A Precise Framework for Source-Level Control-Flow Analysis

Idriss Riouak*, Christoph Reichenbach*, Görel Hedin*, and Niklas Fors*
**idriss.riouak*, *christoph.reichenbach*, *gorel.hedin*, and *niklas.fors* (*@cs.lth.se*)
Department of Computer Science, Lund University, Sweden

***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.**

***Index Terms*—Control flow, Attributed Grammars, Static Analysis, Declarative, Dataflow.**

I. INTRODUCTION

Static program analysis plays an important role in software development, and may help developers detect subtle bugs such as null pointer exceptions [16] or security vulnerabilities [33].

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) [13] and addresses these challenges. We build on an earlier approach that also used RAGs [35] 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 [9]. 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.

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

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.
- 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 II) and INTRAJ, along with underlying design decisions and implementation details (Section III), present our client analyses (Section IV) and evaluation (Section V), discuss related work (Section VI) and conclude (Section VII).

II. RAGS AND THE INTRACFG FRAMEWORK

Attribute grammars, originally introduced by Knuth [21], 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

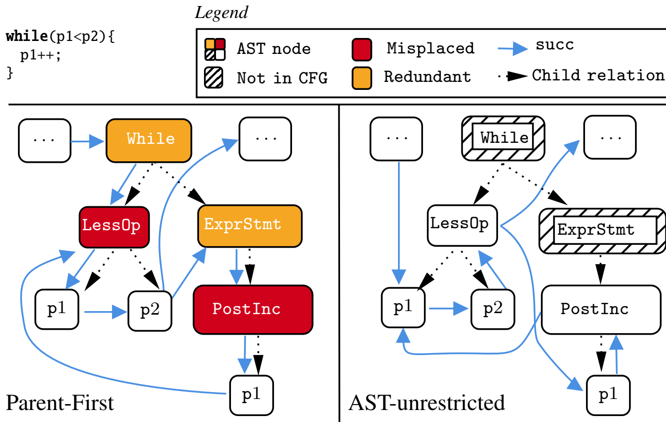

Fig. 1: In the Parent-First CFG (left) a parent always precedes its children, resulting in redundant and misplaced nodes. The AST-unrestricted CFG (right) is correct and minimal.

of expressions. Inherited attributes are useful for propagating information downwards, e.g., for environment information.

Reference Attribute Grammars (RAGs) [13] 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 [14], which supports RAGs as well as the following attribute grammar extensions that we use here:

Higher-order attributes (HOAs) [42] 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 [27]. 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 [26]. 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 [12].

Attribution aspects are modules that use inter-type declarations to declare a set of attributes, equations, collection contributions, etc. for specific node types [14], 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 [18]. JastAdd exclusively uses this evaluation strategy.

A. RAG frameworks for control flow

Our work is the second to construct CFGs in a RAG framework, following the earlier JASTADDJ-INTRAFLOW [35]. 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 [15], [37] 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.

B. 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

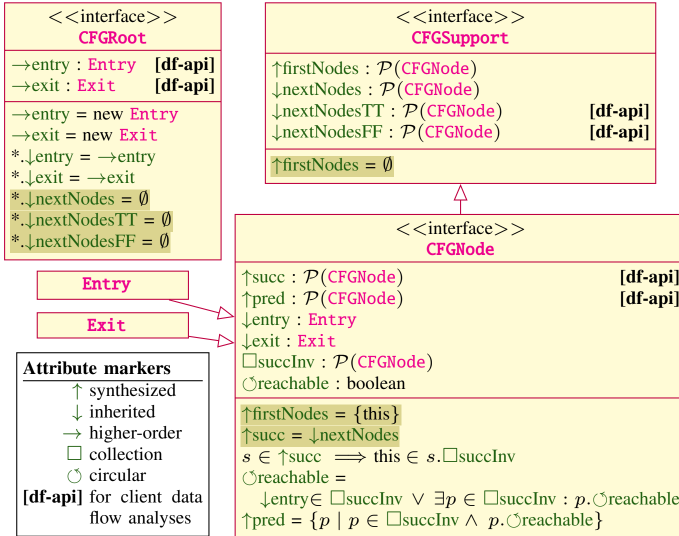

Fig. 2: The INTRACFG framework with interfaces CFGRoot, CFGSupport, CFGNode, and synthetic AST types Entry, Exit. Highlighted attribute equations are default equations, intended for overriding.

CFGSupport interface marks AST nodes in a location that may contain CFGNodes. All CFGNodes are CFGSupport nodes, but CFGSupport nodes that are not CFGNodes 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 CFGNodes. We optionally prefix attribute names with ↑, ↓, →, □, or ⟳ to highlight the AST traversal underlying their computation. For the different kinds of attributes, we use the following equations, for attributes x and expressions e:

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

Inherited attributes: c.↓x $= e$ gives AST child node c and its descendants access to e through ↓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, $*$.↓x $= e$.

Higher-order attributes: →x $= e$ where e must construct a fresh AST subtree.

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

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

This pseudocode translates straightforwardly to more verbose JastAdd code that uses Java for the right-hand sides in our equations. INTRACFG is 45 LOC of JastAdd code.[1]

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

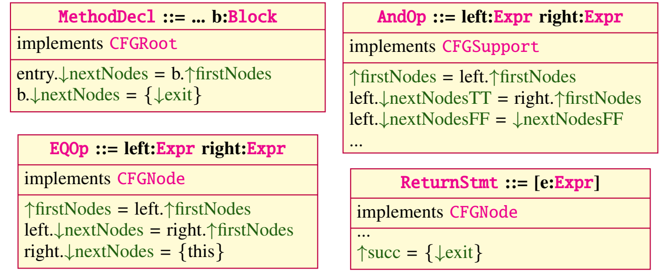

Fig. 3: Example application of the INTRACFG framework.

```
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.

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 IV.

C. 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.

The ↓nextNodes attribute contains the CFGNodes 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.

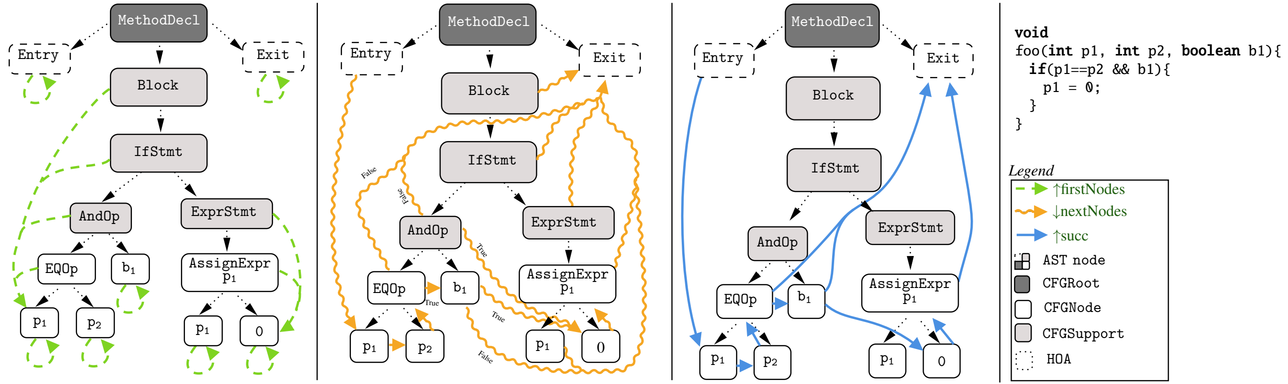

Fig. 4: Visualization of the attributes ↑firstNodes, ↓nextNodes and ↑succ. For boolean expressions (AndOp and EQOp), the subsets ↓nextNodesTT and ↓nextNodesFF are shown instead of ↓nextNodes, marked by True and False, respectively.

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

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

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

Figure 4 illustrates these attributes on a small program in a language with methods, statements, and expressions. Here, MethodDecl is a CFGRoot and thus automatically has fresh Entry and Exit nodes. Nodes in the control flow, e.g., identifiers and the equality-check operator, EQOp, are CFGNodes, 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., p_1) is not part of the flow (cf. Section III-A).

D. 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 CFGNodes 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).

III. 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.

A. 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.

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

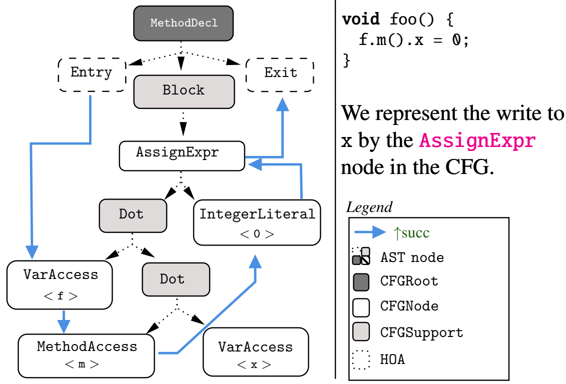

Fig. 5: An assignment with a complex left-hand side.

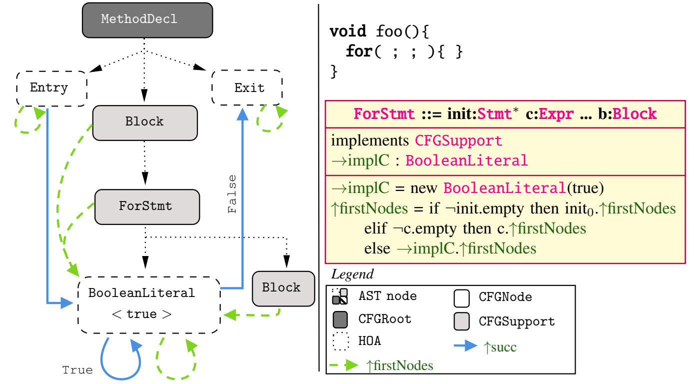

Fig. 6: CFG for method with empty *ForStmt*. The HOA →implC reifies the implicit *true* condition.

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 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.

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.

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

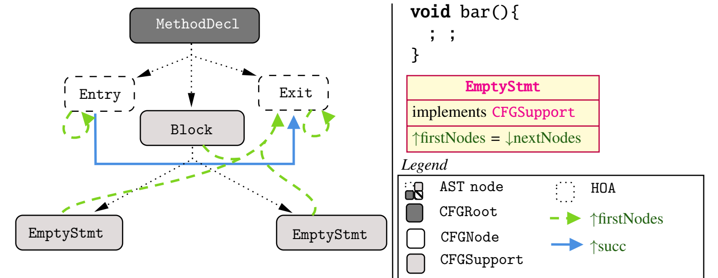

Fig. 7: The CFG can entirely skip AST nodes.

`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

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

B. 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`, 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`:

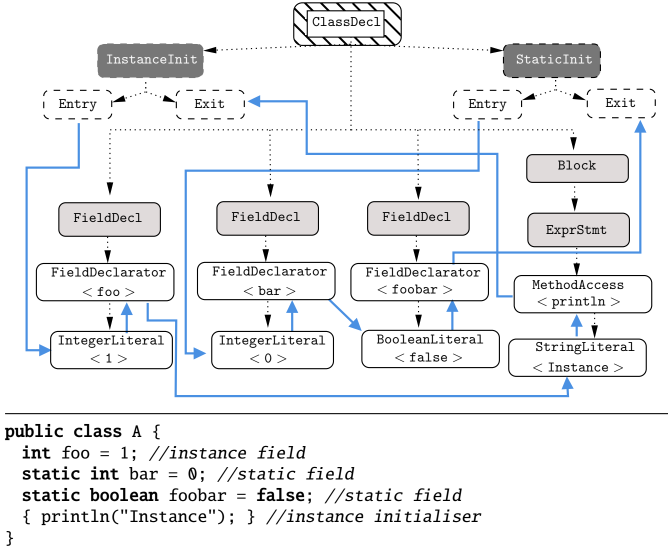

Fig. 8: Example of class that interleaves static and instance initialisers. The →instanceInit and →staticInit HOAs represent the CFGs for each kind of initialisers.

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.

C. Exceptions Modelling

Control flow for exceptions is complex to model and often requires non-trivial approximations [1], [4], [17]. 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 exception 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
}
```

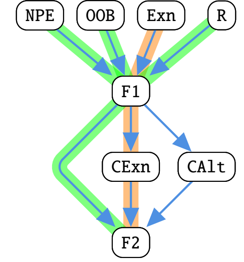

Fig. 9: Complex exception flow in a conservative CFG. Only the flow paths in green and orange are realisable.

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 [28], 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. [4], 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 V shows some of the practical strengths and weaknesses of our heuristic.

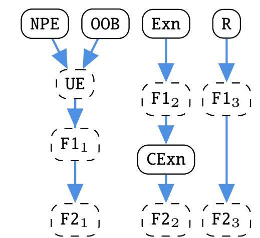

Fig. 10: Path-sensitive variant of the CFG from Figure 9, used in IntraJ.

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.

IV. Client Analysis

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

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

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

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.

A. 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 $\top = \textbf{nully}$ signifies *value may be* `null` and $\bot = \textbf{nonnull}$ 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 [6]. Our monotonic transfer function $f_{NPA} : (\mathcal{A} \to \mathcal{L}_2) \times E \to (\mathcal{A} \to \mathcal{L}_2)$ is straightforward:

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

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.

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 $\circlearrowleft\text{in}_{NPA}$, which

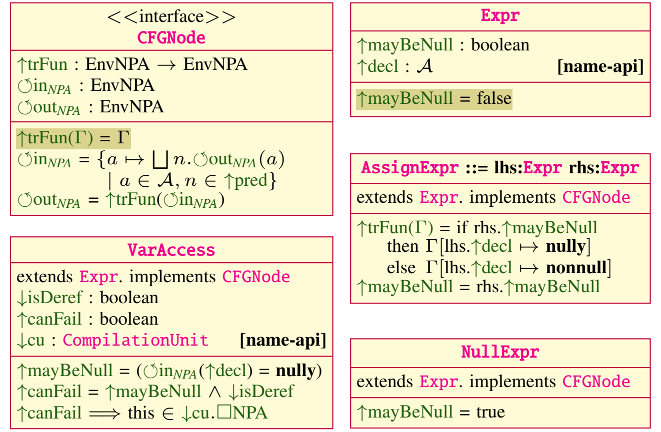

Fig. 11: Partial implementation of our NPA. We obtain ↑decl and ↓cu from ExtendJ's name analysis API.

merges all evidence that flows in from control flow predecessors, and $\circlearrowleft\text{out}_{NPA}$, which applies the local transfer function ↑trFun to $\circlearrowleft\text{in}_{NPA}$. While NPA is a forward analysis, JastAdd's on-demand semantics mean that we query *backwards*, following ↑pred edges, when we compute $\circlearrowleft\text{in}_{NPA}$ on demand. $\circlearrowleft\text{in}_{NPA}$ and $\circlearrowleft\text{out}_{NPA}$ 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 ↓cu.□NPA, the compilation unit-wide collection attribute of likely `null` pointer dereferences, whenever ↑mayBeNull holds and when the `VarAccess` is also a proper prefix of an access path and must therefore be dereferenced (↓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.

B. 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 [29]. 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:

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

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

We encode the f_{LVA} using RAGs in a similar way as done in [35]: Figure 12 shows our computation where circular attributes $\circlearrowleft\text{in}_{LVA}$ and $\circlearrowleft\text{out}_{LVA}$ represent variables live

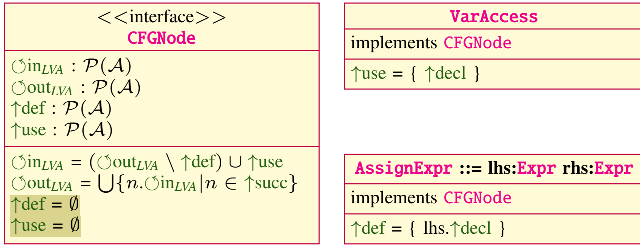

Fig. 12: Partial implementation of our LVA.

before/after a `CFGNode`. Here, $\circlearrowleft out_{LVA}$ reads from $\uparrow$succ nodes, since we are implementing an on-demand backward analysis. `VarAccess` and `AssignExpr` override $\uparrow$use and $\uparrow$def, respectively. Since the CFG traverses through the right-hand side of each assignment, this specification suffices to capture the analysis of our Java language fragment. Our full implementation for Java 7 takes up 38 lines of code.

C. 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.

V. Evaluation and Results

We demonstrate the utility of IntraCFG and IntraJ[3] by evaluating the client analyses that we describe in Section IV 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 [2], as well as JFreeChart (JFC), which is a superset of the Chart benchmark. These benchmarks mostly subsume the ones used by JJI [35], except for replacing Bloat by the more readily available and larger PMD. Table I 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%.

A. Precision

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

[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 I: 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.

a) Dead Assignment Analysis: JJI and SQ provide subtly different DAA variants. JJI's DAA corresponds largely to our LVA (Section IV-B) with minimal filtering, while SQ additionally applies the default value filtering heuristic from Section IV-C. 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 checkers. 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 IV-C). 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 III-C). These do not affect JJI (and possibly SQ), since JJI conservatively merges exceptional and regular control flow.

b) 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

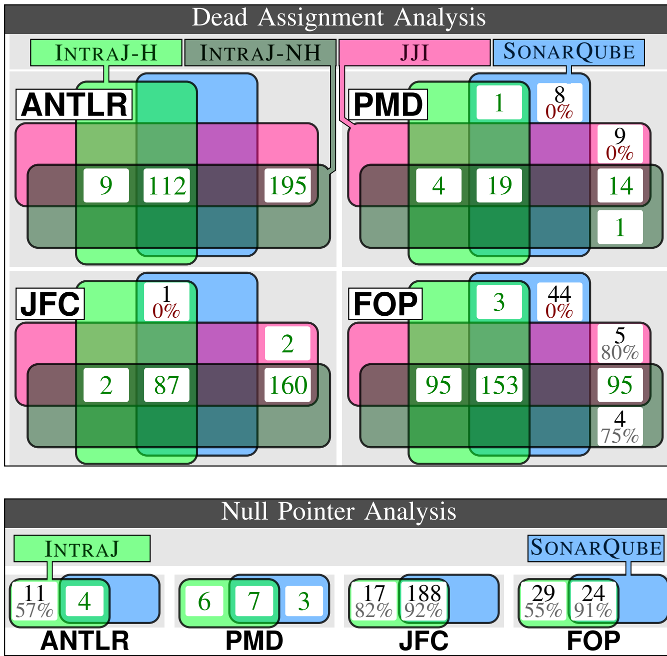

Fig. 13: Venn diagram: number of reports shared across checkers, and percentage of true positives (unless 100%).

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

B. 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.

Table II summarises our results. The Baseline column 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 II 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 $\%_{\text{JJI}}$ and $\%_{\text{SQ}}$ 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).

VI. Related Work

Our work is most similar to JASTADDJ-INTRAFLOW [35], 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 [35] only supported Java 5.

Earlier work on adding control flow to attribute grammars includes a language extension to the Silver attribute grammar system [39], [40] which supports that AST nodes are marked as CFG nodes, and successors are defined using an inherited

TABLE II: Benchmark mean execution time and 95% confidence intervals over 50 data points per reported number.

Benchmark	Steady state		
	INTRAJ(s)	JJI(s)	$\%_{JJI}$
ANTLR	0.05±0.00	0.04±0.00	125
	0.12±0.00	0.13±0.00	92
	0.27±0.01	-	-
PMD	0.07±0.00	0.06±0.00	116
	0.12±0.00	0.16±0.00	75
	0.26±0.00	-	-
JFC	0.12±0.00	0.12±0.00	100
	0.25±0.00	0.34±0.00	73
	0.60±0.01	-	-
FOP	0.14±0.00	0.17±0.00	82
	0.26±0.00	0.39±0.00	66
	0.67±0.01	-	-

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 [7] 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 [10] supports data flow analysis and program transformation on byte code. Meanwhile, Flix [24] 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 [25]. 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 [8] illustrates how to synthesise fact extractors from a JastAdd-based language, and we expect that it can directly expose INTRAJ edges.

FlowSpec [34] 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 [22] 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 [3], term rewriting systems [41], and XPath processors [5].

Our work has focused on intraprocedural data flow analyses [6], [19], [20]. However, existing (IR-based) program analysis tools like Soot [38], Wala [11], or Opal [15] 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 [23], analogously to Datalog-based analyses [32]. While we expect that it is possible to integrate highly scalable data flow algorithms like IFDS, IDE [30], [31], or SPPD [36] 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.

VII. 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.

REFERENCES

[1] A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Provably correct control flow graphs from java bytecode programs with exceptions. *International journal on software tools for technology transfer*, 18(6):653–684, 2016.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 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*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[3] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of OOPSLA '09*, pages 243–262, New York, NY, USA, 2009. ACM.

[4] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. *ACM SIGSOFT Software Engineering Notes*, 24(5):21–31, 1999.

[5] T. Copeland. *PMD applied*, volume 10. Centennial Books Arexandria, Va, USA, 2005.

[6] P. Cousot and R. 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, pages 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[7] C. De Roover. A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. 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.

[8] A. Dura, H. Balldin, and C. Reichenbach. Metadl: Analysing datalog in datalog. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 38–43. ACM, 2019.

[9] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, 2007.

[10] H. Falconer, P. H. J. Kelly, D. M. Ingram, M. R. Mellor, T. Field, and O. Beckmann. A declarative framework for analysis and optimization. In S. Krishnamurthi and M. Odersky, editors, *Compiler Construction*, pages 218–232, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[11] S. Fink and J. Dolby. Wala–the tj watson libraries for analysis, 2012.

[12] N. Fors, E. Söderberg, and G. Hedin. Principles and patterns of jastadd-style reference attribute grammars. In R. Lämmel, L. Tratt, and J. de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 86–100. ACM, 2020.

[13] G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[14] G. Hedin and E. Magnusson. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[15] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini. 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*, pages 184–196, 2020.

[16] D. Hovemeyer, J. Spacco, and W. 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*, pages 13–19, 2005.

[17] J.-W. Jo and B.-M. Chang. Constructing control flow graph for java by decoupling exception flow from normal flow. In *International Conference on Computational Science and Its Applications*, pages 106–113. Springer, 2004.

[18] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 1984.

[19] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[20] G. 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, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[21] D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[22] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. *ACM SIGPLAN Notices*, 40(1):364–377, 2005.

[23] O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In *International Conference on Compiler Construction*, pages 153–169. Springer, 2003.

[24] M. Madsen and O. Lhoták. Fixpoints for the masses: programming with first-class datalog constraints. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[25] M. Madsen, M.-H. Yee, and O. Lhoták. From datalog to flix: a declarative language for fixed points on lattices. *ACM SIGPLAN Notices*, 51(6):194–208, 2016.

[26] E. Magnusson, T. Ekman, and G. Hedin. Extending attribute grammars with collection attributes–evaluation and applications. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 69–80. IEEE, 2007.

[27] E. Magnusson and G. Hedin. Circular reference attributed grammars—their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[28] J. Öqvist. *Contributions to Declarative Implementation of Static Program Analysis*. PhD thesis, Lund University, 2018.

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

[30] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[31] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[32] Y. Smaragdakis and M. Bravenboer. Using datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*, pages 245–251. Springer, 2010.

[33] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 248–259, 2015.

[34] J. Smits, G. Wachsmuth, and E. Visser. Flowspec: A declarative specification language for intra-procedural flow-sensitive data-flow analysis. *Journal of Computer Languages*, 57:100924, 2020.

[35] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Sci. Comput. Program.*, 78(10):1809–1827, Oct. 2013.

[36] J. Späth, K. Ali, and E. Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL), Jan. 2019.

[37] T. Szabó. *Incrementalizing Static Analyses in Datalog*. PhD thesis, Johannes Gutenberg-Universität Mainz.

[38] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, Riverton, NJ, USA, 2010. IBM Corp.

[39] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1):39–54, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[40] E. Van Wyk and L. Krishnan. Using verified data-flow analysis-based optimizations in attribute grammars. *Electronic Notes in Theoretical Computer Science*, 176(3):109–122, 2007.

[41] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. *Lecture Notes in Computer Science*, 3016:216–238, June 2004.

[42] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *ACM SIGPLAN Notices*, 24(7):131–145, 1989.