Noname manuscript No.
(will be inserted by the editor)

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

Idriss Riouak · Niklas Fors · Görel Hedin · Christoph Reichenbach

Received: date / Accepted: date

Abstract Static analysis tools play a crucial role in software development by detecting bugs and vulnerabilities. However, running these tools separately from the code editing process often causes developers to switch contexts, which can reduce productivity. Previous work has shown how Reference Attribute Grammars (RAGs) can be used for declarative implementation of competitive tooling for intraprocedural control-flow and dataflow analysis of Java source code, embodied in the tool IntraJ. In this paper, we demonstrate how IntraJ can be leveraged to provide interactive analysis results directly in the editor, similar to compile-time error detection, relying on automatic on-demand evaluation of RAGs. We discuss the architecture of IntraJ, and demonstrate how it can be integrated into the development process in three different ways: in the command line, in an editor integration based on the Language Server Protocol, and in an integration with the debugging tool CodeProber. We showcase the extensibility of IntraJ by illustrating how new client analyses and language constructs can be added to the framework through RAG specifications. Finally, we evaluate the interactive performance of IntraJ on a set of real-world Java benchmarks, demonstrating that IntraJ can provide interactive feedback to developers, achieving a response time of under 0.1 seconds for most compilation units.

Keywords Static Program Analysis, Reference Attribute Grammars, Control Flow Analysis, Dataflow Analysis, Bug Detection, Interactive Bug Detection

Idriss Riouak · Niklas Fors · Görel Hedin · Christoph Reichenbach
Computer Science Department, Lund University, Sweden
E-mail: {idriss.riouak, niklas.fors, gorel.hedin, christoph.reichenbach}@cs.lth.se

1 Introduction

Detecting software bugs early in the development process is crucial. Early detection not only significantly reduces costs but can also prevent severe consequences, such as system failures, security breaches, and financial losses [35].

As a result, software development is increasingly relying on static analysis tools to identify defects and vulnerabilities. Tools like *Infer* [9], *PMD* [8], and *SpotBugs* (successor of FindBugs [3]), are becoming integral parts of the development process, providing automated analysis of code quality and security issues. These tools are typically run as part of automated pipeline enabling the detection of post-editing issues [34]. While this approach is effective, studies have shown that developers strongly prefer to view analysis results directly in the IDE, where issues can be addressed seamlessly within their workflow [7].

Detecting and displaying static analysis issues directly in the editor provides interactive feedback similar to compile-time error detection, which enhances usability by reducing context switching and helping maintain the developer's focus and productivity. Ideally, response times should be within 0.1 seconds, to ensure that a system feels instantaneous to the user [29]. While this level of interactivity cannot always be achieved, the challenge of delivering consistently low-latency, interactive static analysis remains largely unaddressed.

In this paper we revisit our previous work on IntraJ [33], a precise and efficient framework for intraprocedural control-flow and dataflow analysis of Java source code. Here, we demonstrate how IntraJ can be leveraged to provide interactive analysis results directly in the editor, similar to compile-time error detection. We describe the architecture of IntraJ and demonstrate its extensibility and efficiency for interactive analyses, highlighting its modularity and responsiveness. IntraJ achieves low latency through two key approaches: *on-demand evaluation*, computing only the necessary parts of the analysis relevant to the current editing context, and *source-level analysis*, performing analysis directly on the abstract syntax tree rather than bytecode, avoiding the overhead of bytecode generation. This approach minimizes computational resources and ensures efficient and responsive analysis.

IntraJ is built using Reference Attribute Grammars (RAGs) [14], a high-level declarative formalism that inherently supports on-demand evaluation. Our framework is implemented as an extension of the ExtendJ [11] Java compiler, which already uses RAGs for name binding and type analysis. The IntraJ API includes methods for traversing the control-flow graph and computing dataflow analyses, enabling the development of custom analyses using RAGs.

IntraJ offers an extensible API, allowing developers to integrate additional analyses as RAG specifications. These extensions can leverage both IntraJ and ExtendJ APIs, benefiting from automatic on-demand evaluation.

We showcase IntraJ across multiple development environments, including a command line tool, an editor integrated with the Language Server Protocol, and the debugging tool CodeProber [1]. Our previous work [33] showed that IntraJ achieves precision comparable to industrial-strength tools like *SonarQube* and has efficient performance on whole program analyses. In this paper, we evaluate the performance of IntraJ in interactive environments. We present results that highlight its ability to deliver real-time analysis with no noticeable latency, even in large codebases. Our results show that for most projects, IntraJ can analyze 99% of the compilation units (one at a time) in less than 0.1 seconds. This demonstrates IntraJ's suitability for integration into modern development workflows where immediate feedback is crucial.

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

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

We then discuss limitations (Section 7) and related work (Section 8). Finally, we present our conclusions and outline future work (Section 9).

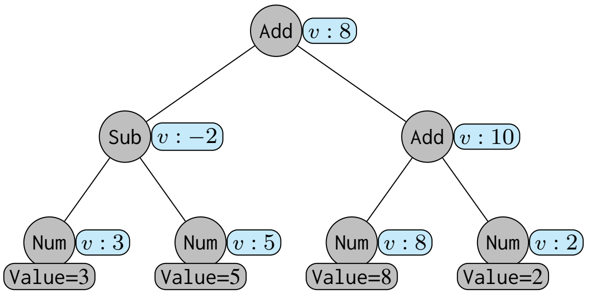


Fig. 1: Decorated AST representing the expression `(3-5) + (8+2)`. Attribute v is the value of the expression.

Listing 1: Abstract grammar for the expression language.

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

Listing 2: Specification of the attribute v.

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

2 Background

This section introduces Reference Attribute Grammars for defining language semantics and monotone frameworks for reasoning about program dataflow properties. We then illustrate how RAGs can express monotone frameworks.

2.1 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) [14] are a formalism for specifying how programming languages should be evaluated and analyzed. RAGs represent programs as abstract syntax trees (ASTs) decorated with computed properties called *attributes*. RAGs extend Knuth's Attribute Grammars [20] by allowing attributes to reference other AST nodes. This extension allows RAGs to define relations between nodes in the abstract syntax tree and to superimpose graphs over the AST, including name bindings, type hierarchies, and control-flow graphs.

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

Listing 1 shows the abstract grammar for this language. To define the abstract grammar and attributes, we use the meta-compilation system JastAdd [16]. Given a RAG specification, JastAdd generates Java code for the AST classes and the attribute evaluation methods. Thus, the abstract grammar in the example above defines four Java classes to represent the different program elements.

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

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

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

This example uses only *synthesized* attributes (indicated by the keyword `syn`), meaning that their defining equations are evaluated in the context of the node to which the attribute belongs, analogously to Java methods. JastAdd supports other kinds of attributes beyond synthesized ones, such as *inherited attributes*, which are evaluated in the context of the parent node and are used for providing nodes with contextual information. This allows passing information downwards in the AST, which IntraJ uses heavily when constructing the control-flow graph. Attributes may transitively depend on their own values, as long as they are explicitly declared to be *circular attributes* [32,27,13,17]. *Circular attributes* allow computing fixed points, which is an essential part of dataflow analysis.

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

2.2 Monotone Frameworks

Many interesting program properties depend on the order in which different parts of the program execute, and on how the contents of the program's variables change over time. To answer questions about e.g. redundant computations or the variables' contents and liveness, modern production compilers and many software tools rely on *monotone frameworks* [19,18], a unifying theoretical approach that enables analyses such as *live variables*, *available expressions*, or *reaching definitions*.

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

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

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

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

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

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

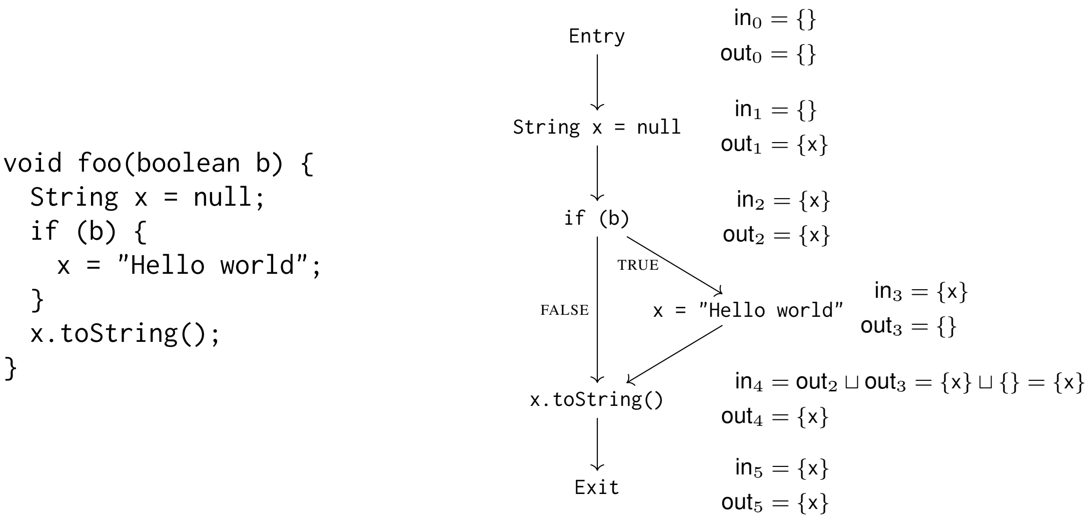
since this CFG node assigns `null` to variable `x`. If there is a CFG edge from node i to node k, propagate the out_i to be in_k, as long as i is the only predecessor of k. If node k has multiple predecessors, we must reconcile (or *join*) the incoming information from all predecessors, as for in_4 in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

The general rule for computing in_i and out_i is thus:

$$\mathsf{in}_i = \bigsqcup_{p \in \mathrm{pred}(i)} \mathsf{out}_p \qquad (1)$$

$$\mathsf{out}_i = f_i(\mathsf{in}_i) \qquad (2)$$

This kind of analysis is called a *forward analysis* because it propagates information from the `Entry` node to the `Exit`



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

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

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

node, but monotone frameworks also support *backward analyses* that traverse the CFG in the opposite direction (e.g., for liveness analysis).

If the source program contains a loop, the CFGs may be circular, which means that the results of out_n for some n may flow back into in_n for the same n, directly or indirectly. In these cases we must compute a fixed point, iterating our computation until none of the in_i or out_i change. Monotone frameworks define sufficient conditions over the transfer functions and the join operator to ensure that such a fixed point always exists, essentially by requiring that we never discard information and that the information for each CFG node reaches a saturation point after a finite number of updates [30].

To express monotone frameworks in RAGs, we can describe transfer functions and the join operator directly as attribute equations over in and out sets. Since these attributes will generally recursively depend on themselves, we express them as circular attributes, which automatically support efficient on-demand fixed point computation [27,31]. We will illustrate such attribute encoding in the next subsection.

2.3 Monotone Frameworks as RAGs

RAGs provide a natural way to express monotone frameworks by using attributes to represent key concepts such as transfer functions and information flow between CFG nodes. To illustrate, we use the null-pointer analysis example in Figure 2, which we express using attributes in JastAdd.

The analysis is defined for each CFGNode, an interface implemented by all nodes that appear in the CFG. Additional details about the CFGNode interface and its implementations are provided in the next section.

Using this approach, we implement a simple null-pointer analysis focusing on two key attributes: inNPA and outNPA, which represent the dataflow information entering and leaving each CFG node, respectively. To clearly distinguish attributes specific to the null-pointer analysis, we append NPA to attribute names.

Transfer Function. With RAGs, the transfer function can be modeled using a synthesized attribute (using the keyword **syn**). This attribute specifies how each CFGNode processes the incoming information and transforms it based on the node's local context:

Listing 3: Definition of the transferNPA synthesized attribute for Null Pointer Analysis. The default implementation propagates the input nullability information, while specialized equations for assignment nodes (Assign), update the nullability state of variables.

```
syn NullDomain CFGNode.transferNPA(NullDomain domain) {
  return domain;
}

eq Assign.transferNPA(NullDomain domain) {
  Variable decl = getDeclaration();
  if (canRightHandSideBeNull()) {
    domain.put(decl, Nullness.MAYBENULL);
  } else {
    domain.put(decl, Nullness.NOTNULL);
  }
  return domain;
}
```

The default implementation of the transfer function, i.e., transferNPA, for the CFGNode interface simply propagates the input NullDomain, leaving it unchanged. NullDomain is a map of variables to nullness information, which is either

MAYBENULL, NOTNULL or BOTTOM. By using the keyword **eq**, we override the default definition of the attribute for specific AST nodes. For example, the equation for Assign nodes modifies the nullability information in NullDomain based on the value of canRightHandSideBeNull(). This mechanism allows specialized behavior for specific AST node types, ensuring that the transfer function accurately reflects the semantics of each CFG node in the analysis.

Fixed-Point Computation. Circular attributes allow defining dependencies that require fixed-point computation [32,26], essential for analyses over cyclic CFGs, e.g., programs with loop constructs. Listing 4 shows how the equations 1-2 are implemented as circular attributes in the RAG framework.

Listing 4: Definition of circular attributes inNPA and outNPA for Null Pointer Analysis (NPA).

```
syn NullDomain CFGNode.inNPA() circular[new NullDomain()] {
  NullDomain res = new NullDomain();
  for (CFGNode p: pred()) {
    res.join(p.outNPA());
  }
  return res;
}

syn NullDomain CFGNode.outNPA() circular[new NullDomain()] {
  return transferNPA(new NullDomain(inNPA()));
}
```

The inNPA attribute joins the outgoing information (outNPA) from all the predecessor nodes using a join operation. The attribute outNPA applies the transfer function on the incoming information (inNPA). A circular attribute also needs a bottom value, which in this case is an empty variable map (new NullDomain()). These circular definitions are evaluated iteratively until a fixed point is reached, ensuring that the analysis converges to a stable result.

Error Detection. To detect potential null-pointer dereference errors, we define the synthesized attribute isNullable, which checks whether a variable is dereferenced and appears as null in the abstract domain. This criterion ensures precise detection of possible errors, as illustrated by the following synthesized attribute:

```
syn boolean VarAccess.isNullable() {
  Nullness n = inNPA().get(varDecl());
  return isDereferenceOperation() && n == Nullness.MAYBENULL;
}
```

3 IntraJ Architecture

IntraJ is a framework for the construction of CFGs for Java 11[1]. It is entirely built using RAGs and offers an exception-sensitive control-flow analysis, considering both checked and unchecked exceptions. On top of IntraJ, we have implemented a number of dataflow analyses, such as *Live Variable analysis*, *Reaching Definition analysis*, and a *Null Pointer Dereference analysis*.

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

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

3.1 The ExtendJ Compiler

ExtendJ is an open-source extensible Java compiler implemented using RAGs. It currently supports Java 4–11, including parsing, compile-time checking, and bytecode generation. ExtendJ defines the node types of a Java AST, such as different kinds of declarations, statements, and expressions. It also defines a number of RAG attributes for these node types, like name bindings, types, compile-time errors, and bytecode. ExtendJ offers thousands of predefined attributes available as an API for the programmer to use when building analyses or transformations. As for any RAG-based system, the attributes are computed on demand, and the computation is driven by the attribute API[2].

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

3.2 The IntraCFG component

IntraJ is an instance of IntraCFG, a language-agnostic RAG component for CFG construction. IntraCFG introduces language-independent abstractions for CFG node specification through a set of *node type interfaces*, which are analogous to Java interfaces but include overridable default attribute equations. Language developers can implement IntraCFG by adapting these interfaces to the AST nodes of their language, thus benefiting from a flexible CFG construction mechanism that adjusts to the syntactic and semantic requirements of a given language [33].

[1] The complete implementation of IntraJ is available on GitHub at https://github.com/lu-cs-sde/IntraJ.

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

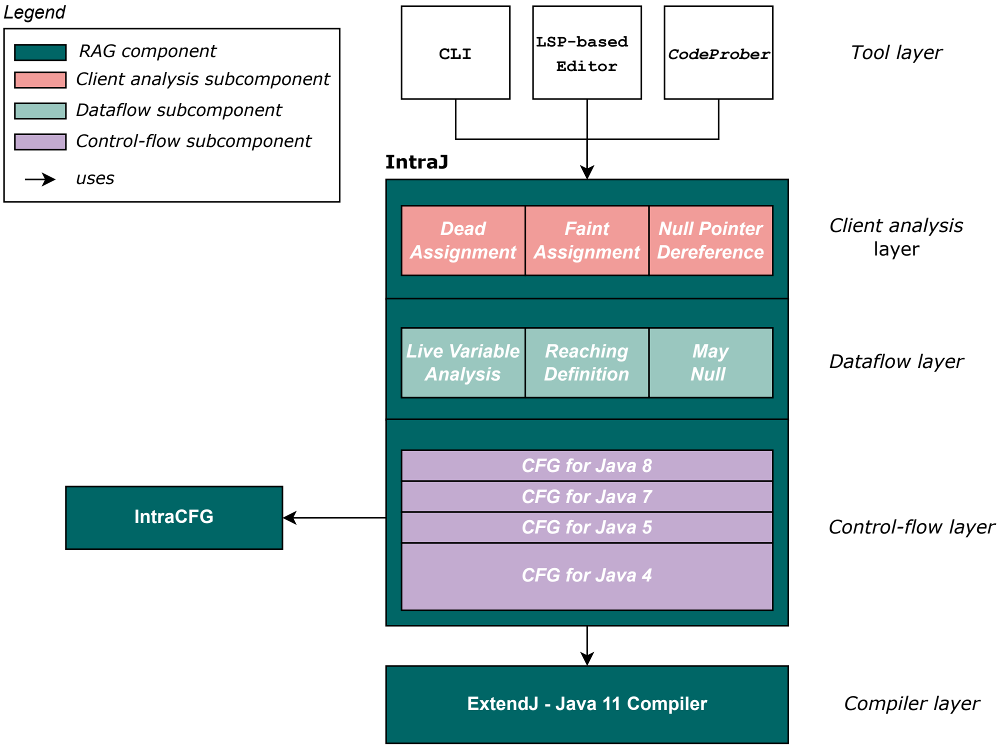


Fig. 3: Layered architecture of INTRAJ. Each INTRAJ layer builds upon the attributes of the underlying layers.

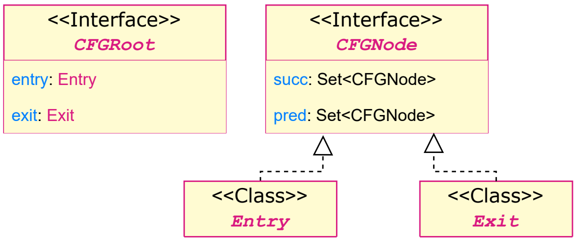


Fig. 4: Main API of INTRACFG.

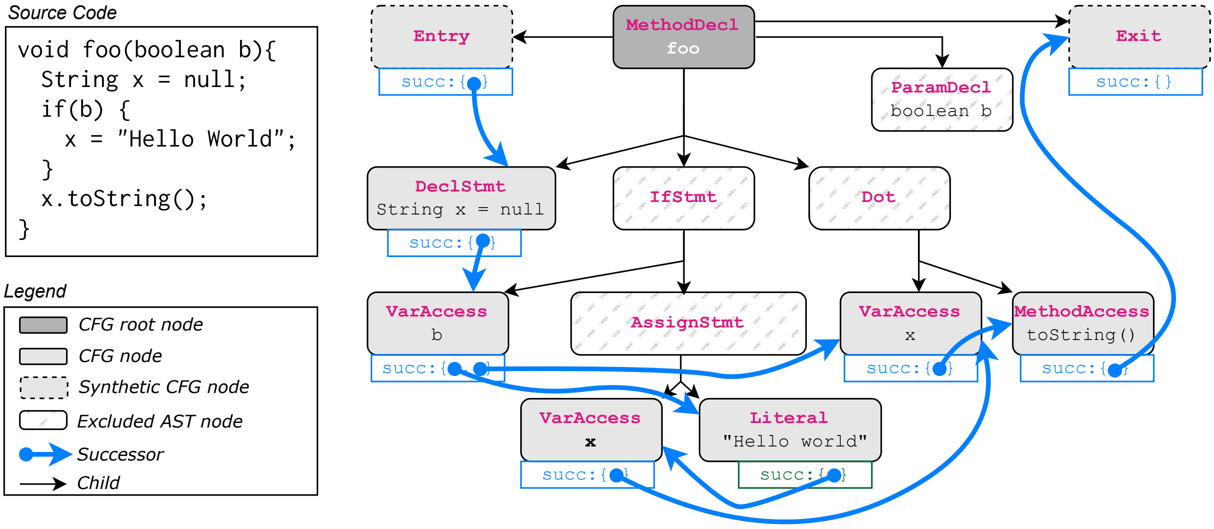
Figure 4 shows the main API of INTRACFG in the form of interfaces and classes. The `CFGRoot` interface is intended for AST node types that represent subroutines, e.g., method and constructor declarations in Java. Each AST node type that implements the `CFGRoot` interface is automatically extended with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

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

3.3 INTRAJ Control-Flow Analysis

To define control-flow graphs for Java, INTRAJ adds the INTRACFG interfaces to selected EXTENDJ node types, and adds and overrides attributes to define the detailed control flow. For INTRAJ, we have chosen to construct CFGs at the expression level. While this design does not increase the precision of the analysis itself, it provides finer-grained control over the flow of execution, allowing the analysis to consider individual expressions. This approach may require the developer to define transfer functions for more AST node types, ensuring that each expression within a block is traversed explicitly, rather than treating entire basic blocks as single units.

However, this additional effort can often be reduced by using node type interfaces, which allow shared behavior to be defined across multiple AST node types. For example, as shown in Listing 3, the `transferNPA` transfer function is defined for the `Assign` interface. This single definition applies uniformly to all AST node types that implement `Assign`, including `VariableDeclarator`, `AssignExpr`, and `ImplicitAssignment`. This approach significantly reduces redundancy, as it eliminates the need to define the transfer function separately for each implementing node type.



Source Code

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

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

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

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

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

The IntraJ CFG specification is structured into subcomponents, as shown in Figure 3, one for each version of Java[3]. This matches a similar subcomponent structure inside ExtendJ, allowing Java compilers to be built for different language versions. If support for newer Java versions, such as Java 12, were added to ExtendJ, only the subcomponent for Java 12 would need to be implemented, while the existing components could be reused.

A more detailed description on the construction of the IntraJ CFGs can be found in [33].

3.4 Dataflow and Client Analyses

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

The analyses in the client layer make use of the general dataflow analyses to interpret the results in a way suitable for

[3] No subcomponent is needed for Java 6, Java 9, Java 10, or Java 11, as these versions did not introduce any new language constructs or semantics affecting control flow.

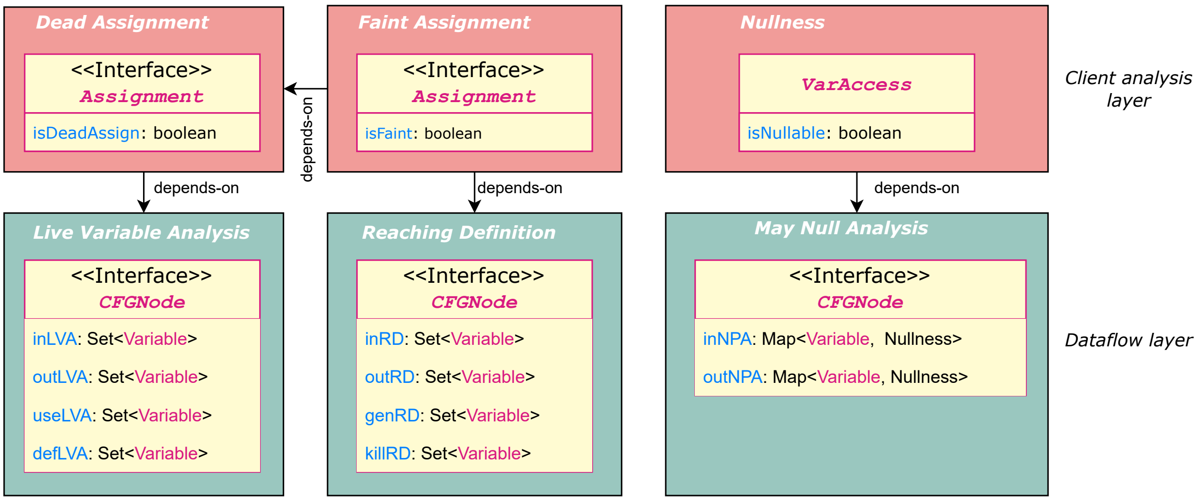


Fig. 6: The APIs exposed by the INTRAJ's dataflow and client analyses.

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

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

The faint assignment analysis [30] is a client analysis that demands the computation of dead assignment analysis to find assignments that are not dead themselves, but are indirectly dead, i.e., used only by dead assignments or other faint assignments.

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

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

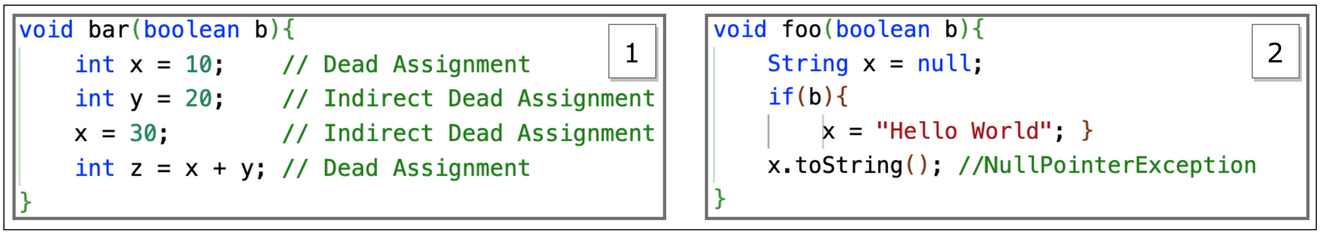
3.5 Demand-Driven Analyses

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

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

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

To compute the `outNPA` of a node, the `inNPA` is combined with a transfer function that may use attributes defined by EXTENDJ, such as the `decl` attribute linking a `VarAccess` to its declaration (not shown in the figure). These attributes are also evaluated on demand. Thus, when a tool queries a specific attribute, only a subset of all available attributes will be evaluated. Usually, this subset is very small,



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

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

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

even for the first query when no attributes have yet been memoized.

4 Tool Integration

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

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

4.1 Command line Integration

Originally, IntraJ was developed as a command line tool, exhibiting competitive performance when compared to existing industrial tools [33]. The command line integration is suitable for continuous integration pipelines, as it can be easily integrated into existing workflows.

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

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

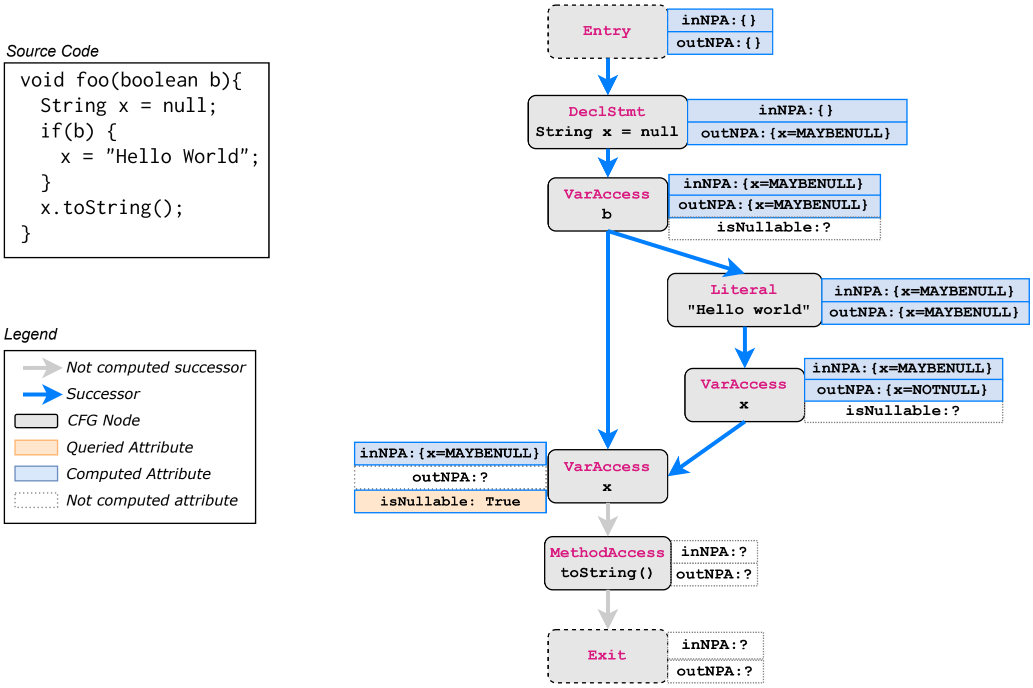
4.2 Editor Integration

The demand-driven evaluation in IntraJ makes it very suitable for integration with interactive tooling: analyses can be performed efficiently on individual program elements or files, even if the analysis depends on information in a larger project. To investigate this kind of integration, we used the MagpieBridge framework [25], which facilitates the integration of static analysers with IDEs that support LSP. MagpieBridge provides an abstraction layer between the LSP protocol and the static analysis tool, allowing for the development of IDE plug-ins with a very low effort. It reruns the analyses after each save in the editor. The supported LSP functionalities include diagnostics, code actions (e.g., quick fixes), and code-lens.

Figure 10 illustrates the integration of IntraJ with different IDEs via MagpieBridge. ServerAnalysis is a component we developed to handle the communication between IntraJ and the MagpieBridge Server. It is responsible for maintaining a record of the active analyses and forwarding events in the editor, such as the save command or opening of a file, to IntraJ. The analysis results are then sent back to MagpieBridge, which subsequently forwards them to the editor, displaying warnings, quick fixes, and explanations to the developer.

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

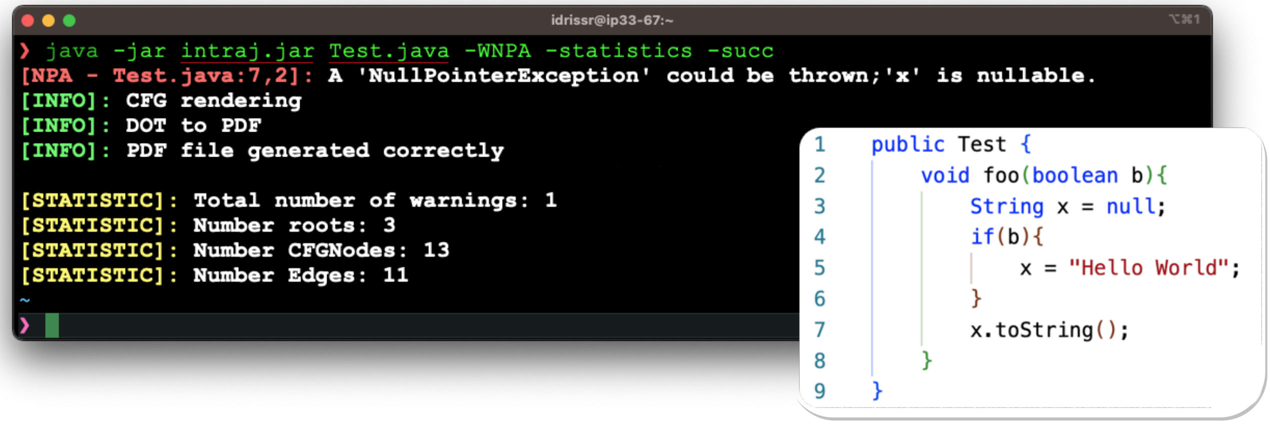
Figure 11 illustrates an example of interaction between IntraJ and Visual Studio Code. More specifically, it illustrates an instance of a potential `NullPointerException` detected by IntraJ and its representation within the IDE. The lightbulb icon (💡) indicates



Source Code

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

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



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

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

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

Fig. 9: Example of INTRAJ running on the command line.

that a quick fix is available, which can be applied by clicking on the icon.

4.3 CODEPROBER Integration

To investigate an even tighter integration with the editor, we have integrated INTRAJ with CODEPROBER [1], a tool for visualising and exploring the results of compilers and static analysers on an edited program. CODEPROBER supports exploring partial analysis results such as properties of AST nodes, and is therefore particularly suited for RAG-based tools that use on-demand evaluation. It is browser-based and can support visualizations beyond what is possible via the Language Server Protocol. The visualizations are live and updated as the user edits the analyzed program. Unlike LSP-based tools, CODEPROBER triggers attribute evaluation on each keystroke, enabling real-time interaction

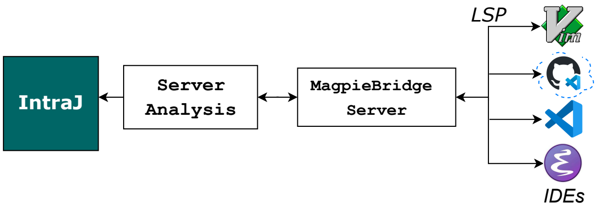
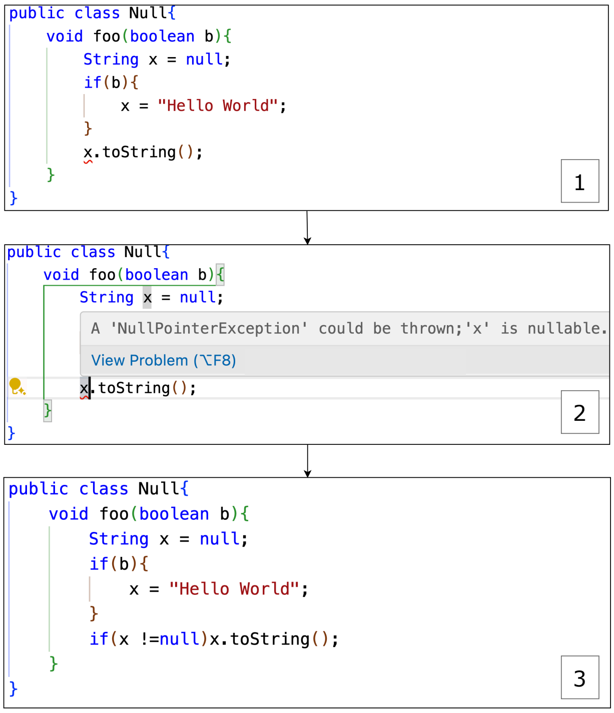


Fig. 10: Integration of IntraJ with IDEs through the use of the MagpieBridge framework.



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

1

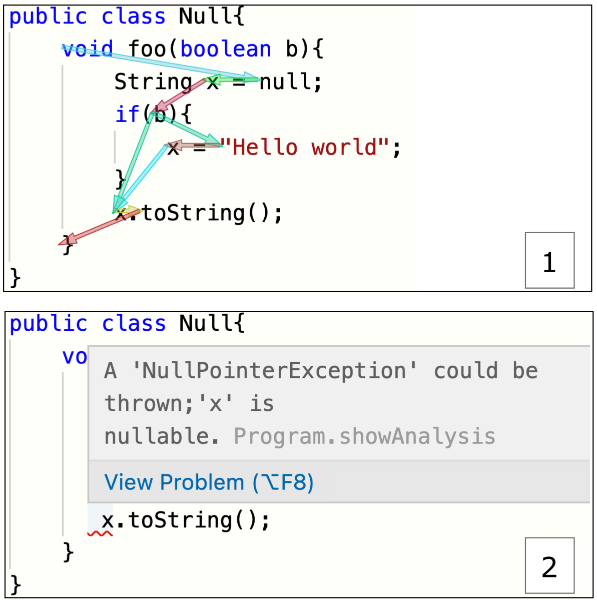
```
public class Null{
    void foo(boolean b){
        String x = null;
        A 'NullPointerException' could be thrown;'x' is nullable.
        View Problem (⌥F8)
        x.toString();
    }
}
```

2

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

3

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



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

1

```
public class Null{
    vo
      A 'NullPointerException' could be
      thrown;'x' is
      nullable. Program.showAnalysis
      View Problem (⌥F8)
        x.toString();
    }
}
```

2

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

with the analysis results. In contrast, LSP-based tools typically perform analysis less frequently, such as on file save or when a file is opened. As demonstrated in Section 6, IntraJ is fast enough to support the execution of analyses on each keystroke, making it suitable for integration with CodeProber.

The example in Figure 12 shows the visual representation of the CFG on top of the source code. The CFG is generated by IntraJ and visualised by CodeProber. It also shows the diagnostics that are displayed when hovering over the squiggly lines. The CFGs and the analysis results are computed automatically at each keystroke, allowing developers to interact with the results in real-time.[4]

5 Extending IntraJ

IntraJ is inherently extensible due to its use of Reference Attribute Grammars [14].

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

5.1 Extending IntraJ's Functionality

IntraJ can be extended by adding new modules that provide additional attributes and equations to the existing AST. JastAdd provides a modular extension mechanism for RAGs, which allows new modules to be added to IntraJ without modifying the existing codebase. These mod-

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

Listing 5: Definition of the isFaint attribute, which determines if an assignment is implicitly dead.

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

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

ules can be independently developed and, if desired, subsequently integrated into IntraJ to provide supplementary functionality. To illustrate this extensibility, we provide examples from various IntraJ submodules: one demonstrating the addition of a new analysis, another showcasing a separate analysis for information flow, and a third example detailing how existing analyses can be extended to support newer Java versions.

5.2 Addition of New Analyses

The existing dataflow and client analyses serve as examples for how to add new dataflow-based analyses. As an example, consider the Faint Assignment module. It uses results from the ordinary Dead Assignment module and from the Reaching Definitions (RD) module. The key attribute that defines this analysis is the isFaint attribute, which is shown in Listing 5. This attribute is defined on the CFGNode interface, which is implemented by all nodes in the CFG. The attribute definition is written as a method body, but without side-effects, using other attributes as needed (see API Figure 6). It uses the isDeadAssign attribute, computed by the ordinary Dead Assignment module. It also uses a local attribute allUses (not shown) that in turn is defined using the inRD attribute from the Reaching Definition component. Because the analysis is implemented using RAGs, it is automatically evaluated on-demand.

5.3 Information Flow

Another example of an extension to the IntraJ framework is SinfoJ [38], an information flow analysis similar to JFlow [28]. The goal of this analysis is to detect if sensitive data is leaked to untrusted sources. The developer classifies variables according to how sensitive they are. In SinfoJ, the following lattice is used for labeling variables (from lower to higher): Bottom → Unclassified → Confidential → Secret → TopSecret. The analysis can then, for example, identify if a variable labeled as TopSecret is leaking information to variables classified with lower security labels.

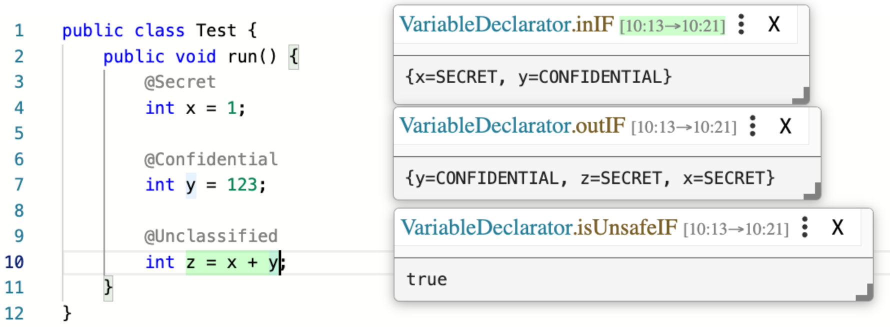
A simple example is shown in Figure 13, where variables are labelled using Java annotations. The analysis detects that the variable z is leaking information, since it is labeled as Unclassified, but uses information from variables with security labels Secret and Confidential.

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

In the implementation, in and out are defined as the attributes inIF and outIF (**IF** as in **I**nformation **F**low). Figure 13 shows the value of these attributes for the last variable declaration z. In this example, inIF is transformed into outIF by adding z=Secret, since that is the highest label used in the right-hand side of the assignment. The figure also includes the attribute isUnsafeIF, which has the value true for this declaration, meaning it is an information flow violation. It may be tedious to annotate all variables with labels, thus some of them can be omitted and automatically derived. For instance, it would be possible to omit the annotation for z. Then, the label would be derived to Secret, and the declaration would no longer yield a violation.

Part of the definition of the information flow is shown in Listing 6. The inIF and outIF are implemented like a normal forward analysis, using the attribute pred provided by IntraJ to propagate information forward in the control flow graph. Both attributes compute a value of the type IFDomain, which is a mapping from variables to security labels. The attribute inIF joins all outIF of its predecessors by keeping the highest label of each variable. The outIF attribute applies the transfer function to inIF, e.g., the effect of the CFG node. The transfer function is shown for variable declarations, where the highest label is used from either the annotation or the right-hand side of the declaration (if it has one). The attributes inIF and outIF are circular, like inNPA and outNPA (described in Section 2.3). The bottom value for the circular attributes is an empty variable map (new IFDomain()).

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



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

VariableDeclarator.inIF [10:13→10:21]

```
{x=SECRET, y=CONFIDENTIAL}
```

VariableDeclarator.outIF [10:13→10:21]

```
{y=CONFIDENTIAL, z=SECRET, x=SECRET}
```

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

```
true
```

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

Listing 6: Part of the definition of information flow in SINFOJ. (Source code from SINFOJ implementation. Complete code available at `https://bitbucket.org/jastadd/infoflow-exjobb-max-soller`)

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

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

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

is a screenshot of SINFOJ in CODEPROBER, illustrating how attributes easily can be explored when developing analyses.

5.4 New Language Constructs

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

Listing 7: Adding support for `EnhFor` construct to INTRAJ.

```
EnhFor implements CFGNode;

// Defines the first node that should be traversed when visiting an
// Enhanced For statement
eq EnhFor.firstNodes() = getExpr().firstNodes();

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

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

The code in Listing 7 shows an example of how the CFG is extended to support the *Enhanced For statement* introduced in Java 5. As can be seen from the Listing, the number of lines of code required to support the new language constructs is very small: The first two lines specify that the new construct also serve as CFG nodes. The remaining lines are equations that override default attribute definitions from INTRACFG (`firstNodes`, `nextNodes`, `nextContinue`) in order to define the control flow for the `EnhFor` AST node.

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

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

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

6 Evaluation

This section presents the performance evaluation of IntraJ, focusing on its suitability for on-demand analysis in interactive environments. While precision is essential for static analysis, this evaluation emphasizes the tool's performance. The precision of IntraJ has already been assessed in our previous work [33], where we demonstrated that IntraJ achieves precision comparable to SonarQube, a widely used industrial tool, and JastAddJ-IntraFlow [36], an academic RAG-based analysis tool. Here, our objective is to evaluate IntraJ's responsiveness and efficiency in interactive analysis scenarios, focusing on the time required to analyze compilation units.

The evaluation aims to demonstrate that IntraJ can provide interactive feedback to developers by analyzing most compilation units in under 0.1 seconds. To simulate an interactive environment, we evaluate each compilation unit independently, measuring the tool's performance in a realistic setting.

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

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

The evaluation of IntraJ uses Java benchmark projects selected from real-world applications, varying in both type and size. These benchmarks included projects from the DaCapo [5] and Qualitas [42] benchmark suites, such as `antlr` and `jfreechart`, alongside additional projects specifically chosen to cover a broad spectrum of applications, e.g., `extendj`. The selected projects ranged from 6,000 to 320,000 lines of code, ensuring a diverse set of benchmarks for the evaluation. A summary of the benchmarks used in the evaluation is presented in Table 1.

6.1 Dead Assignment Analysis

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

The results of this analysis are detailed in Table 2.

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

6.2 Null-Pointer Dereference Analysis

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

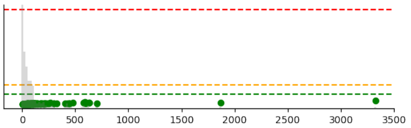
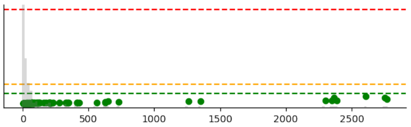
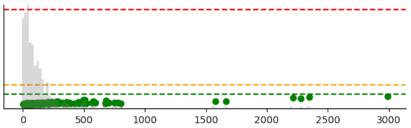
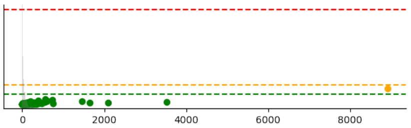
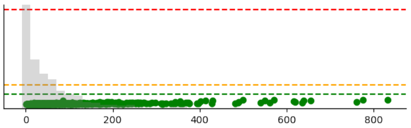
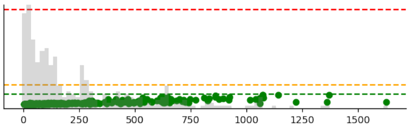
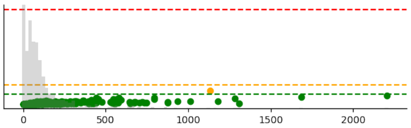
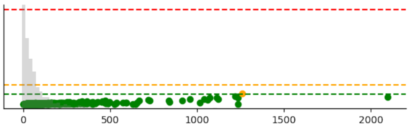
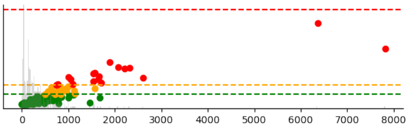
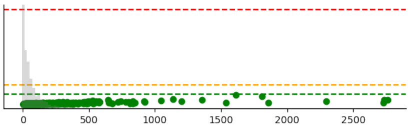
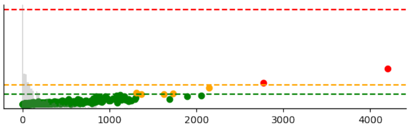
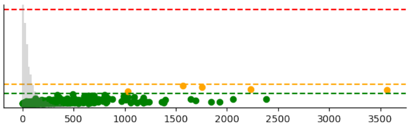
Benchmark	Analysis Time Range	File Count	Mean Analysis Time (s)	Mean File LOC	Measurments
commons-jxpath	≤ 0.1s	213 (100%)	0.0015	114	
	0.1s - 0.2s	0	–	–	
	0.2s - 1.0s	0	–	–	
antlr	≤ 0.1s	192 (100%)	0.0027	175	
	0.1s - 0.2s	0	–	–	
	0.2s - 1.0s	0	–	–	
jackson-core	≤ 0.1s	280 (100%)	0.0054	174	
	0.1s -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

Table 2: Steady-state performance of IntraJ for *Dead Assignment Analysis* for single compilation units across different Java benchmarks. Each plot overlays a histogram and a scatterplot, with the X axis representing LOC for both. The histogram (gray) shows the distribution of compilation unit sizes for each project, with relative frequency on the Y axis. The scatterplot shows the analysis times for each compilation unit on the Y axis, marked green (≤ 0.1 seconds), orange (0.1–0.2 seconds), or red (0.2–1.0 seconds). The dashed lines represent the boundaries at 0.1s (green), 0.2s (orange), and 1.0s (red).

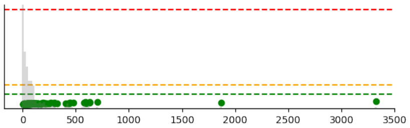
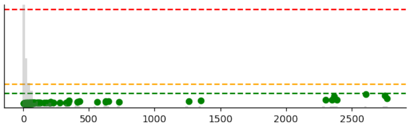
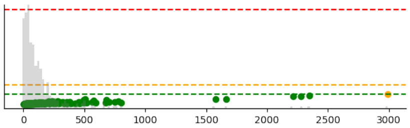
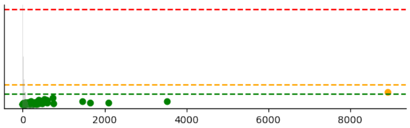
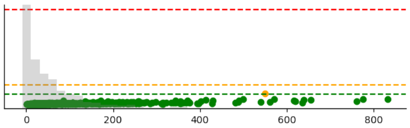
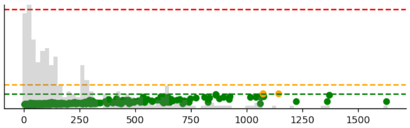
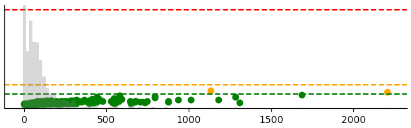
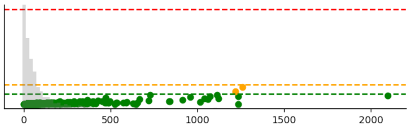
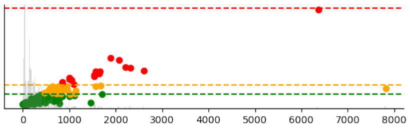
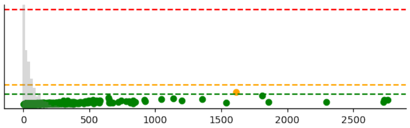
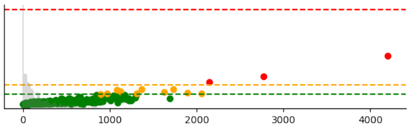
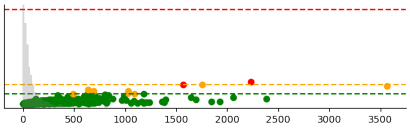
Benchmark	Analysis Time Range	File Count	Mean Analysis Time (s)	Mean File LOC	Measurments
COMMONS-JXPATH	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	213 (100%) 0 0	0.0021 – –	114 – –	
ANTLR	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	192 (100.00%) 0 0	0.0039 – –	175 – –	
JACKSON-CORE	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	279 (99.6%) 1 (0.4%) 0	0.0070 0.1033 –	163 2990 –	
PMD	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	751 (99.9%) 1 (0.1%) 0	0.0028 0.1249 –	69 8913 –	
STRUTS	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	1110 (99.9%) 1 (0.1%) 0	0.0031 0.1115 –	73 550 –	
JODA-TIME	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	327 (99.1%) 3 (0.9%) 0	0.0130 0.1078 –	255 1097 –	
JFREECHART	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	734 (99.7%) 2 (0.3%) 0	0.0064 0.1344 –	126 1669 –	
FOP	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	1045 (99.8%) 2 (0.2%) 0	0.0033 0.1555 –	96 1239 –	
EXTENDJ	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	349 (88.1%) 31 (7.8%) 16 (4.0%)	0.0253 0.1500 0.3632	239 1073 1903	
CASTOR	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	1710 (99.9%) 1 (0.1%) 0	0.0025 0.1286 –	93 1610 –	
WEKA	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	1210 (98.9%) 10 (0.8%) 3 (0.3%)	0.0085 0.1260 0.3431	184 1405 3041	
POI	≤ 0.1s 0.1s - 0.2s 0.2s - 1.0s	2948 (99.6%) 9 (0.3%) 2 (0.1%)	0.0048 0.1344 0.2144	107 1276 1902	

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

Null-pointer dereference analysis is generally more computationally intensive than dead assignment analysis due to the requirement of constructing control-flow graphs in both directions. Despite the increased computational requirements, the analysis times remain low, with the majority of compilation units being processed in under 0.1 seconds. The results for the *Null-Pointer Dereference Analysis* are presented in Table 3.

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

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

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

7 Limitations

While INTRAJ demonstrates promising results in providing efficient, demand-driven analysis for Java code within interactive development environments, there are engineering limitations to its approach in both precision and performance, which we discuss here alongside potential future directions.

INTRAJ's CFGs are not sound, i.e., the CFGs may not capture all possible control flows in the analyzed code. One source of unsoundness is exception flow: INTRAJ models checked and unchecked exception flows for exceptions that are explicitly thrown, but does not account for exceptions that may occur implicitly at runtime, such as `StackOverflow` or `OutOfMemoryError`. As a result, certain runtime exceptions are not reflected in the CFG unless explicitly declared, potentially impacting the accuracy of the analysis in scenarios involving implicit exception handling.

Another source of unsoundness arises from Java reflection and the Java Native Interface (JNI), neither of which are addressed by INTRAJ. With reflection, developers can access fields and methods that might not be visible in the source code, or at least not directly, posing significant challenges for static analysis. This dynamic behavior is a research topic in itself [22,4,24]. This limitation aligns with the perspective of the Soundiness Manifesto [23], which suggests making deliberate trade-offs in soundness for usability and responsiveness. Our experiments demonstrate that static analysis can be made more interactive and responsive, enhancing its usability without compromising its practical utility.

The analyses presented here are limited to an intraprocedural scope, as extending INTRAJ to interprocedural analysis would likely impact the responsiveness required for IDE integration. Balancing the precision benefits of interprocedural analysis with the need for low response times remains an area for future research.

8 Related Work

The challenge of balancing analysis complexity with tool responsiveness is a well-known issue in the field of static analysis.

Existing extensible static analysis frameworks like SPOTBUGS [39], SOOT [21], or INFER [6] are generally designed for throughput, rather than responsiveness, reflecting their original intended use for batch program analysis. Depending on the internal architecture, increasing responsiveness for interactive use may require nontrivial re-engineering.

For example, Distefano et al. manually re-engineered parts of INFER [9] to support incremental updates, in order to scale to larger code bases with frequent changes.

Arzt et al. exploited properties of the IFDS/IDE analysis framework underlying SOOT to incrementalise analysis [2], but still needed to make some architectural adjustments.

Prior work has demonstrated strategies that allow analysis frameworks to automatically incrementalise declaratively specified analyses. Dura et al. demonstrate this at the file level [10] for various bug checkers, while Szabo et al. show fine-grained incrementality for a points-to analysis [41]. Both approaches use declarative logic programming to specify their analyses.

INTRAJ builds on the RAG framework JASTADD [15], which provides declarative interfaces between attributes computed by different components, while implementing these components in Java rather than in a declarative logic language. This design naturally ensures that INTRAJ-based analyses operate in a demand-driven fashion, computing attributes only when required, thereby optimizing efficiency.

In recent years, demand-driven static analysis frameworks have focused on improving the efficiency and responsiveness of analyses for interactive development environments. Stein et al. [40] introduced the concept of demanded summarization, an approach for incrementally updating compositional analyses by dynamically reusing existing summaries for unmodified code. This technique demonstrates how incremental abstract interpretation can maintain analysis precision and consistency even in the face of complex changes, supporting interactive feedback within IDEs.

Similarly, Erhard et al. [12] presented an approach for multithreaded programs, which addresses the challenges of incremental analysis in concurrent settings. Their framework limits reanalysis to affected code sections, significantly reducing response time and enhancing the interactivity of analyses within IDEs for multithreaded C programs.

Finally, Söderberg et al. [37] proposed a strategy for using dynamic dependency tracking to extend the demand-driven RAG evaluation model to an incremental analysis model that can re-use information from unchanged parts of the program, potentially further increasing responsiveness during interactive editing.

9 Conclusions and Future Development

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

We discussed the architecture of IntraJ in detail, illustrating how it takes advantage of demand-driven evaluation with practical examples. To support these claims, we conducted experiments on real-world and diverse codebases, demonstrating that IntraJ can analyze most compilation units in under 0.1 seconds with the provided analyses.

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

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

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

Acknowledgements

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

References

1. Alaküla, A.R., Hedin, G., Fors, N., Pop, A.: Property probes: Live exploration of program analysis results. J. Syst. Softw. **211**, 111980 (2024). URL `https://doi.org/10.1016/j.jss.2024.111980`
2. Arzt, S., Bodden, E.: Reviser: efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In: Proceedings of the 36th International Conference on Software Engineering, pp. 288–298 (2014)
3. Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J.D., Penix, J.: Using static analysis to find bugs. IEEE software **25**(5), 22–29 (2008)
4. Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., d'Amorim, M., Ernst, M.D.: Static analysis of implicit control flow: Resolving java reflection and android intents (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 669–679 (2015). DOI 10.1109/ASE.2015.69
5. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: 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, pp. 169–190. ACM Press, New York, NY, USA (2006). DOI 10.1145/1167473.1167488. URL `http://doi.acm.org/10.1145/1167473.1167488`
6. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3, pp. 459–465. Springer (2011)
7. Christakis, M., Bird, C.: What developers want and need from program analysis: an empirical study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE '16, p. 332–343. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2970276.2970347. URL `https://doi.org/10.1145/2970276.2970347`
8. Copeland, T.: PMD applied, vol. 10. Centennial Books Alexandria, Va, USA (2005)
9. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019). DOI 10.1145/3338112. URL `https://doi.org/10.1145/3338112`
10. Dura, A., Reichenbach, C., Söderberg, E.: JavaDL: Automatically Incrementalizing Java Bug Pattern Detection. In: Proceedings of the ACM on Programming Languages. ACM (2021). DOI 10.1145/3485542
11. Ekman, T., Hedin, G.: The Jastadd extensible java compiler. In: R.P. Gabriel, D.F. Bacon, C.V. Lopes, G.L.S. Jr. (eds.) 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, pp. 1–18. ACM (2007). DOI 10.1145/1297027.1297029. URL `https://doi.org/10.1145/1297027.1297029`
12. Erhard, J., Saan, S., Tilscher, S., Schwarz, M., Holter, K., Vojdani, V., Seidl, H.: Interactive abstract interpretation: reanalyzing multithreaded c programs for cheap. International Journal on Software Tools for Technology Transfer (2024). DOI 10.1007/s10009-024-00768-9. Publisher Copyright: © The Author(s) 2024.
13. Farrow, R.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. ACM SIGPLAN Notices **21**(7), 85–98 (1986)
14. Hedin, G.: Reference attributed grammars. Informatica (Slovenia) **24**(3) (2000)

15. Hedin, G., Magnusson, E.: Jastadd - a java-based system for implementing front ends. Electron. Notes Theor. Comput. Sci. **44**(2), 59–78 (2001). DOI 10.1016/S1571-0661(04)80920-4. URL https://doi.org/10.1016/S1571-0661(04)80920-4
16. Hedin, G., Magnusson, E.: Jastadd—an aspect-oriented compiler construction system. Science of Computer Programming **47**(1), 37–58 (2003)
17. Jones, L.G., Simon, J.: Hierarchical VLSI design systems based on attribute grammars. In: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 58–69 (1986)
18. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta informatica **7**(3), 305–317 (1977)
19. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '73, p. 194–206. Association for Computing Machinery, New York, NY, USA (1973). DOI 10.1145/512927.512945. URL https://doi.org/10.1145/512927.512945
20. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory **2**(2), 127–145 (1968)
21. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011), vol. 15 (2011)
22. Li, Y., Tan, T., Xue, J.: Understanding and analyzing java reflection. ACM Trans. Softw. Eng. Methodol. **28**(2) (2019). DOI 10.1145/3295739. URL https://doi.org/10.1145/3295739
23. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015). DOI 10.1145/2644805. URL https://doi.org/10.1145/2644805
24. Livshits, V.B., Whaley, J., Lam, M.S.: Reflection analysis for java. In: K. Yi (ed.) Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3780, pp. 139–160. Springer (2005). DOI 10.1007/11575467_11. URL https://doi.org/10.1007/11575467_11
25. Luo, L., Dolby, J., Bodden, E.: MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In: A.F. Donaldson (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 134, pp. 21:1–21:25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). DOI 10.4230/LIPIcs.ECOOP.2019.21. URL http://drops.dagstuhl.de/opus/volltexte/2019/10813
26. Magnusson, E., Hedin, G.: Circular reference attributed grammars — their evaluation and applications. Science of Computer Programming **68**(1), 21–37 (2007). DOI https://doi.org/10.1016/j.scico.2005.06.005. URL https://www.sciencedirect.com/science/article/pii/S0167642307000767. Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA '03)
27. Magnusson, E., Hedin, G.: Circular reference attributed grammars—their evaluation and applications. Science of Computer Programming **68**(1), 21–37 (2007)
28. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: A.W. Appel, A. Aiken (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999, pp. 228–241. ACM (1999). DOI 10.1145/292540.292561. URL https://doi.org/10.1145/292540.292561
29. Nielsen, J.: Usability engineering. Morgan Kaufmann (1994)
30. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Publishing Company, Incorporated (2010)
31. Öqvist, J.: Contributions to declarative implementation of static program analysis. Ph.D. thesis, Lund University, Sweden (2018). URL http://lup.lub.lu.se/record/82b210fc-6d15-4f0a-82ff-24b024925d23
32. Riouak, I., Fors, N., Öqvist, J., Hedin, G., Reichenbach, C.: Efficient demand evaluation of fixed-point attributes using static analysis. In: Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering, SLE '24, pp. 56–69. Association for Computing Machinery, New York, NY, USA (2024). DOI 10.1145/3687997.3695644. URL https://doi.org/10.1145/3687997.3695644
33. Riouak, I., Reichenbach, C., Hedin, G., Fors, N.: A precise framework for source-level control-flow analysis. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 1–11. IEEE (2021). DOI 10.1109/SCAM52516.2021.00009
34. Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., Winter, C.: Tricorder: Building a program analysis ecosystem. In: A. Bertolino, G. Canfora, S.G. Elbaum (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pp. 598–608. IEEE Computer Society (2015). DOI 10.1109/ICSE.2015.76. URL https://doi.org/10.1109/ICSE.2015.76
35. Sawyer, K.: Mystery of orbiter crash solved. Washington Post p. A1 (1999). URL https://www.washingtonpost.com/wp-srv/national/longterm/space/stories/orbiter100199.htm. Last accessed: 2024-07-26
36. Söderberg, E., Ekman, T., Hedin, G., Magnusson, E.: Extensible intraprocedural flow analysis at the abstract syntax tree level. Sci. Comput. Program. **78**(10), 1809–1827 (2013). DOI 10.1016/J.SCICO.2012.02.002. URL https://doi.org/10.1016/j.scico.2012.02.002
37. Söderberg, E., Hedin, G.: Incremental evaluation of reference attribute grammars using dynamic dependency tracking (2012). LU-CS-TR:2012-249 (2012).
38. Soller, M.: Sinfoj: A simple information flow analysis with reference attribute grammars (2023). Available at http://lup.lub.lu.se/student-papers/record/9149210
39. SpotBugs. https://spotbugs.github.io/. Accessed: 2023-02-17
40. Stein, B., Chang, B.Y.E., Sridharan, M.: Interactive abstract interpretation with demanded summarization. ACM Trans. Program. Lang. Syst. **46**(1) (2024). DOI 10.1145/3648441. URL https://doi.org/10.1145/3648441
41. Szabó, T., Erdweg, S., Bergmann, G.: Incremental whole-program analysis in datalog with lattices. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 1–15 (2021)
42. Tempero, E., Anslow, G., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. Software Engineering Conference, pp. 336–345 (2010)