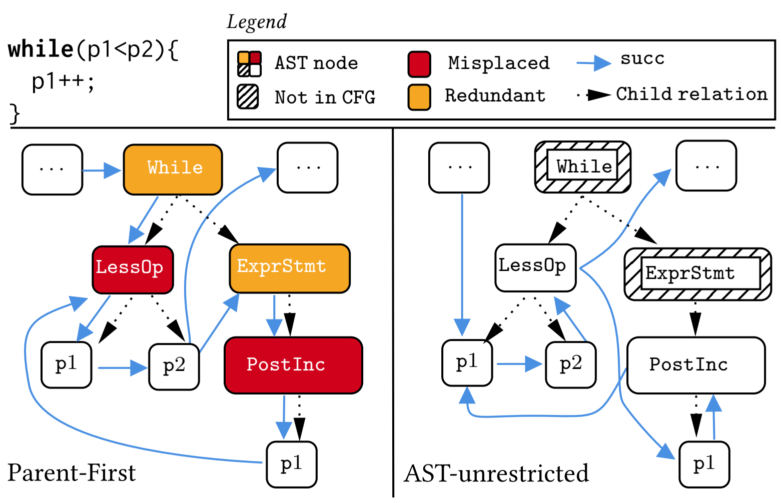


Figure 1: In the Parent-First CFG (left) a parent always precedes its children, resulting in redundant and misplaced nodes. The AST-unrestricted CFG (right) is correct and minimal.

- 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 3) and IntraJ, along with underlying design decisions and implementation details (Section 4), present our client analyses (Section 5) and evaluation (Section 6), discuss related work (Section 7) and conclude (Section 8).

3 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) [VSK89] 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 [MH07]. 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.

3.1 RAG frameworks for control flow

Our work is the second to construct CFGs in a RAG framework, following the earlier JASTADDJ-INTRAFLOW [Söd+13]. 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.

3.2 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 $\circlearrowleft$ to highlight the AST traversal underlying their computation. For the different kinds of attributes, we use the following equations, for attributes x

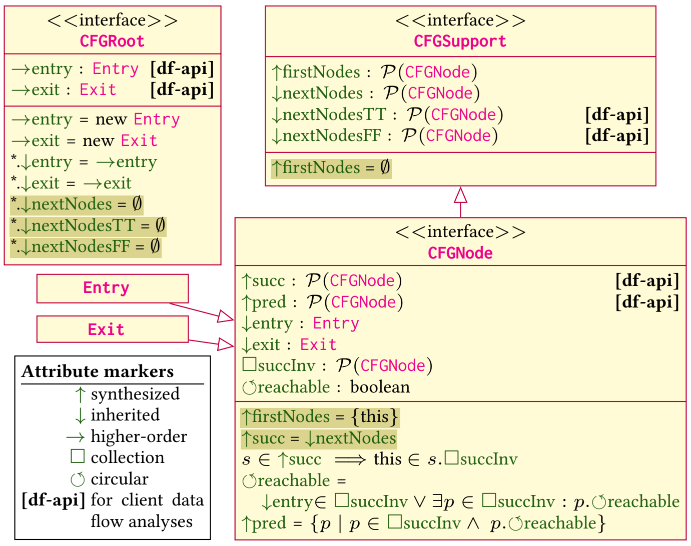


Figure 2: The IntraCFG framework with interfaces CFGRoot, CFGSupport, CFGNode, and synthetic AST types Entry, Exit. Highlighted attribute equations are default equations, intended for overriding.

and expressions e:

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

Inherited attributes: $c.\downarrow\mathrm{x} = e$ gives AST child node c and its descendants access to e through $\downarrow\mathrm{x}$, 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, $*.\downarrow\mathrm{x} = e$.

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

Circular attributes: $\circlearrowleft\mathrm{x} = e$, 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 $P \implies e \in n.\square\mathrm{x}$ to contribute the value of expression e to collection attribute $\square\mathrm{x}$ in node n if P holds. In this paper, all collection attributes are sets.

This pseudocode translates straightforwardly to more verbose JastAdd code

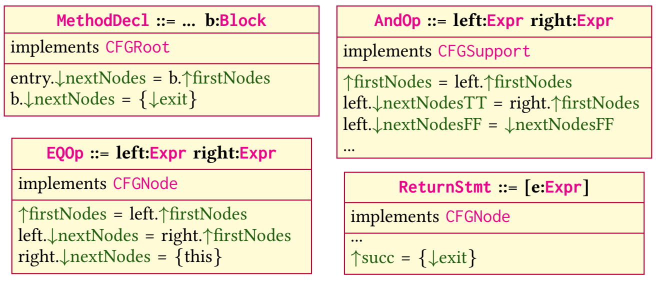

MethodDecl ::= ... b:Block
implements CFGRoot
entry.↓nextNodes = b.↑firstNodes
b.↓nextNodes = {↓exit}
AndOp ::= left:Expr right:Expr
implements CFGSupport
↑firstNodes = left.↑firstNodes
left.↓nextNodesTT = right.↑firstNodes
left.↓nextNodesFF = ↓nextNodesFF
...
EQOp ::= left:Expr right:Expr
implements 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 5.

3.3 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:

↓nextNodesTT ∪ ↓nextNodesFF = ↓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 4.1).

3.4 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;
}
```

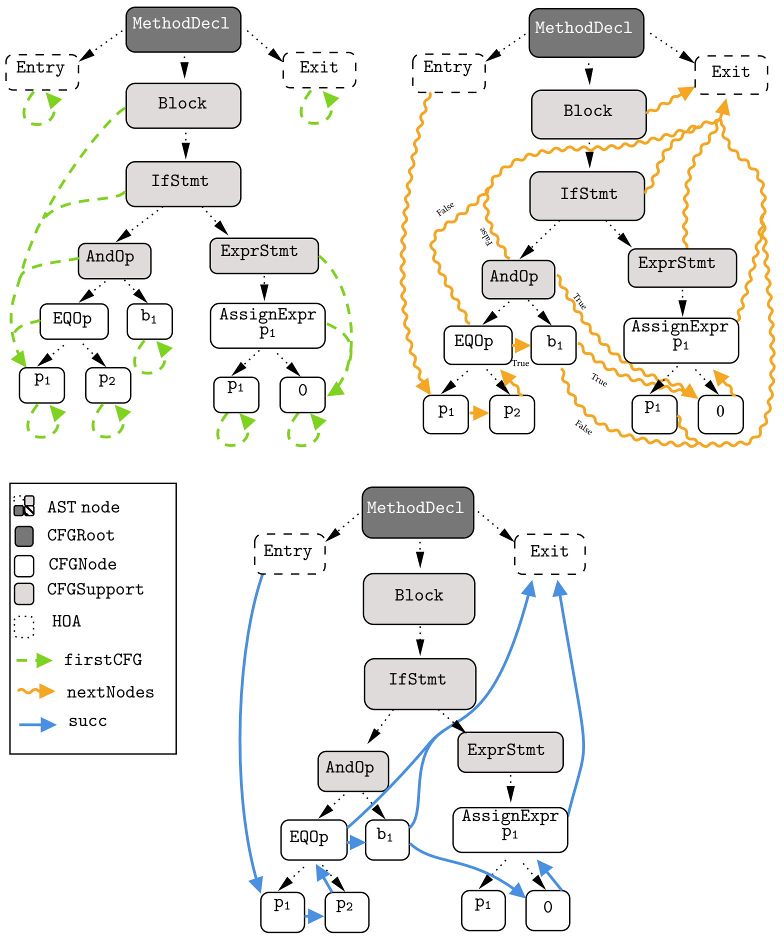


Figure 4: Visualization of the attributes ↑firstNodes, ↓nextNodes and ↑succ. For boolean expressions (AndOp and EQOp), the subsets ↓nextNodesTT and ↓nextNodesFF are shown instead of ↓nextNodes, marked by True and False, respectively.

4 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.

4.1 Statements and Expressions

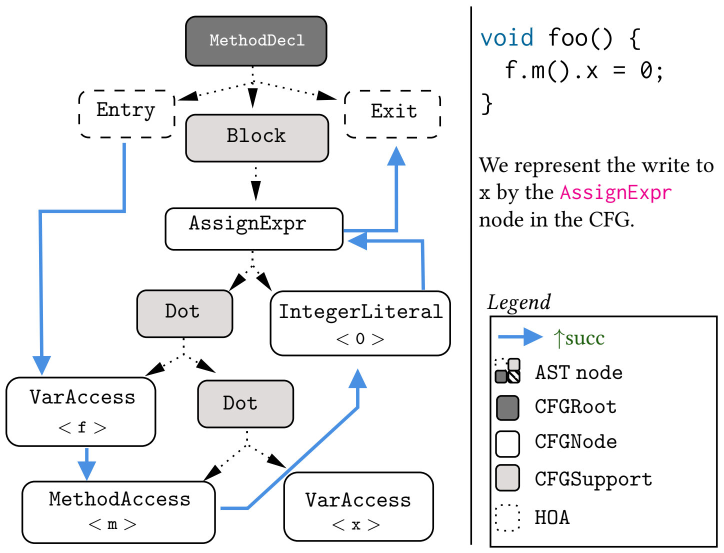
When a language implementer specialises IntraCFG for a given language, they must decide which AST nodes should be part of the CFG, i.e., mix in (implement) the `CFGNode` interface. As a general design principle, we included AST nodes that correspond to a single action at runtime. This includes operations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

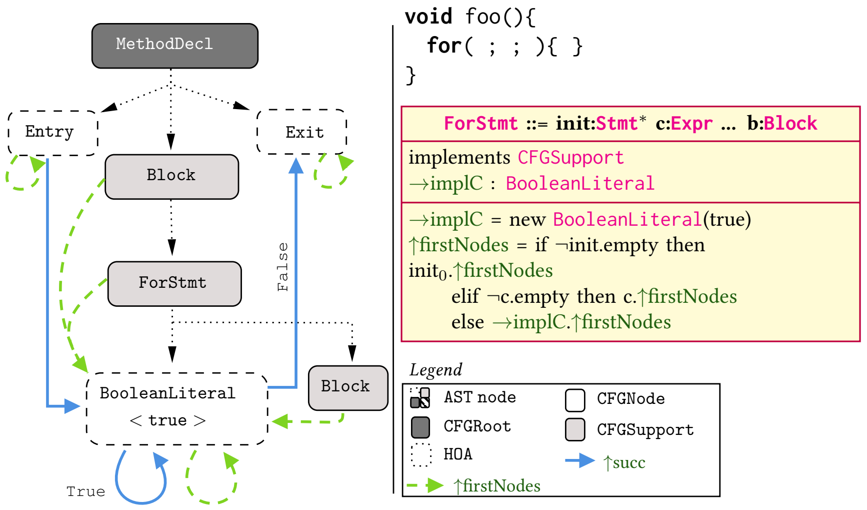
We represent the write to x by the AssignExpr node in the CFG.

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
init₀.↑firstNodes
      elif ¬c.empty then c.↑firstNodes
      else →implC.↑firstNodes
```

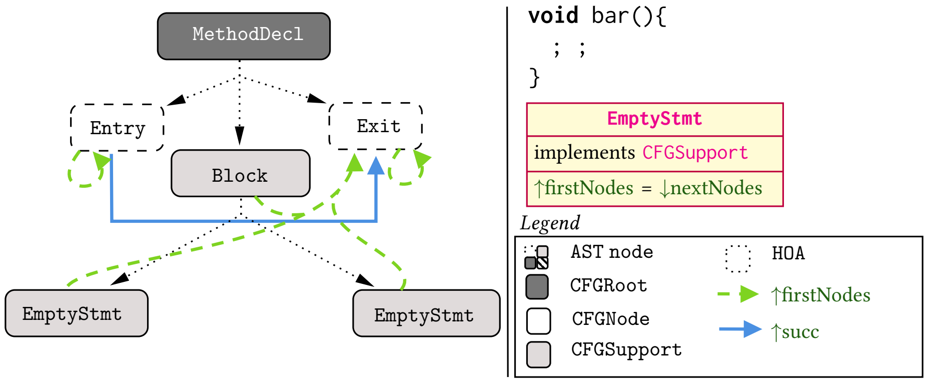
Figure 6: CFG for method with empty *ForStmt*. The HOA →implC reifies the implicit *true* condition.

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

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

4.2 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,



```
void bar(){
  ; ;
}
```

Figure 7: The CFG can entirely skip AST nodes.

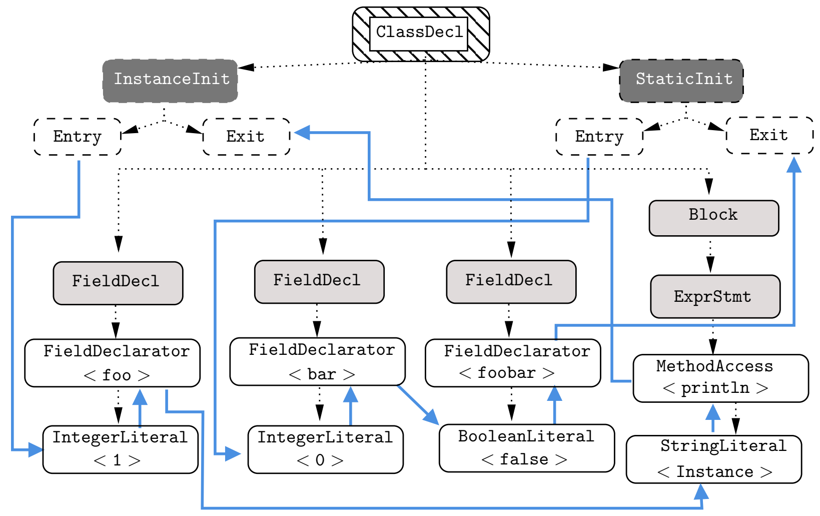
another static field `foobar`, and by an instance initialiser block printing the string `"Instance"`.

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.

4.3 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
}
```

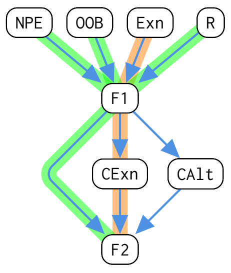


Figure 9: Complex exception flow in a conservative CFG. Only the flow paths in green and orange are realisable.

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.

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 6 shows some of the practical strengths and weaknesses of our heuristic.

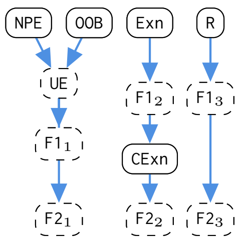


Figure 10: Path-sensitive variant of the CFG from Figure 9, used in INTRAJ.

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.

5 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:

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

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.

5.1 Null Pointer Exception Analysis

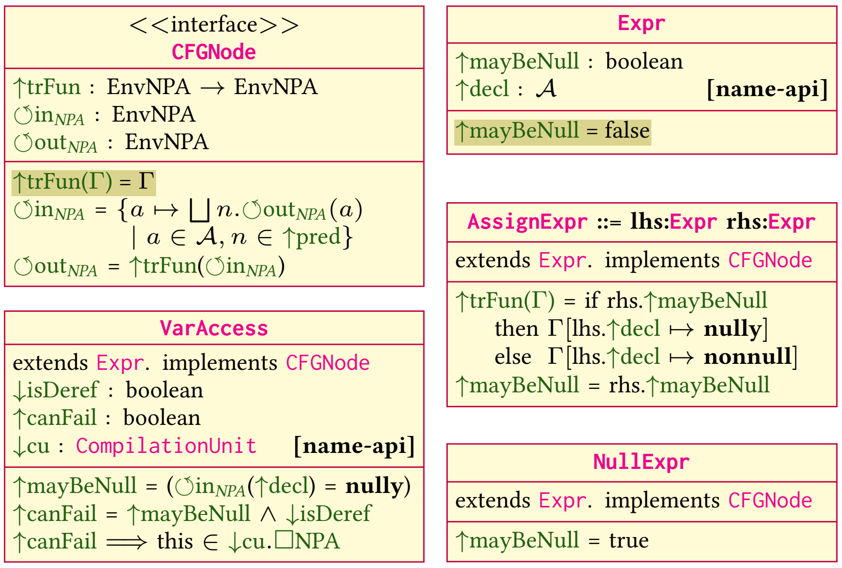
In our simplified language, a field access `x.f` fails (in Java: throws a Null Pointer Exception) if `x` is `null`. Null Pointer Exception Analysis (NPA) detects whether a given field dereference *may* fail (e.g. in the SonarQube NPA variant) or *must* fail (e.g. in the Eclipse JDT NPA variant) and can 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 $\longrightarrow$ 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$_{NPA}$)

Expr

$\uparrow$mayBeNull : boolean
$\uparrow$decl : $\mathcal{A}$ **[name-api]**

$\uparrow$mayBeNull = false

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

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.

5.2 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+13]: Figure 12 shows our computation where circular attributes ↻in$_{LVA}$ and ↻out$_{LVA}$ represent variables live before/after a `CFGNode`. Here, ↻out$_{LVA}$ reads from ↑succ nodes, since we are implementing an on-demand backward analysis. `VarAccess` and `AssignExpr` override ↑use and ↑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.

5.3 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.

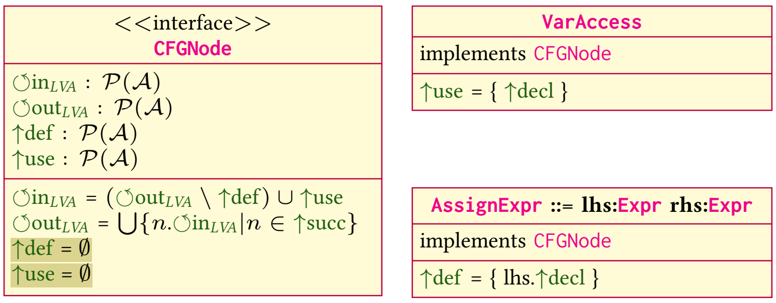

<<interface>>
CFGNode
↻in$_{LVA}$: $\mathcal{P}(\mathcal{A})$
↻out$_{LVA}$: $\mathcal{P}(\mathcal{A})$
↑def : $\mathcal{P}(\mathcal{A})$
↑use : $\mathcal{P}(\mathcal{A})$
↻in$_{LVA}$ = (↻out$_{LVA}$ \ ↑def) ∪ ↑use
↻out$_{LVA}$ = $\bigcup\{n.$↻in$_{LVA} | n \in$ ↑succ$\}$
↑def = ∅
↑use = ∅
VarAccess
implements CFGNode
↑use = { ↑decl }
AssignExpr ::= lhs:Expr rhs:Expr
implements CFGNode
↑def = { lhs.↑decl }

Figure 12: Partial implementation of our LVA.

6 Evaluation and Results

We demonstrate the utility of IntraCFG and IntraJ[3] by evaluating the client analyses that we describe in Section 5 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+13], 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%.

6.1 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 5.2) with minimal filtering, while SQ additionally applies the default value filtering heuristic from Section 5.3. 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 5.3). 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 4.3). 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

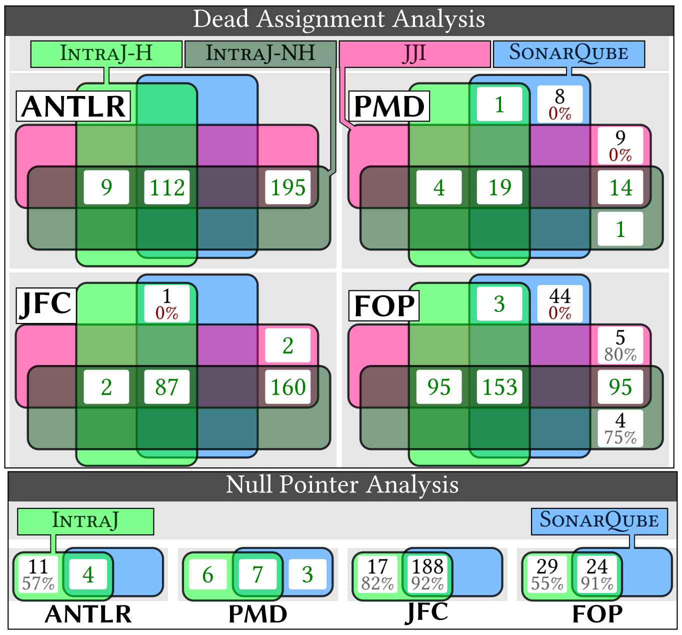


Figure 13: Venn diagram: number of reports shared across checkers, and percentage of true positives (unless 100%).

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

6.2 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 $\%_{JJI}$ and $\%_{SQ}$ 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)		
	IntraJ	JJI	SQ
ANTLR	2.14±0.01	1.34±0.01	4.91±0.05
	IntraJ	JJI	SQ
PMD	3.56±0.01	2.34±0.02	10.76±0.09
	IntraJ	JJI	SQ
JFC	4.29±0.01	3.14±0.02	10.81±0.11
	IntraJ	JJI	SQ
FOP	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	-	-

7 Related Work

Our work is most similar to JASTADDJ-INTRAFLOW [Söd+13], 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+13] 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+10] 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 [MYL16a]. 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+10b], 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.

8 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 Programing, 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.

[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 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.

[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.

[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.

[MH07] 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+13] 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+10b] 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+10] 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.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. “Higher Order Attribute Grammars”. In: *SIGPLAN Not.* 24.7 (1989), 131–145.

JFeature: Know Your Corpus

1 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.

2 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

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. 236–241s.* Limassol, Cyprus, 2022. DOI: 10.1109/SCAM55253.2022.00033.
An artifact, including the source code and evaluation results, was submitted for the paper at the ROSE (**R**ecognizing and Rewarding **O**pen Science in **SE**) festival and awarded with the badges "*Open Research Objects*".
DOI: 10.5281/zenodo.7053768.

are. For research on software tools, the dominant methodology for demonstrating effectiveness is to apply these tool 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+10], 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 [EH07b] 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 [EH07b] 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 3 introduces JFeature and discusses the design decisions that underpin the tool. Section 4 shows the results of applying JFeature to the four corpora. Section 5 illustrates how JFeature can be extended to extract new features, taking advantage of properties in the underlying Java compiler. Section 6 outlines future applications of JFeature. Section 7 discusses related work, and Section 8 summarizes our conclusions.

3 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.

3.1 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.

3.2 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).

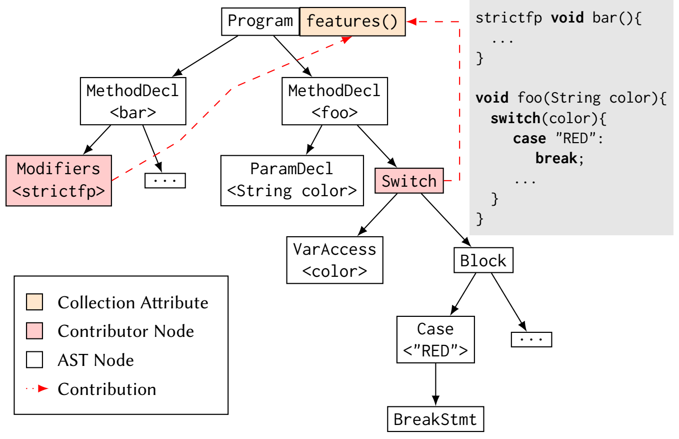
STRING IN SWITCH is a semantic feature in that it depends on the type of the switch expression. It cannot be identified with simple local AST queries or

[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.

4 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.

4.1 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.

4.2 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,

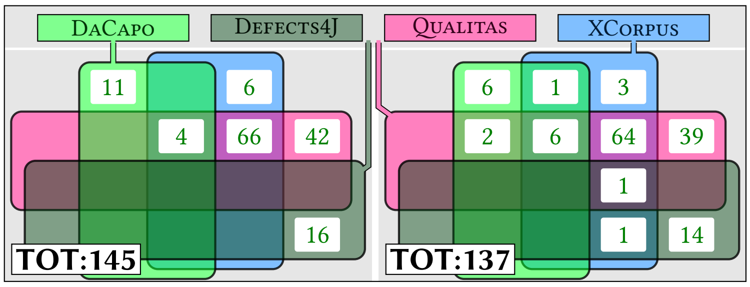


Figure 2: Project overlap. In the left diagram, two projects with the same name but different versions are counted as distinct—the diagram to the right shows overlap when versions are disregarded.

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.

5 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.

6 Use cases for JFeature

We already discussed two possible use cases for JFeature: corpus evaluation (Section 4), and extending JFeature to identify specific features (Section 5). In this section, we discuss two additional use cases: longitudinal studies and project mining.

6.1 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.

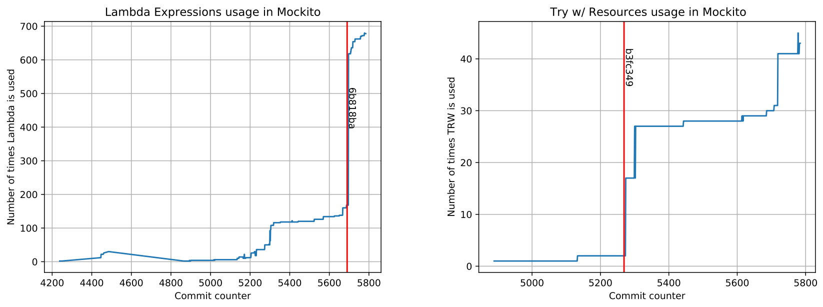


Figure 3: Usage of Lambda Expressions and Try With Resources in Mockito over time.

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.

6.2 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.

7 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* [LDB19], to infer and download libraries automatically. Currently, JavaProjectService infers the dependencies for projects using *Gradle* or *Maven* as build system.

8 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 Programing, 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.

[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 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.

[LDB19] 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+10] 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.