JavaDL: Automatically Incrementalizing Java Bug Pattern Detection

ALEXANDRU DURA, Lund University, Sweden
CHRISTOPH REICHENBACH, Lund University, Sweden
EMMA SÖDERBERG, Lund University, Sweden

Static checker frameworks support software developers by automatically discovering bugs that fit general-purpose bug patterns. These frameworks ship with hundreds of detectors for such patterns and allow developers to add custom detectors for their own projects. However, existing frameworks generally encode detectors in imperative specifications, with extensive details of not only *what* to detect but also *how*. These details complicate detector maintenance and evolution, and also interfere with the framework's ability to change how detection is done, for instance, to make the detectors incremental.

In this paper, we present JavaDL, a Datalog-based declarative specification language for bug pattern detection in Java code. JavaDL seamlessly supports both exhaustive and incremental evaluation from the same detector specification. This specification allows developers to describe local detector components via *syntactic pattern matching*, and nonlocal (e.g., interprocedural) reasoning via *Datalog-style logical rules*. We compare our approach against the well-established SpotBugs and Error Prone tools by re-implementing several of their detectors in JavaDL. We find that our implementations are substantially smaller and similarly effective at detecting bugs on the Defects4J benchmark suite, and run with competitive runtime performance. In our experiments, neither incremental nor exhaustive analysis can consistently outperform the other, which highlights the value of our ability to transparently switch execution modes. We argue that our approach showcases the potential of *clear-box static checker frameworks* that constrain the bug detector specification language to enable the framework to adapt and enhance the detectors.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Constraint and logic languages**; *Domain specific languages*; • **Theory of computation** → *Pattern matching*.

Additional Key Words and Phrases: Datalog, Syntactic Patterns, Static Analysis Frameworks, Software Bugs

ACM Reference Format:
Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021. JavaDL: Automatically Incrementalizing Java Bug Pattern Detection. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 165 (October 2021), 31 pages. https://doi.org/10.1145/3485542

1 INTRODUCTION

Static bug checkers have become an essential tool for many developers. For example, Vassallo et al. [2020] show that many Open Source developers run tools like FindBugs [Ayewah et al. 2008] (now SpotBugs), Checkstyle [Burn et al. 2021], PMD [Copeland 2005], SonarQube [Brandhof et al. 2014], and Error Prone [Aftandilian et al. 2012] on a daily basis for quality assurance. Moreover, they found that 66% of the Open Source projects that they surveyed mandate that contributors must

Authors' addresses: Alexandru Dura, Dept. of Computer Science, Lund University, Box 118, Lund, 221 00, Sweden, alexandru.dura@cs.lth.se; Christoph Reichenbach, Dept. of Computer Science, Lund University, Box 118, Lund, 221 00, Sweden, christoph.reichenbach@cs.lth.se; Emma Söderberg, Dept. of Computer Science, Lund University, Box 118, Lund, 221 00, Sweden.

2475-1421/2021/10-ART165
https://doi.org/10.1145/3485542

use static checkers in some form. Checker frameworks typically offer a selection of bug pattern detectors, and then provide extension mechanisms through which developers can add more checks. Examples of such specialized checks include project-specific language idioms and coding standards as well as bug patterns connected to a specific API. Developers who build custom detectors mostly write plugins in an imperative general-purpose language to examine some program representation that is specific to the checker framework, e.g., an abstract syntax tree or a byte code representation.

However, given the properties of imperative specifications, these checker implementations are *black-box detectors* in that the checker framework knows little about the internal wiring of these detectors beyond how to run them. Thus, the framework cannot examine the internals of each detector to e.g. analyze the version of the input language that the checker can handle or the types of bugs that might report. It must instead rely on static and manually curated meta-information or on dynamic (and incomplete) sampling. We argue that black-box bug checker architectures limit innovation at the architectural level, and propose *clear-box bug checker frameworks* as an alternative. Clear-box architectures enable the checker framework to *automatically* enhance detectors, e.g. by deriving confidence scores [Raghothaman et al. 2018] to rank bug reports, tracing the detector's chain of reasoning to produce explanations [Zhao et al. 2020], or incrementalizing computations to speed up execution [Szabó et al. 2018]. By contrast, a black-box checker framework must ask each detector developer to manually integrate such features, requiring $O(n)$ effort over the number of detectors as opposed to $O(1)$ for clear-box frameworks. Notably, no established black-box checker framework (to the best of our knowledge) supports any combination of the above features.

In this paper, we explore the value of clear-box bug detection in JavaDL, a novel bug checker framework for Java that can process a wide range of user-specified bug detectors and execute them both incrementally and exhaustively. Incremental evaluation can often outperform exhaustive evaluation by re-using intermediate results, but for larger changes, the cost of retaining intermediate results may be greater than the benefits from re-use. JavaDL follows Raghothaman et al. [2018] and Szabó et al. [2018] in building on Datalog [Ceri et al. 1989], a declarative logic programming language that has become a popular choice for implementing analyses over powerset lattices, including complex points-to analyses [Balatsouras and Smaragdakis 2016]. Datalog-based analyses have two parts: (1) *fact extraction*, which today is generally a black-box analysis that maps the input program to a set of *input facts* (e.g., Def-Use edges [Heo et al. 2019]), and (2) *fixpoint computation*, which runs the actual Datalog program on the input facts. JavaDL minimizes its dependency on black-box fact extractors by providing syntactic pattern matching on the input abstract syntax tree (AST) for the full Java 8 syntax. This pattern matching support eliminates much of the *interoperability* concern that Madsen et al. [2016] observed for Datalog in program analysis, i.e., the need for "*tedious mapping*" and manual (de)serialization code in fact extraction.

For convenience, JavaDL additionally provides detectors with a small but extensible set of facts (name and type analysis information) from the ExtendJ Java compiler [Ekman and Hedin 2007].

We take JavaDL as one point in the design space for clear-box static checkers and explore how it compares to contemporary static checker tools with regard to expressiveness and performance. Our experiments show that JavaDL offers performance comparable to that of existing checkers and enables the concise implementation of different types of bug detectors, in the common, broad sense of the term [Reichenbach 2021], i.e. including both syntactic "bad smell" detectors and semantic, inter-procedural detectors that reason across compilation units. Our results also show that JavaDL can effectively incrementalize these checkers, but that incremental analysis may not always outperform exhaustive analysis. Our contributions in this paper are the following:

- the first (to the best of our knowledge) static checker framework with support for syntactic pattern matching over the full Java grammar (up to Java 8),

```
1 EDB('AnalyzedFiles, "AnalyzedFiles.csv", "csv").
2 java ('AnalyzedFiles) {
3   Warning(ls, cs, le, ce, f) :- CovariantEquals(c), NOT(ObjectEquals(c)), SRC(c, ls, cs, le, ce, f).
4   CovariantEquals(c) :- c ⟨: class #_ { .. public boolean equals(#t #_) { .. } .. } :⟩, DECL(#t, c).
5   ObjectEquals(c) :- c ⟨: class #_ { .. public boolean equals(Object #_) { .. } .. } :⟩.
6 }
7 OUTPUT('Warning, "Warnings.csv", "csv").
```

Fig. 1. Covariant equals check implemented in JavaDL

- a Datalog language extension for syntactic pattern matching that extends over our earlier work [Dura et al. 2019] by handling both syntactic ambiguity and semantic information,
- an incremental evaluator for inter-procedural Datalog-based program analysis,
- a prototype implementation of our approach, the JavaDL system,[1]
- a comparison of performance and quality between JavaDL, SpotBugs, and Error Prone, with a validated [Dura et al. 2021a] and an improved artifact [Dura et al. 2021b] for reproducibility.

2 THE JAVADL LANGUAGE

As an introduction, consider the Covariant `equals()` bug pattern, shared by the FindBugs and Error Prone checkers. In Java, all objects inherit the method `equals(Object)` for equality checking. If class $C \neq$ `Object` wants to provide custom equality tests, it must *override* `Object.equals(Object)` with its own C`.equals(Object)` method. Note that Java requires that C`.equals` must accept *any* `Object` parameter. However, programmers may miss this requirement and instead define a method that only accepts a subtype of `Object`, e.g. C`.equals(`C`)`. Since Java also supports *overloading*, this method may work as expected in some contexts but cause subtle bugs elsewhere.

Figure 1 shows a complete JavaDL specification that includes a detector for this pattern. For now, we skip over the input and output handling and focus on the detector itself, in lines 3–5.

Here, line 3 declares that the checker records a Warning at source location ⟨`ls`, `cs`, `le`, `ce`, `f`⟩ (in order: start line & column, end line & column, source file) whenever class `c` defines an `equals(c)` method (CovariantEquals, line 4) but no `equals(Object)` method (ObjectEquals, line 5).

The code in this line is a standard Datalog Horn clause, $P_0(\overline{x_0})$:- $P_1(\overline{x_1}), \ldots, P_n(\overline{x_n})$., where P_i are predicate symbols and $\overline{x_i}$ are sequences of variables and constants. The semantics of these clauses are that for any assignment ρ that maps all variables in $\overline{x_0}, \ldots, \overline{x_n}$ to constants, whenever for all $i \in \overline{1..n}$ the relations P_i contain the tuple $\rho(\overline{x_i})$, the tuple $\rho(\overline{x_0})$ must also be in the relation P_0. We call $P_0(\overline{x_0})$ the *head literal* and the other $P_i(\overline{x_i})$ the *body literals* and follow most Datalog dialects in allowing body literals to be negated (with the obvious semantics). When $n \neq 0$, the clause is also called a *rule*, and if $n = 0$, it is called a *fact* and usually omits the left arrow ':-'.

Line 4 shows how JavaDL extends clauses with pattern matching to define CovariantEquals:

```
c ⟨: class #_ { .. public boolean equals(#t #_) { .. } .. } :⟩
```

This pattern captures all class declarations `c` that declare an `equals` method with a single parameter. The pattern matching braces ⟨: · · · :⟩ contain Java source code, extended with *syntactic metavariables* such as `#t`, which here binds to the type name of the formal parameter to `equals`, and *wildcards* (`#_`) that we here use to ignore the name of that same parameter. The *gaps* (`..`) in the pattern ignore *sequences* of program elements: here, they allow any number of declarations to precede or follow the declaration of `equals`, and any sequence of statements inside the `equals` method body.

This pattern binds `#t` and `c` to AST nodes in the Java input program. To check if `c` defines a method `equals(c #_)`, we must also ensure that `c` and `#t` describe the same type. JavaDL represents each type by the type's declaration AST node, so `c` already *is* a type, but `#t` is only a

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

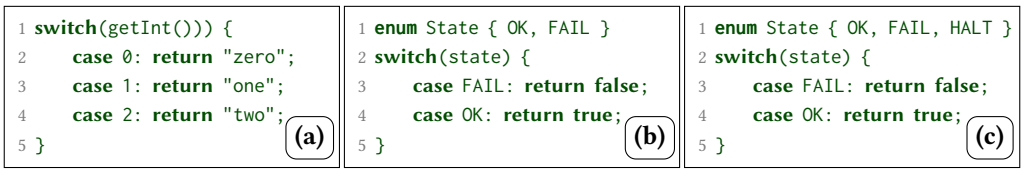


Fig. 2. Examples for the Missing **switch** Default bug pattern. Code box (a) lacks a default case. Box (b) does not, since it covers all possible **enum** cases explicitly. Box (c) uses the same **switch** block as box (b) but extends the **enum** with (HALT), meaning that the **switch** again lacks one case.

name and could resolve to different types in different contexts. We resolve its type with the *semantic predicate* DECL and ensure that that type is c by requiring that DECL(#t, c) must hold.

ObjectEquals in line 5 analogously captures all classes c that override Object.equals(Object).

We have now seen the essence of how JavaDL combines Datalog, syntactic patterns, and semantic predicates like DECL and SRC to analyze source code. JavaDL provides some additional features for handling input and output, which we demonstrate in the remaining parts of Figure 1.

Returning to the top of our example, line 1 specifies that the relation AnalyzedFiles contains all tuples from an external data source (a .csv file). Line 2 then starts a java block over AnalyzedFiles. This block sets the scope for pattern matching and semantic predicates in lines 3–5: our bug detector will analyze all the source files listed in the file AnalyzedFiles.csv. Finally, line 7 declares that all tuples in the relation Warning must be written to the file Warnings.csv.

2.1 Non-Local and Semantic Analyses

From the perspective of a typical imperative checker framework, the properties we describe in Figure 1 amount to boolean checks on Java classes. To see how JavaDL expresses more complex semantic properties, consider the Missing **switch** Default bug pattern. Figure 2a shows a **switch** statement over int values that only captures three cases; this code may indicate that the programmer forgot to handle other possible cases. Both Error Prone and SpotBugs flag such code, and JavaDL can capture this case with syntactic pattern matching and a small amount of logical reasoning.

However, relying purely on syntactic reasoning leads to a false positive for Figure 2b. This **switch** over an enumeration is exhaustive and therefore requires no **default**. SpotBugs and Error Prone will not flag the **switch** in Figure 2b, but *will* flag the nonexhaustive **switch** in Figure 2c.

To distinguish these three cases, we need to know that variable state is an **enum** value and what values it can take. That means that we need state's static type, and find that type's possible **enum** values (e.g., FAIL and OK), which can be *non-local information* if **enum** State resides in a different compilation unit. (More precise analyses are also conceivable, cf. Section 5.2.1.)

Figure 3 shows how we can specify this bug pattern in JavaDL. Predicate Switch (line 1) matches all **switch** statements, while SwitchHasDefault(s) (line 2) matches only those who contain **default** clauses. Next, CaseOnEnum(s, τ, m) (line 3) captures all **case** branches for **enum** member m, plus the surrounding **switch** statement s and **enum** type τ. Here, DECL(#c, m) maps **enum** values from Figure 2 to their declarations, while TYPE(#v, τ) relates the expression #v that we are discriminating over to its type τ, which is an **enum** type iff $\tau\langle\!:$**enum** #_ { .. }$:\!\rangle$ holds.

CaseOnEnum(s, τ, m) thus uses both semantic and non-local information: DECL and TYPE provide the results of ExtendJ's name and type analysis, respectively. TYPE(e, τ) relates expressions e and their Java types τ, and we follow ExtendJ in identifying τ with class, interface, or enum definitions for all types for which we can find one in a source or .class/.jar file. In the latter case, we use ExtendJ's partial decompilation facilities to map the high-level bytecode class structure

```
1  Switch(s) :-            s ⟨:switch (#_) { .. }:⟩.
2  SwitchHasDefault(s) :- s ⟨:switch (#_) { .. default: .. }:⟩.
3  CaseOnEnum(s, τ, m) :- s ⟨:switch (#v) { .. case #c: .. }:⟩, DECL(#c, m) TYPE(#v, τ), τ ⟨:enum #_ { .. }:⟩.
4  EnumMember(τ, #m) :-
5      τ ⟨:enum #_ { .., #m , .. ; .. }:⟩, ID(#m, _). // Filter out declarations #m that are not enum values
6  SwitchWithoutEnumMember(s, τ) :-
7      CaseOnEnum(s, τ, _), EnumMember(τ, m), NOT(CaseOnEnum(s, τ, m)).
8  SwitchOnAllEnumMembers(s) :-
9      CaseOnEnum(s, τ, _), NOT(SwitchWithoutEnumMember(s, τ)).
10 SwitchWithoutDefault(s) :-
11     Switch(s), NOT(SwitchHasDefault(s)), NOT(SwitchOnAllEnumMembers(s)).
```

Fig. 3. JavaDL specification for the SwitchNoDefault bug pattern, excluding input and output handling.

(excluding method bodies) into an AST representation. For example, when applied to Figure 2b, CaseOnEnum would contain $\langle s, \texttt{State}, \texttt{OK}\rangle$ and $\langle s, \texttt{State}, \texttt{FAIL}\rangle$, where s is the **switch** block in line 2, and `State`, `OK`, and `FAIL` are the declaration sites of **enum** type `State` and its values `OK` and `FAIL`, respectively. These semantics are independent of whether `State` is defined in the same source file as the **switch** statement, in a different source file, or in Java bytecode.

Continuing our example, predicate EnumMember(τ, #m) in lines 4–5 relates **enum** types τ and their member values #m, while SwitchWithoutEnumMember(s, τ) checks whether switch s over **enum** type τ is missing at least one enum value (m, in line 7), SwitchOnAllEnumMembers(s) checks that SwitchWithoutEnumMember does not hold for s, and SwitchWithoutDefault(s) finally collects all **switch** statements that neither have a default nor cover all **enum** members.

Non-local bug patterns like SwitchWithoutDefault are the interesting cases for incremental analysis: if the code changes in either the **switch** statement or in the **enum** type, we must re-evaluate all potentially affected reports, ideally without re-evaluating the unaffected ones (Section 4).

2.2 Language Definition

Figure 4 summarizes the JavaDL syntax. Our language is based on Datalog with negation, where literals in rule bodies may be negated NOT(P(...)), following the standard ("stratification") requirement that they must not transitively depend on their own negations (e.g., P(x) :- NOT(P(x)).). JavaDL extends Datalog with `java` blocks, syntactic patterns, and predicate references to support syntactic pattern matching.

Syntactic patterns allow us to match terms in a given input program. JavaDL surrounds syntactic patterns J by quotes $\langle: J :\rangle$ that we write <: J :> in the source code. Here, J is any Java code fragment that can be rooted at a single AST node; these can be statements, classes, types, import statements or any other syntactic production from the Java grammar. For instance, the pattern $\langle:$ `#e.toString()` $:\rangle$ will match any calls to the method `toString()` in the input program and bind the expression on which the call is made to the pattern metavariable #e. Pattern metavariables like #e describe AST nodes, and we can use them outside of syntactic

Program	$::= \overline{D_i}$
Declaration	$D ::= C \mid \texttt{java}(r)\{\overline{C_i}\}$
Clause	$C ::= \overline{H_i} \texttt{ :- } \overline{B_j}.$
Head literal	$H ::= P(\overline{t_i})$
Body literal	$B ::= H \mid \text{NOT}(H) \mid \langle: J :\rangle \mid v \langle: J :\rangle$
Term	$t ::= v \mid \_ \mid k \mid r \mid e$
Predicate reference	$r ::= \texttt{'}P$
Expression	$e ::= v \mid \texttt{to\_number}(e) \mid e\texttt{+}e \mid e\texttt{*}e \mid \ldots$
Variable	$v ::= v_b \mid v_m$
Predicate symbol	$P \in \mathit{PredSym}$
Constant	$k \in \mathbb{Z} \cup \mathit{String}$
Pattern language	$J \in \mathit{JavaPatternLanguage}$
Basic variable	$v_b \in \mathit{Var}$
Pattern metavar.	$v_m \in \#v_b$

Fig. 4. Syntax for the JavaDL language.

patterns as normal Datalog variables. In some cases we are interested in reasoning about the root of a syntactic pattern explicitly. JavaDL's *rooted matching* syntax expresses e.g. the set of occurrences of the literal 0 in the input program as `z⟨:0:⟩`; as usual in logic programming, we can use this pattern both to test whether `z` is an AST node for a `0` and to find all `z` with that property. Using pattern metavariables for rooted matching allows us to recurse, e.g. to conservatively underapproximate the set of all expressions that evaluate to `0`:

```
Zero(#z) :- #z⟨:0:⟩.
Zero(#z) :- #z⟨:0 + #z2:⟩, Zero(#z2). // And so on.
```

We can thus derive complex semantic properties directly from syntactic patterns.

When matching patterns, we often want to leave parts inside the pattern unspecified; we can do so by using gaps (`..`). For example, in Figure 1 we used gaps to ignore declarations inside class bodies, since these were irrelevant to our analysis. We also allow gaps within sequences of elements. For instance, the pattern `⟨:new #T[] { #first , .. }:⟩`, will match array initializers with at least one (but possibly more) elements, and bind the first element to `#first`.

Meanwhile, *predicate references* quote a predicate, e.g. `'Srcs`, and turns it into a logical object, akin to a pointer to a C variable or a `java.lang.Class` object. We use this mechanism to describe properties *about* predicates, and especially to expand predicates that describe a set of source files into ASTs through a `java` block:

```
java('Srcs) { . . . body . . . }
```

All syntactic patterns in the *body* of this block will reason about precisely the ASTs of the programs whose sources are in Srcs. This design gives us a scope for syntactic patterns without having to make AST roots explicit everywhere; we can think of `java` blocks as "broadcasting" implicit parameters to all rules in the body. This design follows our earlier MetaDL `analyze` blocks [Dura et al. 2019], adding rooted patterns and support for syntactic ambiguity (Section 3.3.3). While `java` blocks are not essential to our approach and could be implicit, we include them to enable future use cases e.g. for pre-analyses that grow the list of source files to handle dynamic class loading.

2.3 Pseudopredicates

JavaDL incorporates a number of language constructs with the syntactic form of predicates but special semantics. We categorize these *pseudopredicates* into three groups: *I/O pseudopredicates* for accessing external data, *infinite pseudopredicates* for arithmetic and comparison, and *semantic predicates* that expose precomputed semantic information. We have already introduced our two I/O pseudopredicates, which may only occur as facts (i.e., not in rules):

- EDB('P, `"TabulatedP.csv"`, `"csv"`) imports an external database (SQLite3 or CSV) into a relation (here: P). Like facts from `java` blocks, these imported relations provide *ground facts*, rather than derived facts; the 'E' in EDB ('extensional') alludes to this property.
- OUTPUT('R, `"TabulatedR.db"`, `"sqlite"`) exports the contents of the R predicate to the database file `"TabuletedR.db"`.

Our infinite pseudopredicates are the only predicates that allow (nested) arithmetic expressions inside their literals. We only allow these pseudopredicates in rule bodies:

- EQ(e_0, e_1) holds iff the value of e_o is equal to the value of e_1.
- LT(e_0, e_1) holds iff the value of e_0 is less than the value of e_1.
- BIND(v, e) holds iff the variable v equals the value of expression e. Unlike EQ, BIND can also bind variables.

Finally, semantic predicates provide additional information about programs under analysis. They are only available inside `java` blocks:

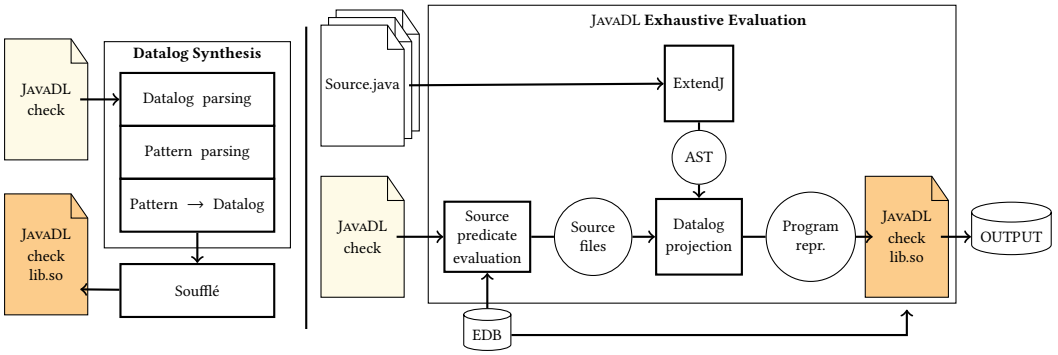


Fig. 5. Compilation of a JavaDL checker specification to a Datalog library (left) and exhaustive evaluation of a compiled JavaDL checker on a Java program (right).

- ID(n, m) relates the node n to its name m, if the node represents a named AST fragment (e.g., a variable, class or method).
- SRC$(n, l_s, c_s, l_e, c_e, f)$ relates the AST node n to its source code location, where the pairs $\langle l_s, c_s \rangle$ and $\langle l_e, c_e \rangle$ represent its start and end positions in the source file f.
- PARENT(n, c) relates the AST node c to its parent n.
- INT(n, s) relates the integer literal n to its value s, represented as a string. We provide similar predicates for the numeric literals (FLOAT, LONG, etc.).
- STR(n, s) relates the string literal n to its string value s.
- MOD(n, m) relates a declaration n to its modifiers (`public`, `static` etc.).
- DECL(n, n_d) relates the name n to its declaration n_d.
- TYPE(n, n_t) relates an AST node that represents a name or expression n to the AST node n_t that represents the node's type. Due to limitations in our frontend, there are corner cases in Java 8 where this relation may be inaccurate.

We did not find it necessary to add a feature for referencing specific Java types. As we show later in Figure 9 for `java.lang.String`, the existing JavaDL features suffice for this task.

The TYPE and DECL relations allow JavaDL code to refer to code from external `jar` files, though pattern matching support on such compiled code excludes statements and expressions.

Types. JavaDL is a statically typed language, without explicit type declarations. Instead, it relies on monomorphic type inference. Currently, JavaDL supports four types for variables: *Int*, *String*, *ASTNode* for AST nodes, and *PredRef* for predicate references.

3 IMPLEMENTATION

JavaDL can run bug checkers either exhaustively (on the entire program) or incrementally (only on the parts of a program that have changed). We first describe our implementation for exhaustive evaluation, and discuss the refinements for incremental evaluation in Section 4.

3.1 Architecture

Figure 5 shows an overview of our prototype implementation, with an initial *synthesis* step illustrated to the left and the steps of the *exhaustive evaluation* to the right.

In Datalog synthesis, our prototype compiles JavaDL into efficient native checker code with the help of the high-performance Datalog compiler Soufflé [Scholz et al. 2016]. In this step, our compiler first parses the Datalog fragment of the JavaDL program using an LALR parser, but leaves syntactic

patterns (⟨: . . . :⟩) unprocessed. It calls a separate parser to parse these (often highly ambiguous) patterns (Section 3.3) and then translates them into Datalog queries (Section 3.3.2). The compiler then emits Datalog in Soufflé's Datalog dialect (adding explicit type annotations and input/output specifications), and calls Soufflé to generate the native checker library. This library implements the Datalog portion of the bug checker's exhaustive variant and is independent of the program under analysis, so we only need to re-synthesize it when we add, remove, or alter bug detectors.

In exhaustive evaluation, the driver first determines the set of Java files for each `java` block to analyze[2], then passes these files to the Java compiler ExtendJ [Ekman and Hedin 2007]. The driver then translates relevant parts of ExtendJ's Java AST representation into in-memory tables for our native checker binary ("Datalog projection"), by flattening the AST structure (Section 3.2) and tabulating semantic information (e.g., the TYPE and DECL) predicates. Finally, the driver passes these tables to the native library, which completes Datalog evaluation and reports all detected bugs.

3.2 Program Representation

Our syntactic pattern matcher operates entirely on logical relations. Thus, we compile syntactic Java patterns into special pattern-matching Horn clauses (Section 3.3) during Datalog synthesis, and translate Java source code into logical facts. Since ExtendJ's AST faithfully preserves the concrete syntax, we do not distinguish between ASTs and parse trees.

Our approach extends the relational representation of ASTs from our earlier work [Dura et al. 2019] to compactly represent ExtendJ's Java AST, together with terminal symbols, source location information, and semantic attributes (TYPE, DECL etc.). We map AST nodes to 64-bit integers, and represent the connections between them with the following three relations:

(1) AST(k : *String*, n : *ASTNode*, i : *Int*, c : *ASTNode*, t : *String*) represents a node n, of syntactic production k. If n has no children, we store the node information as a single fact AST(k, n, -1, -1, t). If k represents a terminal name or literal, then t is additionally the corresponding lexeme. Otherwise, if n is a nonterminal, c represents the ith child node.
(2) ATTR(n : *ASTNode*, a : *String*, m : *ASTNode*) holds if node n's attribute a has value m.
(3) SRC(n : *ASTNode*, l_s : *Int*, c_s : *Int*, l_e : *Int*, c_e : *Int*, f : *String*) relates node n and its source location, represented as starting and ending line and column information, and file name.

3.3 Syntactic Pattern Matching

Our JavaDL compiler translates syntactic patterns into logical literals. This delegates the heavy lifting of syntactic pattern matching to the Datalog engine and allows us to freely intersperse pattern matching and other forms of program analysis.

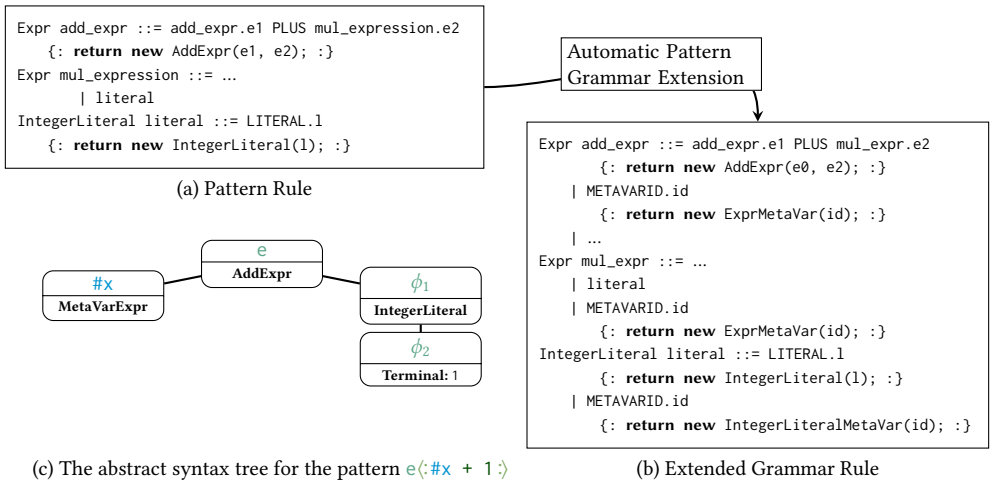
3.3.1 The Java Pattern Grammar. We construct our pattern grammar by transforming the ExtendJ Java grammar. For each syntactic category, represented by a nonterminal N, we introduce two fresh nonterminals N_p, which adds metavariables (#v), and N_l, which also adds gaps (. .):

$$\begin{array}{lcl} N_p & ::= & N \mid \mathit{MetaVarID} \\ N_l & ::= & N_p \mid \texttt{". ."} \end{array}$$

We then update all other rules in the BNF grammar to replace right-hand-side occurrences of N by N_p, except for sequences of N (e.g., parameter lists), which we replace by N_l. This process is fully automatic to support future changes to ExtendJ.

Our pattern grammar accepts any Java program but introduces ambiguities. For example, the pattern ⟨:`#t #n() { .. }`:⟩ can be parsed both as a constructor definition or as a method definition. The pattern compiler statically detects such ambiguities and emits a warning, but defaults to

[2] If a P in a `java ('P)` block is not loaded directly, we additionally use partial interpretation (not explored in this work).



(a) Pattern Rule

(b) Extended Grammar Rule

(c) The abstract syntax tree for the pattern e⟨:#x + 1:⟩

Fig. 6. Pattern grammar generation and pattern parsing.

allowing any of the possible parses. The bug detector writer can then choose to disambiguate by using other predicates in the same clause as the pattern. In the example above, the rule could e.g. test if `#t` is a modifier or a type with the literal MOD(`#t`, `_`). If the bug detector writer instead chooses to retain this ambiguity, `#t` will match both constructor and method definitions. These ambiguities only affect syntactic patterns; when we later parse Java input programs, we rely on ExtendJ and its unambiguous Java grammar.

3.3.2 Relational Representation of Patterns. We transform syntactic patterns to Datalog clauses, following our earlier work [Dura et al. 2019] and De Roover et al. [2007], except for encoding the semantic category of the matched node in tuples rather than in predicates.

To illustrate, let us first consider how ExtendJ would parse the expression `1 + 1`. Figure 6a shows the corresponding part of the ExtendJ Java grammar: ExtendJ will represent the syntactic category `add_expr` by an AST node **AddExpr** with two **`IntegerLiteral`**`(1)` child nodes.

We parse the pattern e⟨:`#x + 1`:⟩ similarly, except with the grammar in Figure 6b, which adds rules for gaps and metavariables. We represent gaps and metavariables by custom AST nodes, and assign each node a *node identifier* (Figure 6c). For metavariables and the roots of rooted patterns, we set these identifiers to be the corresponding (meta)variables. For all other nodes, we introduce a fresh variable (e.g. ϕ_1, ϕ_2).

For each such pattern p, we then create a fresh Datalog predicate $\textsc{Pat}_p$, with its arity equal to the number of free variables in the pattern, including the optional root. We then replace all occurrences of p by $\textsc{Pat}_p$ to desugar JavaDL to plain Datalog.

```
Pat(e, #x) :- AST("AddExpr", e, 0, #x, _),
              AST("AddExpr", e, 1, φ1, _),
              Supertype(κ1, "Expr"),
              AST(κ1, φ1, 0, φ2, _),
              AST("Terminal", φ2, _, "1").
```

Fig. 7. Expanded pattern.

In Figure 6c, the **MetaVarExpr** node corresponds to the syntactic category **Expr** in the abstract grammar. Since we tag our AST nodes with their syntactic production (e.g., **IntegerLiteral**) rather than with nonterminal syntactic categories, we encode their subtyping relationship in an internal predicate Supertype(*String*, *String*).

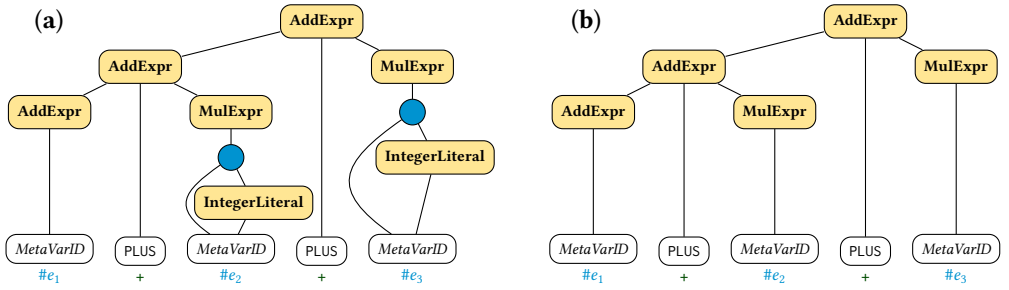


Fig. 8. The original (a) and simplified (b) SPPFs for the expression `#e1 + #e2 + #e3`. ● denotes alternatives.

We generate this predicate statically, based on the analysis of the type hierarchy in the abstract grammar, and use it to set a bound on the syntactic categories that a metavariable can match. This transforms the pattern `e⟨:#x + 1:⟩` into the Datalog rule in Figure 7. We use the same approach to translate gaps and the constraints that they introduce as in MetaDL [Dura et al. 2019].

3.3.3 Parsing Ambiguous Patterns. Our automatically generated pattern grammar (Figure 6b) is highly ambiguous, and unsuitable for most parsing algorithms. We therefore adapt Scott [2008]'s shared packed parse forest (SPPF) variant of the Earley parser. This algorithm compresses ambiguity effectively, but solves only part of the ambiguity in our case.

To illustrate, consider the pattern `⟨:#e1 + #e2 + #e3:⟩`. Parsing this pattern yields the SPPF in Figure 8a, which contains two SPPF *alternative* nodes. These are not parse tree nodes but instead mark a choice in the tree: either child of an alternative node will yield a correct parse tree.

While a purely syntactic system could match over these alternatives directly, JavaDL must map syntactic categories to AST node names (Section 3.3.2). However, ExtendJ's parser uses a black-box specification of the mapping between concrete and abstract syntax that only provides us with opaque 'parser actions' (`{: ... :}` in Figure 6) that we can call, but not inspect automatically.

To avoid having to produce all four ASTs for our example, or 2^n in general, we simplify the parse forests by shortening the path from each metavariable or gap terminal to the root. Concretely, we find all derivation chains of the form $N \longrightarrow M \longrightarrow \mathit{MetaVarID}$ and replace them with $N \longrightarrow \mathit{MetaVarID}$. We do this transformation recursively, starting from the leaves towards the root, while also merging alternatives if they contain the same derivation chain. We perform a similar transformation for gaps. For the example in Figure 8a, we apply the transformation twice for N = **MulExpr** and M = **IntegerLiteral**. The intuition behind this transformation is that instead of enumerating all the possible syntactic categories a metavariable could have, we provide an upper bound for them.

Going back to the example in Figure 8a, the metavariable `#e3` could match the syntactic categories **IntegerLiteral** or **Expr**. After the transformation (Figure 8b), `#e3` matches all the nodes having the syntactic category **Expr**, which is more general and includes **IntegerLiteral**.

In effect, the transformation merges multiple derivation chains that end up in a *MetaVarID* (or a gap) and yields an SPPF with significantly fewer edges. In practice, this enables us to quickly enumerate the parse trees. For the addition pattern discussed above, this approach produces a single parse tree (Figure 8b).

3.4 Semantic Attribute Extraction

ExtendJ is a full-fledged Java compiler, and therefore performs a variety of program analyses. Some of these are useful for writing bug detectors, so we export them for easy access in JavaDL code.

```
1 NewString(τ, f, l, c) :- n⟨:new String(#v):⟩, TYPE(#v, τ), SRC(n, l, c, _, _, f).
2 StringClass(s) :- s⟨:.. class String { .. }:⟩, SRC(s, _,_,_,_, "java/lang/String.class").
3 BadNewString(f, l, c) :- NewString(τ, f, l, c), StringClass(τ).
```

Fig. 9. Check for wasteful String construction, split into three separate predicates to simplify our exposition.

ExtendJ is implemented in the *Reference Attribute Grammar* system JastAdd [Hedin and Magnusson 2003], meaning that its program analyses are computed as attributes over AST nodes, and that these attributes may be references to other AST nodes. For example, ExtendJ computes the declaration site of a variable or a method as a reference to the AST node that contains the node's declaration, and the type of a variable as a reference to the AST node of the type's class definition. Some of these AST nodes are not AST nodes in the classical sense. ExtendJ computes *synthetic AST nodes* (higher-order attributes [Vogt et al. 1989]) on-demand to represent e.g. primitive types and classes loaded from external jar files.

Since ExtendJ runs on the Java Virtual Machine and Soufflé is a native executable, implementing on-the-fly attribute evaluation through callbacks to Java would require considerable engineering effort. We instead opted to tabulate program representation relations directly into Soufflé's data structures, using Java Native Interface calls. While tabulating the program's original AST requires only a simple tree traversal, the evaluation of attributes may produce new synthetic AST nodes that we must traverse, add to the program representation relation, and recursively re-examine to extract their attributes. We implement a fixed point algorithm here for exhaustive attribute evaluation.

JavaDL currently supports the evaluation of two attributes that may produce fresh AST nodes:

- DECL, mapping a node to the AST node declaring it, and
- TYPE, mapping an AST node to the node declaring its type

In general, the attribute extraction mechanism is easy to extend. It takes no more than 10 lines of code to expose a new attribute as a Datalog relation. This property of our implementation facilitates easy sharing of analyses between ExtendJ and JavaDL.

4 INCREMENTAL EVALUATION

When developers integrate static checkers into their workflow, these checkers often become part of continuous integration or code review [Calcagno et al. 2015; Sadowski et al. 2015]. In those scenarios, the checker runs frequently, and each run sees largely the same source code, meaning that exhaustive analysis can be wasteful.

Consider the analysis in Figure 9. This check, inspired by SpotBugs' `DM_STRING_CTOR`, looks for explicit `String` object constructions, `new String(#v)`, where `#v` is already a `String`. Since Java `String`s are immutable, this construction is inefficient. On line 1, the predicate NewString identifies source locations $\langle f, l, c\rangle$ that explicitly call a `String` constructor with a single argument of type τ. On line 2, StringClass finds the unique `java.lang.String` class. Finally, on line 3 BadNewString filters NewString to only report string constructions from existing string objects.

We see that different parts of one bug detector may need to be updated at different times: we only need to recompute StringClass if the standard library changes, but must update NewString on changed parts of the source code. Finally, we must update BadNewString if either of the other relations changes.

4.1 Incremental Architecture

Changes to source code can lead to the retraction of a previously true fact, e.g., when a developer changes the superclass of some class `C` from `A` to something else. Retractions can have consequences:

if `C` inherited `m` from `A`, it will no longer do so. However, the consequences of retractions may not be obvious: `C` might also have obtained `m` from a second source, e.g., a Java **`default`** interface method.

The literature has proposed a number of different techniques for incremental updates of Datalog-style analyses [Gupta et al. 1993; Szabó et al. 2018]. However, we found these techniques unsuitable for JavaDL: first, they assume that all ground facts are permanently stored in a database. In JavaDL, the majority of ground facts are AST nodes that we derive from source files (i.e., these facts are actually derived and "ground" only from the perspective of Datalog), and serializing these facts to disk would incur nontrivial storage cost. Second, we assign identities to these nodes based on their order in the parsed file, which means that a small change at the beginning of a file would trigger individual retractions for almost all AST nodes in that same file.

While we expect to be able to avoid the second problem through a suitably tuned AST differencing algorithm [Narasimhan et al. 2018] analogously to CFG-based differencing [Arzt and Bodden 2014], this would not address the first problem. We aimed to keep the AST in-memory as much as possible, so we use a conservative provenance tracking scheme that partitions facts by compilation unit, tracks which compilation unit partition contributed to which other partition, and propagates retractions along partition dependencies.

This partitioning scheme reflects both the level of granularity of the ASTs that we obtain from ExtendJ and JavaDL's use as a static checker: such tools typically run e.g. in Continuous Integration environments, where the degree of incrementality that they encounter is that of one or more revision control commits.

Our incremental evaluator first computes a set of *stale* files from a set of modified, added, and deleted source files, to which it transitively adds all other source files that transitively depend on stale files. It then discards data from these stale files and re-analyzes them as needed. This adds several components over the exhaustive analysis (Section 3), which we summarize in Figure 10:

(1) *Predicate Separation* (Section 4.2) sorts JavaDL predicates into *local* predicates (e.g., NewString and StringClass in Figure 9), which we can run on individual source files, and *global* predicates which can depend on multiple source files (e.g., BadNewString in Figure 9).
(2) *Attribute provenance tracking* (Section 4.3) tracks source file dependencies.
(3) The *Intermediate Result DataBase (IRDB)* (Figure 10) caches local predicates, tagged with the source file from which we have derived them, as well as attribute provenance.

Moreover, we split evaluation into two phases:

(1) Local predicate evaluation, whose results (P_L) we cache in the IRDB, and which we only re-run on stale files, and
(2) Global predicate evaluation, which we always run once at the end, which depends on all local predicates and obtains them either from the cache (P_C) or from in-memory data from the current run (P_L) (Section 4.2.4).

4.2 Separating Local and Global Rules

Consider again Figure 9: as we argued above, predicate NewString and the rule defining it refer to information from a single source file. However, that claim is not entirely obvious. While the syntactic pattern only connects nodes from the same compilation unit (n and #v), the pair $\langle \#v, \tau \rangle$ from TYPE may refer to any type τ in any compilation unit. However, we can still evaluate this rule locally: if we replace TYPE with TYPE_f, a subset of TYPE that only knows the type information of all AST nodes in file f, and set f to be the file that contains n, then $\langle \#v, \tau \rangle \in \text{TYPE}_f \iff \langle \#v, \tau \rangle \in \text{TYPE}$. We can apply the exact same reasoning to SRC in the rule for NewString, and to the body of the StringClass rule.

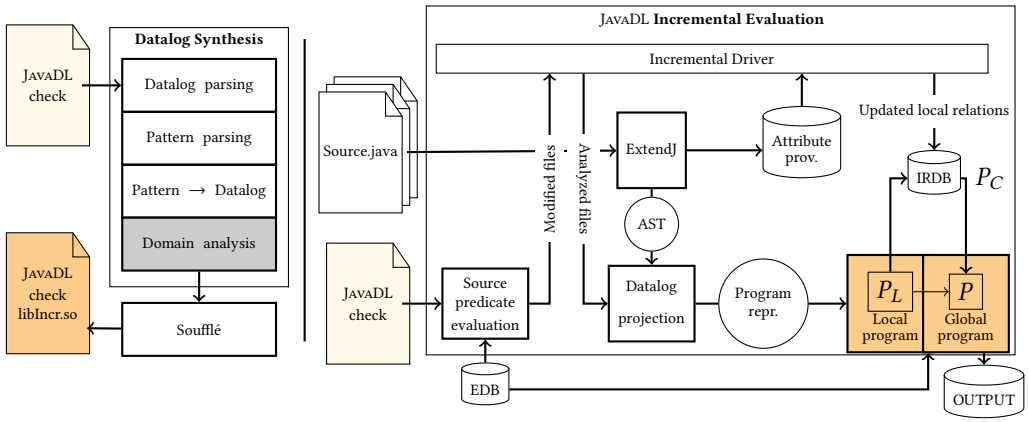


Fig. 10. Incremental evaluation of a JavaDL analysis on a Java program.

However, BadNewString may combine tuples from distinct compilation units: $\langle \tau, \mathtt{f}, \mathtt{l}, \mathtt{c} \rangle$ from any compilation unit that called the `String` constructor, and $\langle \tau \rangle$ from `java/lang/String.class`. Hence, we must compute BadNewString during global evaluation.

We formalize these intuitions in the following, but first note that the separation between local and global predicates is only relevant for predicates with variables of type *ASTNode* or predicates that depend on such rules. In practice, we have encountered user-defined predicates without *ASTNode* variables only as (short) block- or passlists/allowlists, so we do not consider them further here.

For our formalization, we introduce three concepts:

(1) *Domain separation signatures* (DSS) (Section 4.2.1), which capture which of the (*ASTNode*) variables of a predicate or a rule must always be in the same compilation unit. For example, the DSS for the rule body of NewString partitions the variables $\{\mathtt{n}, \#\mathtt{v}, \tau\}$ must separate $\{\mathtt{n}, \#\mathtt{v}\}$ from and $\{\tau\}$, since τ may come from a different compilation unit.
(2) The *local variable set* (Section 4.2.3), which is the set of variables in a rule or parameters in a predicate that represent the compilation unit for which we can evaluate the predicate or rule locally. For example, we can evaluate the rule for NewString in the compilation unit represented by $\{\mathtt{n}, \#\mathtt{v}\}$, but not the one represented by $\{\tau\}$.
(3) *AST predicates*, which are predicates that (transitively) depend on AST facts.

4.2.1 Domain Separation Signatures. Domain Separation Signatures (DSS) describe, for both rules and predicates, which (*ASTNode*) parameters to the head literal must come from the same compilation unit. We denote DSS as partitions over the variables in each rule body:

$$R_1 = \mathtt{n}(\text{:}\ \texttt{new String(\#v)}\ \text{:}\rangle \qquad \mathrm{DSS}(R_1) = \{\{\mathtt{n}, \#\mathtt{v}\}\}$$
$$R_2 = \mathtt{n}(\text{:}\ \texttt{new String(\#v)}\ \text{:}\rangle, \mathrm{TYPE}(\#\mathtt{v}, \tau) \qquad \mathrm{DSS}(R_2) = \{\{\mathtt{n}, \#\mathtt{v}\}, \{\tau\}\}$$

and analogously for predicates, where we partition by parameter index (Figure 11).

Whenever the DSS separates two variables, we may need to defer their binding to global predicate evaluation (with some exceptions, cf. Section 4.2.3). We thus want a DSS that is as *coarse* as possible:

Definition 4.1. Let N be a finite set and σ_1, σ_2 two partitions of N. We say that the partition σ_1 is coarser than σ_2 and we write $\sigma_2 \sqsubseteq \sigma_1$ iff $\forall x. \forall y. x \sigma_2 y \Rightarrow x \sigma_1 y$.

At the same time, we require that in each partition of a DSS, all variables in that partition must always bind to AST nodes in same compilation unit. To find the coarsest possible DSS that satisfies this requirement, we construct a DSS lattice:

```
P(n, #v) :- n⟨:new String(#v):⟩                                  DSS(P) = {{1, 2}}
TYPE(n, τ) :- ....                                               DSS(TYPE) = {{1}, {2}}
SRC(n, _, _, _, _) :- ....                                       DSS(SRC) = {{1}}
NewString(τ, _, _, _) :- P(n, #v), TYPE(#v, τ), SRC(n, _, _, _). DSS(NewString) = {{1}}
StringClass(s) :- s⟨:.. class String { .. } ..:⟩                 DSS(StringClass) = {{1}}
```

Fig. 11. Domain Separation Signatures for predicates. 1 refers to the first parameter, 2 to the second etc.

Definition 4.2. We define the join ($\sqcup$) and meet ($\sqcap$) for partitions over N as follows, and note the resultant top (coarsest and trivial) as well as bottom (most refined) partitions:

$$\begin{array}{lll} x(\sigma_1 \sqcup \sigma_2)y & \Leftrightarrow & \exists z. x\sigma_1 z \wedge z\sigma_2 y \qquad \top = \{N\} \\ x(\sigma_1 \sqcap \sigma_2)y & \Leftrightarrow & x\sigma_1 y \wedge x\sigma_2 y \qquad \bot = \{\{n\} | n \in N\} \end{array}$$

Definition 4.3. Let P be a predicate of type $(\tau_1, ..., \tau_n)$ and $N = \{i | \tau_i = \mathit{ASTNode}\}$. The domain separation signature of predicate P is the coarsest partition σ of N such that if $i\sigma j$ then for all tuples $\langle v_1, \ldots, v_n \rangle \in P$, v_i and v_j must be bound to AST nodes from the same compilation unit.

For a predicate P we denote by $\sigma_{\mathrm{P}}^{\overline{x}}$ the equivalence relation induced by DSS(P) on all $\overline{x} = x_1, \ldots, x_k$. Let $\sigma_{\mathrm{P}} = \mathrm{DSS}(\mathrm{P})$. Then $i\sigma_{\mathrm{P}} j$ implies that any values bound to x_i and x_j must always come from the same compilation unit, so we define $x_i \sigma_{\mathrm{P}}^{\overline{x}} x_j \iff i\sigma_{\mathrm{P}} j$

We now conjunctively combine induced constraints for Datalog rule bodies of the form

$$R = \mathrm{P}_1(\overline{x_1}), \ldots, \mathrm{P}_n(\overline{x_n}), \mathrm{NOT}(\mathrm{P}_{n+1}(\overline{x_{n+1}})), \ldots, \mathrm{NOT}(\mathrm{P}_{n+m}(\overline{x_{n+m}}))$$

Definition 4.4. The domain separation signature of a rule body R is the partition σ_R of the set V of variables of type *ASTNode* in the rule such that

$$\sigma_R = \bigsqcup_{k=1,n} \sigma_{\mathrm{P}_k}^{\overline{x_k}}$$

4.2.2 Inferring Domain Separation Signatures. Since predicate definitions may have recursive dependencies, the above definitions are not yet sufficient for inference. We begin inference by observing that the DSS for syntactic patterns is always trivial (all variables are in the same compilation unit), and similarly for semantic predicates, e.g. $\mathrm{DSS}(\mathrm{ID}) = \{\{1\}\}$. The exceptions are $\mathrm{DSS}(\mathrm{TYPE}) = \mathrm{DSS}(\mathrm{DECL}) = \{\{1\}, \{2\}\}$.

As a converse to our earlier definition, the equivalence relation σ_R on the variables in a rule induces an equivalence relation on the head literal, $\mathrm{P}(\overline{x})$, where we define $i\sigma_{\mathrm{P}[R]}^{\overline{x}} j \iff x_i \sigma_R x_j$ with x_i and x_j in $\overline{x}$. We compute the signatures of the predicates as a fixpoint, starting with $(\sigma_{\mathrm{P}})_0 = \top$:

$$(\sigma_P)_{k+1} = (\sigma_P)_k \sqcap \bigsqcap_{\mathrm{P}(\overline{x}):-\mathrm{R}} \sigma_{\mathrm{P}[R]}^{\overline{x}}$$

The number of iterations is bounded by the height of our (finite) lattice.

Definition 4.5. We say that a DSS σ_{P} is sound iff after *exhaustive* evaluation of the JavaDL program that contains P, whenever $\langle n_1, \ldots, n_k \rangle \in \mathrm{P}$ and $i\sigma_{\mathrm{P}} j$ with $i, j \in \{1, \ldots, k\}$, we always have that n_i and n_j are from the same compilation unit. We extend this definition to σ_R.

Theorem 4.6. *For any* JavaDL *program, DSS inference is sound for all P.*

Proof. Sketch: Assume that $i\ \sigma_{\mathrm{P}}\ j$, but exhaustive evaluation derives $\overline{n} = \langle n_1, \ldots, n_k \rangle \in \mathrm{P}$ s.th. n_i and n_j are AST nodes from different compilation units. Considering the Datalog horn clauses as sequents, we show by coinduction that any proof for $\mathrm{P}(\overline{n})$ must be infinite, which is a contradiction.

First note that P cannot be a built-in predicate, whose DSS are trivially sound. By fixpoint construction of σ_{P}, we must then have for all rule bodies R with $\mathrm{P}(\overline{x})$:- R that $x_i\ \sigma_R\ x_j$. Let

$$R = \mathrm{P}_1(\overline{y_1}), \ldots, \mathrm{P}_n(\overline{y_{n+1}}), \mathsf{NOT}(\mathrm{P}_{n+1}(\overline{y_{n+1}})), \ldots, \mathsf{NOT}(\mathrm{P}_{n+m}(\overline{y_{n+m}}))$$

We can ignore the negated literals, as they do not contribute to DSS and can at most remove tuples. Observe that $y\ \sigma_{\mathrm{P}}^{\overline{x}}\ z$ only holds if y and z are in $\overline{x}$, so $x_i\ \sigma_R\ x_j$ iff there exists a positive literal $\mathrm{P}_k(\overline{y_k})$ with $x_i\ \sigma_{\mathrm{P}_k}^{\overline{y_k}}\ x_j$. Since $x_i \neq x_j$, that literal cannot be built-in: if $\mathrm{P}_k = \textsc{Parent}$ (or similar local AST traversal), n_i and n_j are in the same compilation unit (contradiction), while for TYPE and DECL the DSS does not allow $x_i\ \sigma_{\mathrm{P}_k}^{\overline{y_k}}\ x_j$. By elimination, P_k must be a user-defined predicate. Let $\overline{y_k} = y_0, \ldots, y_m$ s.th. $x_i = y_a$ and $x_j = y_b$. Then P_k is unsound, because there exists $\overline{v} = \langle v_1, \ldots, v_m \rangle \in \mathrm{P}_k$ with $v_a = n_i, v_b = n_j$. Hence v_a and v_b are from different compilation units, but $\mathrm{DSS}P_k)$ claims $a\ \sigma_{\mathrm{P}_k}\ b$. By coinduction hypothesis, the derivation of $\mathrm{P}_k(\overline{v})$ is infinite. □

4.2.3 Local Variable Set. While the DSS tells us which variables are always in the same compilation unit, that information is not sufficient for telling whether we can evaluate a rule locally. For instance, consider the following rules, both of which have the DSS $\{\{1\}, \{2\}\}$:

$$\begin{aligned} \mathrm{Q}(\#\mathsf{n}, \tau) &\text{ :- } \mathrm{TYPE}(\#\mathsf{n}, \tau), \langle\!: \texttt{.. class C \{ .. \#n .. \}} :\!\rangle. \\ \mathrm{U}(n, \tau) &\text{ :- } \mathrm{TYPE}(n, \tau), \tau \langle\!: \texttt{.. class C \{ .. \}} :\!\rangle. \end{aligned}$$

Q finds all fields and methods #n and their types τ in any class named C, while U finds all expressions and declarations n whose type is any class named C. Q we can compute locally, but not U: during incremental evaluation, TYPE only knows the data that we extract from ExtendJ for the compilation unit under analysis, i.e., the types of local nodes, but no external references to local nodes.

Definition 4.7. For a predicate P, we define the local variable set as $\mathsf{LV}(\mathrm{P}) \in \mathsf{DSS}(\mathrm{P})$ such that we can evaluate P locally iff all variables in $\mathsf{LV}(\mathrm{P})$ are bound to local AST nodes.

As we saw with predicates Q and U, we can evaluate TYPE locally only if parameter 1 is bound to a local AST node. Thus, the local variable set of TYPE is $\{1\}$, while for P from Figure 11, the local variable set consists of all its variables, $\{1, 2\}$.

Analogously to domain separation signatures, each predicate induces a local variable set. So if $i \in \mathsf{LV}(\mathrm{P}_k)$ and $L = \mathrm{P}_k(\ldots, x_i, \ldots)$ is a literal, then $x_i \in \mathsf{LV}(L)$ (analogously for negated literals).

Definition 4.8. For a rule $R = L_0$:- $L_1, \ldots, L_k$, we define the local variable set $\mathsf{LV}(R)$ as the element of $\mathsf{DSS}(R)$ that contains at least one local variable from all literals that have a local set, i.e.,

$$\mathsf{LV}(R) = S \in \mathsf{DSS}(R) \text{ s.t. for all } L_i \in \{L_1, \ldots, L_k\}, \mathsf{LV}(L_i) = \emptyset \text{ or } \mathsf{LV}(L_i) \cap S \neq \emptyset.$$

If such an element exists, R is a *local rule*, otherwise we set $\mathsf{LV}(R) = \emptyset$.

Corollary 4.9. *If $\mathsf{LV}(R) = S$ is nonempty, then by construction, all literals in R depend only on AST nodes from the same compilation unit.*

Returning to our example in Figure 9 and labeling the rule bodies on lines 1–3 as R_1, R_2, R_3, respectively, we have the local variable sets: $\mathsf{LV}(R_1) = \{\mathsf{n}, \#\mathsf{v}\}$, $\mathsf{LV}(\textsc{NewString}) = \emptyset$, $\mathsf{LV}(R_2) = \{\mathsf{s}\}$, $\mathsf{LV}(\textsc{StringClass}) = \{\mathsf{s}\}$ and $\mathsf{LV}(R_3) = \emptyset$. R_1 and R_2 are therefore local rules.

4.2.4 Incrementalizing by Rule Localization. If all rules R for a predicate P are local, we consider P a *local predicate*. All other predicates are *global predicates*. For each local P, we introduce two helper predicates P_L and P_C. P_L contains the tuples of P that we compute in the current evaluation pass, while P_C contains the tuples from cached compilation units. We augment P_C and P_L with an additional marker to store its source compilation unit.

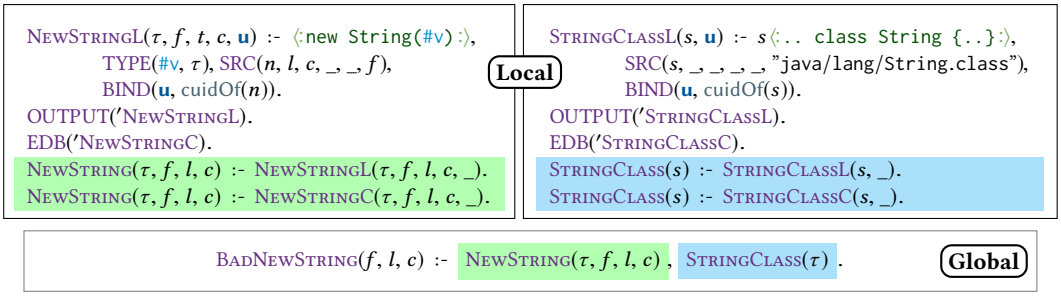


Fig. 12. Global and local rules in the incrementalized version of Figure 9. The shaded areas fuse the updated and cached results and thereby recover the same results as if we had run an exhaustive analysis.

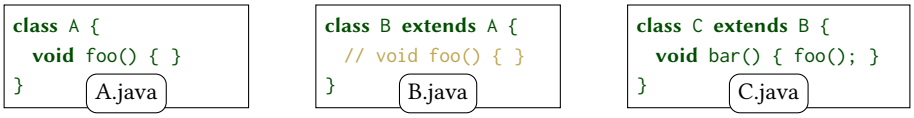


Fig. 13. Nontrivial provenance: The `foo()` call in class `C` references `A`, but also depends on `B`.

JavaDL transforms our running example (Figure 9) into the program in Figure 12. Here, **u** is the compilation unit marker, which we extract via the function cuidOf(n). Since we encode all *ASTNode* objects as unique 64 bit integers that incorporate their compilation unit identifier, cuidOf(n) has practically no overhead. The shaded areas highlight how we fuse P_L and P_C.

4.3 Attribute Provenance

JavaDL rules can reference compilation units in two ways: through pattern-matching, and through semantic predicates (DECL, TYPE). We tabulate the latter directly from ExtendJ attributes. However, individual tuples in these relations can have nontrivial provenance. Consider Figure 13. When we analyze class `C`, DECL for the call `foo()` yields the definition of `foo()` in class `A`. However, the provenance of this information also includes `B.java`, since ExtendJ must check that class `B` does not override that method, i.e., changes to `B` can affect the DECL information in `C`.

To compute attribute provenance, we trace attribute evaluation with a tracing mechanism provided by JastAdd [Söderberg and Hedin 2011], the attribute grammar framework that underlies ExtendJ. We have extended this tracing to report source file accesses in ExtendJ, and to cache summaries for attributes that JastAdd itself caches. This technique allows us to support provenance tracking for arbitrary attributes, including future extensions. Our implementation does not yet track negative dependencies, which can arise with wildcard import statements, e.g. `import java.util.*;`. Adding support for these would require some engineering effort but no changes to our conceptual framework; in our evaluation, we systematically verified that they had no impact on the bug reports.

5 EVALUATION

To understand the value of JavaDL as a proof-of-concept clear-box bug checker framework, we selected an experimental setup (Section 5.1) to explore the following questions:

- **RQ1**: *To what degree can* JavaDL *be used to specify common static checks?* (Section 5.2)
  - **RQ1.1**: *How expressive is* JavaDL *compared to other tools?*
  - **RQ1.2**: *How precise is* JavaDL *compared to other tools?*
- **RQ2**: *How does the execution time for* JavaDL *compare to the state of the art?* (Section 5.3)

– **RQ2.1**: *How does the execution time of* JavaDL *compare to other tools?*
– **RQ2.2**: *How effective is the automatic file-level incrementalization that* JavaDL *enables?*

5.1 Experimental Setup

Selection of Tools and Bug Patterns. Modern bug checkers come with hundreds of bug patterns. To select representative patterns, we turned to a recent study by Habib and Pradel [2018], who compare the SpotBugs, Error Prone, and Infer checkers on an extended version of the Defects4J [Just et al. 2014] data set[3]. The study identifies the five most commonly triggered detectors ("top 5 warnings") for each of the three selected detectors. We selected these warnings for our analysis and compared our approach against SpotBugs' and Error Prone's detector implementations. We excluded both Infer and its Top-5 detectors from our comparison, since (a) Infer's unique separation-logic based approach [O'Hearn 2019; Reynolds 2002] distinguishes it from the checkers that we listed in the introduction to such a degree that it is unclear that a comparison would generalize, and (b) its Top-5 detectors depend on flow-sensitive analyses, for which JavaDL does not currently provide special support (Section 5.2.1), so that they would have required substantial additional effort to add.

Table 1 summarizes these detectors and their implementations. The bug patterns marked with '★' are the Top-5 for SpotBugs (SB-Top-5) or Error Prone (EP-Top-5). SB-Top-5 and EP-Top-5 overlap in one detector (Boxed Primitive Constructor), for a total of nine bug detectors. For completeness, the table also lists the Covariant `equals()` detector that we discuss in Section 2.

Selection of Java Benchmarks. For our evaluation setup we adapted the framework developed by Habib and Pradel [2018] to JavaDL and updated the versions of the Defects4J data set (v2.0[4]) as well as of the checkers (SpotBugs v4.0.3, Error Prone v2.4). We used all the projects from Defects4J, except `Gson`, for which we were unable to retrieve the project properties, and `Collections` that we were not able to compile with ExtendJ. For each Defects4J project, we chose the version tagged as `D4J_`*project_id*`_FIXED_VERSION`, where *id* is the highest bug ID for *project* in the Defects4J database. We implemented two JavaDL programs, one for SB-Top-5 and one for EP-Top-5. Throughout the evaluation, we compare these two JavaDL programs directly against these recent, unaltered versions of SpotBugs and Error Prone.

Gathering Data on Analyzer Precision. To assess precision, we compare the reports of our JavaDL detectors on Defects4J against the reports produced by SpotBugs and Error Prone. We followed Defects4J's configuration in excluding the projects' unit tests from this analysis. To compare JavaDL, we utilize a notion of *relative recall and precision*:

$$recall_T = \frac{|W_{\text{JavaDL}} \cap W_T|}{|W_T|} \qquad precision_T = \frac{|W_{\text{JavaDL}} \cap W_T|}{|W_{\text{JavaDL}}|}$$

where W_T is the set of warnings reported by T. These notions of precision and recall are *relative to a checker* T, so they do not represent ground truth, but rather degree of agreement with T.

5.2 RQ1: Expressiveness and Precision

5.2.1 ***RQ1.1****: Expressiveness.* We were able to express all detectors that we implemented in JavaDL purely with Datalog-style logical rules, syntactic pattern matching, and our small set of carefully selected semantic relations. We encountered no situation that we thought would be simpler to implement imperatively, though we are of course biased.

Table 1 shows the sizes of the bug pattern implementations in Error Prone, SpotBugs, and JavaDL, in lines of code. Some implementations additionally support suggested fixes, the implementation

[3] https://github.com/rjust/defects4j/pull/112
[4] Commit `eaebff11c`

Table 1. Overview of selected bug patterns and their size in Error Prone, SpotBugs, and JavaDL. Patterns marked with ★ are among the Top-5 (Section 5.1). In column **Notes**, $a+b$ means that b of the lines implemented auto-fixups. *share n* means that the detector(s) were sharing their implementation with n additional detectors. Except for DM_NUMBER_CTOR and DM_STRING_CTOR, which were in two separate files (sharing 1 and 51, respectively), all detectors listed above (for all tools) were in a single source file each. **dataflow** means that the detector additionally used the Error Prone or SpotBugs data flow analysis engine (not counted for LOC). Column **Semantics** lists semantic predicates that the JavaDL implementations depended on (Section 5.2.1).

	Baseline Static Checker Framework				JavaDL		
	★ **Bug Pattern**	**ID**	**LOC**	**Notes**	**LOC**	**Rules**	**Semantics**
Error Prone	★ Boxed Primitive Constructor	`BoxedPrimitiveConstructor`	229	115 + 114	9	3	DECL
	★ Missing `@Override`	`MissingOverride`	84	82 + 2	48	30	DECL
	★ Useless Type Parameter	`TypeParameterUnusedInFormals`	108		27	18	DECL
	★ Complex Operator Precedence	`OperatorPrecedence`, `UnnecessaryParentheses`	138	99 + 39	37	37	
	★ == on References	`ReferenceEquality`	97	**dataflow**	88	48	DECL,TYPE
	Covariant `equals()`	`NonOverridingEquals`	132	116 + 16	15	9	DECL
SpotBugs	★ Boxed Primitive Constructor	`DM_NUMBER_CTOR`, `DM_STRING_CTOR`	1415	share 52	9	3	DECL,TYPE
	★ Expose Internal Representation	`EI_EXPOSE_REP`, `MS_EXPOSE_REP`, `EI_EXPOSE_REP2`, `EI_EXPOSE_STATIC_REP2`	138		29	20	DECL
	★ Naming Convention Violation	`NM_METHOD_NAMING_CONVENTION`, `NM_FIELD_NAMING_CONVENTION`, `NM_CLASS_NAMING_CONVENTION`	499	share 12	17	10	
	★ Missing **`switch`** Default	`SF_SWITCH_NO_DEFAULT`	289	share 4	21	8	DECL,TYPE
	★ Field Never Written To	`UWF_UNWRITTEN_FIELD`, `UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD`	1032	share 14 **dataflow**	51	47	DECL
	Covariant `equals()`	`EQ_ABSTRACT_SELF`	541	share 18	15	9	DECL

of several bug checkers may be entangled in the same implementation, and different tools may provide different library functionality (not counted here), so this comparison is not precise. With that in mind, the ratio (tool LOC / JavaDL LOC) yields a range of 1.1–12.8 (mean: 4.9) for Error Prone (when excluding LOCs for fixes) and 4.8–157.2 (mean: 38.4) for SpotBugs. If we normalize for implementation sharing across SpotBugs detectors, i.e., assume that any set of n shared SpotBugs detectors with ℓ LOC could be refactored to precisely $\frac{\ell}{n}$ LOC each, the ratio becomes 1.3–4.8 (mean 2.8), though we expect this to be an under-approximation. Either interpretation indicates that JavaDL requires fewer LOC than the other tools. Detector == on References stands out as being almost the same size for Error Prone and JavaDL. However, Error Prone's implementation utilizes a flow analysis framework whose size we did not include in the measurements. Our implementation of this detector otherwise follows Error Prone's heuristics, including checking whether the reference type defines or inherits a custom `equals()` method (28 LOC in JavaDL).

Based on manual inspection of the SpotBugs and Error Prone detectors, we believe that the main reasons for why JavaDL specifications are more concise are that (a) syntactic patterns can simplify many nested **`if`** statements in imperative code, and (b) looping and filtering is implicit in Datalog.

Limitations in Syntactic Patterns. When encoding bug patterns in JavaDL, we sometimes found that relying directly on syntactic patterns led to code duplication. For example, matching declarations `#d` with the enclosing class ⟨:**`class`** `#c {.. #d ..}`:⟩, enum ⟨:**`enum`** `#c {.. #d ..}`:⟩, or interface ⟨:**`interface`** `#i {.. #d ..}`:⟩ requires three separate rules that we cannot combine

into one pattern ⟨:`#td #c {.. #d ..}`:⟩, since the keywords **class**, **interface**, and **enum** do not themselves form a syntactic category for `#td` in the Java grammar.

Semantic Reasoning. The above limitation materializes especially when we analyze semantic properties that do not cleanly map to a single syntactic construct. For example, the patterns ⟨:`this.f`:⟩ and ⟨:`f`:⟩ are syntactically distinct, but often semantically identical. In those cases, syntactic patterns quickly become ineffective, and we found that we instead relied on Datalog relations that captured semantic properties.

As the right-most column in Table 1 shows, all but two of our detectors rely on the static name analysis information that we extract from ExtendJ, and three also rely on type analysis information, while only two detectors did not include semantic information from ExtendJ.

While we can export additional information from ExtendJ, detectors can also introduce their own helper predicates and, in principle, share them. After implementing the bug patterns, we surveyed our implementation and identified several helper predicates that we consider potentially reusable (possibly with minor refactoring), such as the subtyping relation, convenience extractors for type and membership information, method signature equality checks, and relations to mark defining and using occurrences of fields. While these observations show the importance of semantic information, all of our detectors used multiple syntactic patterns, and the Complex Operator Precedence detector demonstrated that semantic information is not always sufficient for real-life bug detectors, as it relied heavily on syntactic information that is no longer visible e.g. in Java bytecode.

Flow-Sensitive Analysis. Some program analyses, such as null-pointer dereference analysis or taint analysis [Arzt et al. 2014], rely on knowing the program execution order, typically modeled as a control-flow graph (CFG). We have not added CFG information to JavaDL due to the nontrivial engineering effort for building correct and precise CFGs for a mature language like Java, but see no conceptual obstacle to building a JavaDL CFG predicate library. Alternatively, we can import the CFGs that Riouak et al. [2021]'s recent IntraJ system superimposes over ExtendJ AST for Java 7.[5]

To better understand the importance of flow-sensitive analyses, we approximated their prevalence among the SpotBugs and Error Prone detectors by instrumenting each framework's CFG construction code and running individual detectors to analyze the SpotBugs core classes (967 files, 108kLOC) to see which detectors would trigger CFG construction. For SpotBugs, which groups related detectors into 167 *visitors*, we found that 35 of the 167 visitors (21%) triggered CFG construction, and 3 of Error Prone's 500 checkers (0.6%). These numbers are likely under-approximations but indicate that flow-sensitive analysis is important, but only for a minority of today's bug detectors.

We observed that two of the detectors that we had previously implemented were based on detectors that triggered CFG construction (Table 1, column **Notes**).

5.2.2 ***RQ1.2****: Precision.* Table 2 reports on checker precision and recall, comparing the Top-5 SpotBugs and Error Prone detectors against our JavaDL re-implementations. Several of the patterns contain subcategories (Table 1), and we report on the subcategories in cases where our implementation computed them separately. Whenever JavaDL reported bugs that Error Prone or SpotBugs did not report, we manually investigated 10 randomly sampled cases (or all, if less than 10), and proceeded analogously when the baseline tools reported a bug that JavaDL did not report.

For the EP-Top-5, most checkers have a high degree of agreement. Lowest (with 77% relative recall) is `UnnecessaryParentheses`. According to our sampling, the mismatch is due to the JavaDL detector not reporting parentheses surrounding unary operators, and Error Prone not reporting on parentheses surrounding case labels and return values.

[5] This framework was not yet available at the time of our experiments, while ExtendJ's earlier data flow framework [Söderberg et al. 2013] was unmaintained and incompatible with recent versions of ExtendJ.

Table 2. Number of results reported by ErrorProne, SpotBugs, and JavaDL for the selected bug patterns on the Defects4J repositories. "Precision" and "recall" are relative to the baseline tool (cf. Section 5.1). We report on bug subcategories for detectors that implement the subcategories separately.

Static Checker Framework (Baseline Tool)		Number of Results			Precision	
Name	Bug Pattern	JavaDL	Tool	Common	Precision	Recall
EP-Top-5	★ Boxed Primitive Constructor	429	425	423	98.60	99.53
	★ Missing `@Override`	4533	5341	4393	96.91	82.25
	`OperatorPrecedence`	84	100	82	97.62	82.00
	★ == on References	1176	1235	1169	99.40	94.66
	★ Useless Type Parameter	99	95	95	95.96	100.00
	`UnnecessaryParentheses`	257	174	134	52.14	77.01
SB-Top-5	`DM_NUMBER_CTOR`	190	182	158	83.16	86.81
	`DM_STRING_CTOR`	5	5	5	100.00	100.00
	★ Expose Internal Representation	3546	161	108	3.05	67.08
	`NM_CLASS_NAMING_CONVENTION`	0	0	0	N/A	N/A
	`NM_FIELD_NAMING_CONVENTION`	9	0	0	0.00	N/A
	`NM_METHOD_NAMING_CONVENTION`	1120	38	38	3.39	100.00
	★ Missing **`switch`** Default	223	87	81	36.32	93.10
	`UWF_UNWRITTEN_FIELD`	3	1	0	0.00	0.00
	`UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD`	3	0	0	0.00	N/A

For Missing `@Override`, JavaDL fails to produce warnings when the overridden method is a library method, since ExtendJ fails to evaluate attributes on some AST nodes created from bytecode, which excludes those nodes from the TYPE relation. The recall for `OperatorPrecedence` is due to differences in reporting: for instance, for `a && b || c && d`, Error Prone produces two warnings, one per `&&`, while JavaDL produces a single report for the entire expression, whose root is the `||`.

Turning to the bottom (SB-Top-5) half of the table, JavaDL is able to detect all `DM_STRING_CTOR` and `NM_METHOD_NAMING_CONVENTION` warnings reported by SpotBugs. The lower recall rates on Missing **`switch`** Default due to over-reporting in SpotBugs, caused by its lack of access to AST data (SpotBugs operates on Java bytecode [SpotBugs community 2021]). For `UWF_UNWRITTEN_FIELD`, JavaDL does report SpotBugs' single warning, but at the field's definition, rather than the access location. The other two warnings are unwritten fields that SpotBugs fails to detect. Similarly, we found that JavaDL's warnings for `UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD` were correct.

The low precision of Expose Internal Representation is due to JavaDL assuming that all reference types are mutable, while SpotBugs uses domain knowledge to filter out known immutable classes like `String`.

For `DM_NUMBER_CTOR`, we observed that SpotBugs and JavaDL match on 20 additional reports, with SpotBugs reporting the bug one line off from its source location. The remaining 4 reports that SpotBugs produces are constructor calls where the argument is a cast to `String`. JavaDL produces 12 additional correct warnings.

Overall, none of the differences indicate systematic limitations and could be addressed by evolving the JavaDL bug patterns.

5.3 RQ2: Execution Time Performance

Gathering Data on Run-Time Performance. To assess the efficiency of JavaDL for bug detection in a practical usage scenario, we iterated through a series of consecutive Git commits of Defects4J

Table 3. Distribution of commit sizes (touched files, added lines, removed lines) and the sizes of the projects at the start and at the end of the commit range. "M." stands for median and "90%" stands for the 90th percentile.

Project	Touched files/commit				Added lines/commit				Deleted lines/commit				Lines of code		Files	
	Mean	M.	90%	Max.	Mean	M.	90%	Max.	Mean	M.	90%	Max.	Start	End	Start	End
Cli	5.30	2	8	149	211.09	16	216	17844	176.54	4	79	19167	0	7070	0	52
Codec	3.15	1	5	170	57.86	6	111	1835	34.13	3	57	1430	13492	19568	85	122
Compress	3.39	1	4	180	70.52	10	171	1969	35.23	3	52	1965	31478	41838	251	341
Csv	1.71	1	3	20	28.97	5	61	1139	20.35	2	29	1152	3597	6117	25	31
J...Core	3.15	2	6	66	93.39	42	235	2839	33.52	9	75	1513	22477	43803	146	256
J...Databind	3.92	2	7	95	71.44	39	169	967	35.33	9	95	800	102361	114590	816	925
Jsoup	2.91	2	6	28	52.44	16	104	5287	20.40	3	32	2168	8854	20403	58	117
Math	7.16	2	12	210	211.95	30	501	28873	143.60	10	173	28291	6122	22749	74	281
Mockito	5.80	3	13	201	130.39	18	123	38603	33.42	8	69	1412	11014	38868	204	532
Time	8.79	2	10	628	583.62	29	472	127673	164.89	3.5	92	43798	45238	176827	240	656

projects and measured the time that it took to run JavaDL, Error Prone, and SpotBugs on each commit. For these measurements, we included each projects' unit tests, as we would in practice.

To ensure a fair comparison, we configured Error Prone and SpotBugs to only run their respective Top-5 detectors, and ran JavaDL separately for our implementations of SB-Top-5 and EP-Top-5. Unlike Error Prone and SpotBugs, JavaDL supports incremental evaluation, so we ran JavaDL with four configurations in total: {SB-Top-5, EP-Top-5} × {exhaustive, incremental}, with incremental runs re-using the IRDB from the previous incremental run (except for the first run).

For incremental analysis, we ran these experiments on 500 commits of the Defects4J benchmarks[6], and for exhaustive analysis we ran them on 50 equidistant commits from this range, including the first and the last commits. We selected these commits to be the closest predecessors of the commit we used for precision analysis, counting only commits that modified source files.

Table 3 shows the distributions of the number of touched files (modified, added or removed) from the selected sequence, and project sizes in the last four columns. We argue that these numbers show that the projects are realistic software projects at different stages in their life time.

For our performance analysis, we excluded several benchmarks: Chart, since it did not use Git (which our scripts required), and Closure, JxPath and Lang, because some of the commits among the 500 that we examined crashed ExtendJ and corrupted our IRDB. While recovering the IRDB from a backup would be trivial in a production system, we argue that including such runs would impair the representativeness of the incremental measurements. For the remaining projects we made a best effort to process them with all tools but found that SpotBugs and Error Prone's dependence on `javac` prevented many comparisons. We adjusted builds by tweaking compiler settings, adding libraries, or excluding problematic files (without examining the impact on our later measurements). Despite our efforts, some projects had a substantial number of failing builds (Cli: 18, Time: 25, Math: 29, Databind: 35, JacksonXML: 42) for the baseline checkers. We excluded all measurements for the affected revisions from our analysis and removed JacksonXML entirely. For incremental runs, we also the removed measurements for the 5 following/preceding revisions.

We ran our experiments on an Intel(R) Core(TM) i7-11700K CPU system, running at a fixed 3.6 GHz with 128 GiB RAM on Ubuntu 18.04.5 with Linux 5.13.7-051307-generic and OpenJDK 11.0.11+9-Ubuntu-0ubuntu2.18.04 inside a Docker container.

5.3.1 ***RQ2.1****: Performance Comparison.* Figure 14 shows the distribution of the measured running times for the incremental and exhaustive JavaDL analyses compared to the baseline detectors, i.e., Error Prone (above) and SpotBugs (below).

We observe that overall JavaDL performs competitively to the existing checkers, irrespective of code base size. Our incremental checker often but not universally outperforms the exhaustive

[6]Except for Cli where the Git history contains only 360 commits.

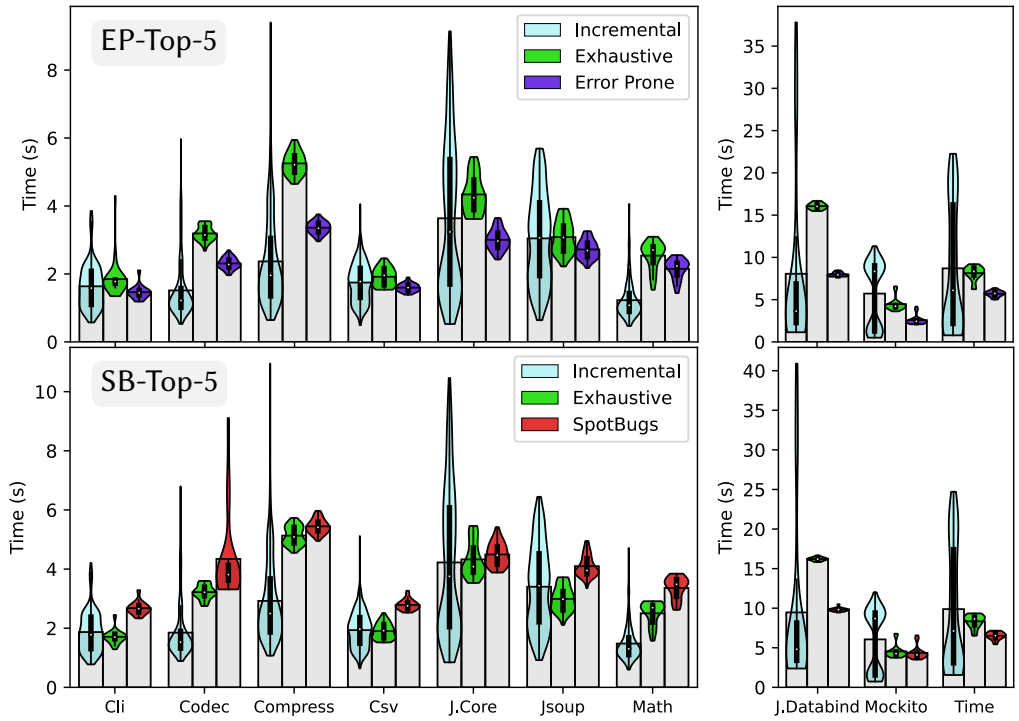
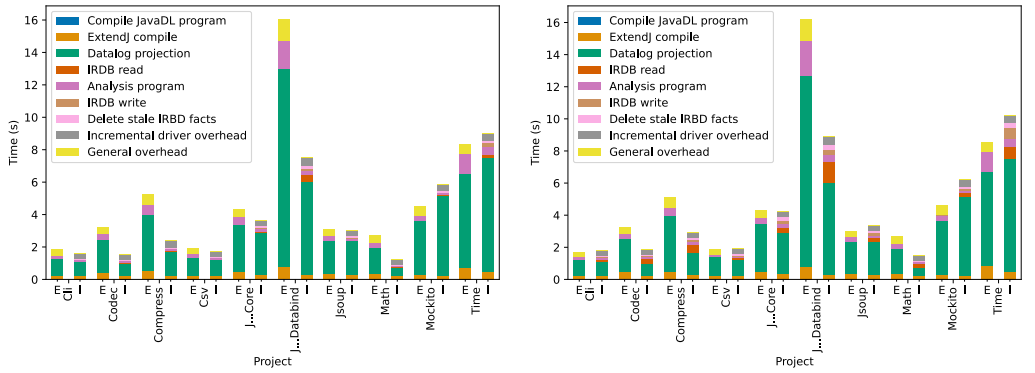


Fig. 14. Running time distribution on 500 consecutive commits for EP-Top-5 (above) and SB-Top-5 (below). The shaded boxes represent the mean.

checker. The similarity between the performance for JavaDL on the SB-Top-5 and EP-Top-5 suggests that JavaDL's performance depends more on the project structure than on the selection of checkers.

We further observe in the EP-Top-5 diagram that the Math, JacksonDatabind and Time projects have peaks in running times that are more than four times higher than the average. While similar in relative magnitude, the peaks have different causes. For Math, the peak is caused by a merge commit from a release branch (`6ef3b2932f`). For JacksonDatabind, the largest running time due to a merge commit (`33840a208b`) that modifies a significant number of files, though we also observed a set of high measurements caused by commits that modify files that are referenced by many other files (e.g., `DeserializationContext.java` in commit `5b8f0d9923`). The maximum running time for the Time project is caused by a major move/rename commit `53feb3fa56`, affecting 176 out of the total of 656 files in the project. Our brief analysis of the running time peaks also showed that for merge commits and major move/rename commits, the running time is higher than the time needed for the initial run of the analysis. A continuous integration system could use this observation to attempt to predict when it is more efficient to delete the IRDB and re-start from scratch, or to temporarily switch to exhaustive analysis.

Overall, our performance measurements suggest that our prototype framework offers run-time performance comparable to that of state-of-the-practice systems, with neither incremental nor exhaustive mode consistently outperforming the other.



(a) EP-Top-5 detectors (b) SB-Top-5 detectors

Fig. 15. Average running time for exhaustive (E) and incremental (I) runs, split by evaluation phases

5.3.2 ***RQ2.2****: Incremental Performance.* To explain the difference in incremental execution time between the projects, we turn to Figure 15, which shows the average time spent by JavaDL in each phase for the EP-Top-5 (a) and SB-Top-5 (b) detectors. While the running time of the analysis program is overall shorter for the incremental runs (including IRDB read and write), it is also clear that the incremental runs suffer from the overhead of the provenance tracking mechanisms that increase the time spent generating the program representation (*Datalog projection*). The Datalog projection phase grows with number of analyzed files, which includes the number of modified files as well as the files that need to be revisited because they contain an attribute whose value depends on modified files. On slower hard disks, we have also observed a second overhead, for deleting stale IRDB facts (not visible here).

We observed that it is mainly not commits that modify many files that contribute most to high running times, but commits that modify files on which other files' attributes depend. We characterize this property with the notion of the *relative degree* of a file F, which is the fraction of the files in the project that contain an attribute whose value was computed using F and whose analyses we must thus re-run when F changes.

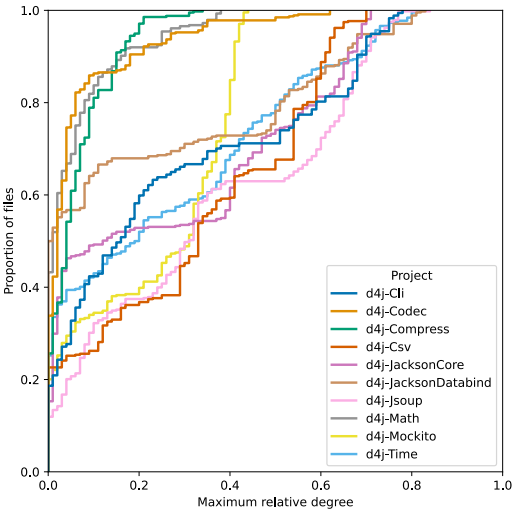


Fig. 16. Cumulative distribution of the maximum rel. degree in the analyzed commits

Figure 16 shows the cumulative distribution function computed over all the commits in each project. We can observe that for the Codec, Compress and Math projects, 80% of the commits have a degree of at most 0.1 and a further 10% a maximum degree of 0.2. They are followed by JacksonDatabind, which also has about 70% of the commits with a maximum degree of 0.2. These are the same projects in which our incremental analysis is most effective at outperforming the exhaustive analysis. Meanwhile, in Mockito 30% of the commits have a degree of less than 0.1 and 30% of the commits have a maximum degree of 0.4, which is also the maximum for this project, in line with the pronounced

bimodal distribution of the running times for the project. These observations indicate that the commit history is correlated with the running times of the incremental analysis, and thus can help inform which evaluation mode to choose for JavaDL for a given commit in a given project.

Analyzing the average running times spent in each phase for SB-Top-5 (Figure 15(b)), we observe that the amount of time needed to read, write and delete facts in the IRBD is more than twice that of the EP-Top-5 detectors. This indicates that the number of facts that are produced in the local evaluation pass but used in the global one is significantly higher in SB-Top-5.

The above observation reflects that we have not yet automated all locality optimizations that we are aware of, and manually optimized only EP-Top-5 for locality. Our main manual optimization was to split rules for locality. For instance, we could compress our earlier example (Figure 9) into:

```
BadNewString(f, l, c) :- n ⟨:new String(#v):⟩, TYPE(#v, τ), SRC(n, l, c, _, _, f),
                         τ ⟨:.. class String { .. }:⟩, SRC(s, _, _, _, _, "java/lang/String.class").
```

However, this rule is global and uses local predicates, so the predicates' tuples would be stored in the IRDB without further optimization. SRC in particular contains one tuple per AST node; materializing it on disk would be expensive. In Figure 9, we avoided this problem by extracting only the source locations of interest into the NewString predicate, analogously to the Magic Sets transformation [Bancilhon et al. 1985]. We expect to automate this process in future work.

5.4 Discussion and Limitations

One limitation of our language that we have so far omitted is that, unlike e.g. Prolog, our Datalog dialect is not Turing-complete but effectively restricted to **PTIME** [Immerman 1999]. While we did not find this restriction limiting in our experiments, it does bar JavaDL from expressing **EXPTIME**-complete analyses (e.g., bounded symbolic execution) or semi-decidable analyses (e.g., Java type analysis [Grigore 2017]). Thus, JavaDL offers less expressivity than afforded by e.g. general-purpose IDE/IFDS-based incrementalization frameworks [Arzt and Bodden 2014]; JavaDL is subject to the usual limitations of Datalog in this space. We defer to Madsen et al. [2016] for a detailed discussion.

We emphasize that this restriction is a trade-off: the syntactic simplicity of JavaDL ensures that JavaDL detectors are *clear-box detectors* and can be analyzed and transformed to integrate confidence score computation [Raghothaman et al. 2018], to explain their chains of reasoning [Zhao et al. 2020] or for incrementalization (as shown in this paper), or for integrations of these techniques, e.g., to use knowledge of recent changes to adjust confidence scores.

This restricted expressive power comes with more easily checkable correctness guarantees. For comparison, existing free-form (or black-box) IDE/IFDS-based approaches place complex *semantic* constraints of distributivity on transfer/flow functions [Reps et al. 1995] to ensure consistent results. Analyses that fail to meet these constraints may work correctly on some inputs but fail in corner cases or when run in different evaluation modes. By contrast, JavaDL projects will (in principle — i.e., not accounting for implementation bugs) evaluate correctly whenever they type-check.

Summary. Overall, our results show that our JavaDL implementation enables both exhaustive and incremental evaluation, with performance comparable to state-of-the-practice tools, and that the JavaDL language can concisely express a variety of both syntactic and semantic checks, both intra- and inter-procedural. While we have only analyzed the concepts that underlie JavaDL on one language (the subset of Java 8 supported by ExtendJ) and on a limited set of bug detectors, we argue that our results yield data points to support the case for clear-box bug detectors as devices for enabling architectural advances in bug checking.

6 RELATED WORK

JavaDL combines four strands of work: domain-specific languages for program analysis, Datalog dialects, pattern matching languages, and systems with alternative but equivalent execution models.

To the best of our knowledge, the first system to demonstrate the connection between DSLs for program analysis, logic programming, and pattern matching was the SOUL system by De Roover et al. [2011]. While SOUL does not aim to support multiple execution models or report performance on par with contemporary bug checkers, other aspects of its design are closely related to ours: SOUL provides syntactic patterns in a similar style as JavaDL [De Roover 2009; De Roover et al. 2007], albeit restricted to five syntactic categories. It performs logical reasoning through a SmallTalk-based Prolog implementation (rather than Datalog), connects to the Eclipse JDT framework (rather than to ExtendJ), and provides data flow analysis support via Soot [Vallée-Rai et al. 2010]. SOUL's use of Prolog results in a top-down evaluation strategy, which permits incremental evaluation and simplifies integration with traditional imperative analysis code, at the cost of performance [Ullman 1989], where Datalog has shown its strength in recent years [Bravenboer and Smaragdakis 2009; Scholz et al. 2016]. The SOUL pattern language also omits some language features such as generics, but captures the majority of the Java syntax supported by Eclipse in 2011, whereas our language is directly based on the grammar and AST of a Java compiler through an automatic transformation (with minimal manual intervention) and thus faithfully captures the entire Java syntax. Unlike SOUL, our work did not yet explore data flow analysis (as we discuss in Section 5.2.1).

Beyond SOUL, Visser [2002] argues in favor of syntactic pattern matching, with Fischer and Visser [2004] opting for concrete syntax over AST matching by arguing that "the simple abstract syntax approach quickly becomes cumbersome as the schemas become larger". Their AutoBayes system combines syntactic pattern matching with Prolog, compared to which DeepWeaver [Falconer et al. 2007] adds program rewriting for implementing aspect weaving. Another related system with syntactic pattern matching is our earlier MetaDL [Dura et al. 2019], which analyzes a Datalog dialect, in which it needs to match only one syntactic category. MetaDL uses a different AST encoding that maps each syntactic rule or AST node to a separate predicate (Section 3.3.2), which we found to be less efficient for Java than our approach here. Like SOUL, MetaDL has not seen a systematic evaluation for bug patterns, unlike JavaDL. A final closely related system is Cohen et al's JTL [Cohen et al. 2006], which provides a Datalog-style language together with a Java-like query syntax for querying Java class files for structural and dataflow information. Similar to JavaDL and SOUL, JTL exposes a number of built-in pre-computed predicates to aid program analysis, though it does not expose the AST and thus limits the amount of reasoning that it allows.

There are many other systems that support syntactic pattern matching. Visser [2002] uses GLR parsing for processing syntactic patterns, instead of Earley parsing as in our work. GLR parsing may improve the performance of our JavaDL specification frontend. To address possible bugs or inefficiencies from concrete syntax patterns, Kats et al. [2011] offer an interactive tool that suggests AST categories for concrete syntax tree patterns. Similarly, but on the concrete syntax level, Huang et al. [2008] automatically infer the syntactic type of syntactic patterns. We believe that this technique can enable optimizations and improve static checking in our syntactic patterns.

Kats et al. [2008] extend syntactic parsing in an extensible compiler that allows language extensions to normalize to mixtures of source code and bytecode, similarly to the Attribute Grammar system Silver [Van Wyk et al. 2010]. This style of normalization allows specifying analyses and transformations at the concrete syntax tree level while matching at a more abstract level, which (in effect) automatically generalizes analyses and transformations. The trade-off is that the effect of a match or transformation becomes less obvious, and that it may become impossible to match syntactic peculiarities that are not visible at the AST level. For example, JavaDL can distinguish

between `(x)` and `((x))`, which is necessary for our UnnecessaryParentheses bug pattern, but a system that uses matching after normalization may lack suitable information (e.g., in SpotBugs).

While not Datalog, two other closely related domain-specific languages are the program query language PQL [Martin et al. 2005] and the earlier commercial .QL [De Moor et al. 2007] system (now CodeQL). Neither system provides support for syntactic matching in the style of JavaDL or SOUL. PQL provides a SQL-like interface that allows not only static analysis, but also dynamic analysis and instrumentation, though it does not provide fine-grained AST / parse tree matching or support for analyzing arithmetic or primitive values. PQL is thus more useful for analyzing dynamic protocols, e.g., to check if a test during execution releases an external resource (e.g., a file handle) exactly once, while JavaDL is more useful for general-purpose static checking. Dynamic information can also be critical for analyzing reflection [Li et al. 2019]. For static tools like JavaDL, the lack of dynamic knowledge could be mitigated through ground facts from external tools [Bodden et al. 2011]. .QL, meanwhile, combines ideas from attribute grammars and SQL, giving it powerful static analysis capabilities, but without JavaDL-style syntactic matching capabilities.

While Datalog and its dialects have aided program analysis for over a decade, most existing tools like DOOP [Bravenboer and Smaragdakis 2009] rely on a separate fact extraction mechanism. DOOP's Java fact extractors translate either a Soot- or WALA-specific IR [Fink and Dolby 2012; Vallée-Rai et al. 2010] to a common representation, encoded as a set of predicates. This predicate-based Datalog IR encodes enough information for DOOP's points-to and call graph analyses, but (1) lacks source locations (necessary for precise error reports), (2) represents the program under analysis linearly, without any notion of nesting of type definitions, blocks or expressions (necessary for syntactic checks such as Complex Operator Precedence), and (3) hardcodes the set of predicates that it exposes to Datalog and requires detail knowledge of DOOP internals and Soot or WALA IRs to evolve or maintain this fact extraction code. By contrast, JavaDL provides the analysis code with a full representation of the analyzed program, which enables a broad range of static checks.

The literature has proposed several extensions to Datalog such as general-purpose lattices in Flix [Madsen et al. 2016] and IncA [Szabó et al. 2018]. These valuable extensions are orthogonal to ours. While JavaDL automates fact extraction by directly exposing the program AST, Basten and Klint [2008] propose a language-parametric scheme for using AST annotations to determine which facts to extract. We hypothesize that JavaDL could automate these AST annotations through bug pattern meta-analysis, to reduce the number of facts that we project from ExtendJ to Soufflé.

Earlier Prolog-based approaches to program analysis include Janzen and De Volder's JQuery system [Janzen and De Volder 2003], based on Prolog extended with aspect pointcut-style predicates and intended for Java code browsing, and a SOUL-like system Eichberg et al. [2007], without syntactic matching but with Eclipse IDE integration and Prolog-based incrementalization. Due to the different (top-down vs. bottom-up) evaluation modes, their techniques are not applicable to JavaDL. By using Prolog, both systems are in principle Turing complete, which may enable them to express more powerful analyses but loses the termination guarantee and bottom-up evaluation strategy (more efficient for whole-program analysis) that Datalog provides. Overall, the perhaps earliest discussion of logic programming for program analysis is due to Reps [1995], who focused on incrementalizing analysis through the Magic Sets transformation [Bancilhon et al. 1985].

Attribute Grammars [Knuth 1968] are another declarative programming paradigm for program analysis. Tools such as AbleC [Kaminski et al. 2017] for C and ExtendJ (formerly JastaddJ) for Java [Ekman and Hedin 2007; Öqvist and Hedin 2013], which underlies JavaDL, provide declarative features for computing attributes of AST nodes by synthesizing information from other AST nodes (including other attributes). Computations can be functional (in AbleC, based on Silver [Van Wyk et al. 2010]) or imperative (in ExtendJ, based on JastAdd [Hedin and Magnusson 2003]). Unlike Datalog, attribute grammar systems provide special support for reasoning over tree structures.

Silver provides support for syntactic pattern matching, but (to the best of our knowledge) no support for computations over general-purpose relations, while JastAdd does not provide syntactic pattern matching support, but can support general-purpose relations [Mey et al. 2018] (a feature not yet explored for purposes of program analysis). Attribute grammars have been used for bug finding [Söderberg et al. 2013], and some analyses may be easier to encode in attribute grammars. If so, JavaDL can import them through ExtendJ attributes. Attribute Grammars support multiple alternative execution models, like Datalog. *Reference attribute* grammars, supported in JastAdd, are evaluated on-demand [Hedin and Magnusson 2003] and can be evaluated both concurrently [Öqvist and Hedin 2017] and incrementally [Söderberg and Hedin 2012].

Arzt and Bodden [2014] have demonstrated incremental program analysis for (distributive) IFDS- and IDE-based analyses in their Reviser system, assuming the presence of a program differencing algorithm, and Cheetah by Do et al. [2017] demonstrates related techniques to prioritizing local bug detection during live editing. Their approaches to granularity are more fine-grained than ours: Reviser operates on CFG node differences, and Cheetah's notions of locality cover eight different layers, of which our file level is only one. Neither system supports syntactic matching. They focus exclusively on semantic properties, and both assume existing facilities to map the object language to graphs for flow analysis, whereas JavaDL only requires an AST.

Lhoták and Hendren [2004]'s Jedd system takes a near-converse approach to ours: they extend Java with special support for operations over relations and SAT solving, and use it in the context of Soot [Vallée-Rai et al. 2010] for imperative program analyses. More recently, Opal [Helm et al. 2020] demonstrated complex and high-performance program analyses by combining imperative implementations of program analyses in a Datalog-like blackboard architecture. While the analyses themselves are not clear-box specifications, Opal obtains some ability to alternate execution modes from hand-written meta-information and manual incrementalization. Other program analysis systems such as Soot [Vallée-Rai et al. 2010], WALA [Fink and Dolby 2012], or Spoon [Pawlak et al. 2016] provide program analysis facilities as regular Java libraries.

7 CONCLUSION

We have introduced JavaDL, the first static checker framework (to the best of our knowledge) that can run any **PTIME**-computable static bug detector on Java from a single specification both exhaustively and incrementally, while automatically rewriting the specification to optimize it for incremental evaluation. Our tool combines syntactic patterns for local reasoning with declarative, Datalog-style nonlocal reasoning. We have demonstrated that the efficiency of our prototype implementation, based on an existing high-performance Datalog engine, is competitive to state-of-the-practice systems, that our specification language can concisely express typical bug detectors, and that its incremental and exhaustive evaluation are both able to outshine each other in different usage scenarios. We argue that our results demonstrate the value and viability of *clear-box bug checker frameworks*, which constrain the bug detector specification language in order to obtain the ability to analyze and transform detector specifications for different usage modes.

ACKNOWLEDGEMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The authors thank Alfred Åkesson, Görel Hedin, Luke Church, the members of the Lund University Software Technology reading group, and especially the anonymous OOPSLA reviewers for their substantial and valuable feedback.

REFERENCES

Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 14–23. https://doi.org/10.1109/SCAM.2012.28

Steven Arzt and Eric Bodden. 2014. 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 (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269. https://doi.org/10.1145/2666356.2594299

Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and Johan Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium*. Springer, 84–104. https://doi.org/10.1007/978-3-662-53413-7_5

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. 1–15.

Bas Basten and Paul Klint. 2008. DeFacto: Language-Parametric Fact Extraction from Source Code. In *Revised Selected Papers of the First International Conference on Software Language Engineering (Lecture Notes in Computer Science)*, Dragan Gašević, Ralf Lämmel, and Eric Van Wyk (Eds.), Vol. 5452. Springer International Publishing, 265–284. https://doi.org/10.1007/978-3-642-00434-6_17

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 241–250. https://doi.org/10.1145/1985793.1985827

Simon Brandhof, Julien Lancelot, Stas Vilchik, Fabrice Bellingard, David Gageot, Jean-Baptiste Vilain, Eric Hartmann, and Freddy Mallet. 2014. SonarQube. Retrieved 10 September 2021 from https://github.com/SonarSource/sonarqube

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of OOPSLA '09*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Oliver Burn, Roman Ivanov, Richard Veach, Pavel Bludov, Andrei Paikin, Ilja Dubinin, Andrei Selkin, Vladislav Lisetskii, Michał Kordas, Ruslan Diachenko, Baratali Izmailov, Daniil Yaroslavtsev, Ivan Sopov, Lars Kühne, Rick Giles, Oleg Sukhodolsky, Michael Studman, and Travis Schneeberger. 2021. Checkstyle 9.0. Retrieved 10 September 2021 from https://checkstyle.sourceforge.io/

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166.

Tal Cohen, Joseph Gil, and Itay Maman. 2006. JTL: the Java tools language. *ACM SIGPLAN Notices* 41, 10 (2006), 89–108.

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

Oege De Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2007. .QL: Object-oriented queries made easy. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 78–133. https://doi.org/10.1007/978-3-540-88643-3_3

Coen De Roover. 2009. *A Logic Meta Programming Foundation for Example-Driven Pattern Detection in Object-Oriented Programs*. Ph.D. Dissertation. Vrije Universiteit Brussel.

Coen De Roover, Theo D'Hondt, Johan Brichau, Carlos Noguera, and Laurence Duchien. 2007. Behavioral similarity matching using concrete source code templates in logic queries. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 92–101. https://doi.org/10.1145/1244381.1244398

Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. 2011. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/2093157.2093168

Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-Time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705

Alexandru Dura, Hampus Balldin, and Christoph Reichenbach. 2019. MetaDL: Analysing Datalog in Datalog. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 38–43. https://doi.org/10.1145/3315568.3329970

Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021a. JavaDL: Automatically Incrementalizing Java Bug Pattern Detection. https://doi.org/10.5281/zenodo.5090141 (artifact, OOPSLA 2021 AEC evaluated).

Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021b. JavaDL: Automatically Incrementalizing Java Bug Pattern Detection. (Sep 2021). https://doi.org/10.5281/zenodo.5090140 (artifact, updated after OOPSLA AEC evaluation).

Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. 2007. Automatic incrementalization of prolog based static analyses. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

Torbjörn Ekman and Görel Hedin. 2007. The JastAdd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (Oct. 2007), 1–18. https://doi.org/10.1145/1297105.1297029

Henry Falconer, Paul H. J. Kelly, David M. Ingram, Michael R. Mellor, Tony Field, and Olav Beckmann. 2007. A Declarative Framework for Analysis and Optimization. In *Compiler Construction*, Shriram Krishnamurthi and Martin Odersky (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 218–232. https://doi.org/10.1007/978-3-540-71229-9_15

Stephen Fink and Julian Dolby. 2012. WALA–The TJ Watson Libraries for Analysis.

Bernd Fischer and Eelco Visser. 2004. *Retrofitting the AutoBayes Program Synthesis System with Concrete Syntax*. Springer Berlin Heidelberg, Berlin, Heidelberg, 239–253. https://doi.org/10.1007/978-3-540-25935-0_14

Radu Grigore. 2017. Java Generics are Turing Complete. *ACM SIGPLAN Notices* 52, 1 (2017), 73–85. https://doi.org/10.1145/3093333.3009871

Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 317–328. https://doi.org/10.1145/3238147.3238213

Görel Hedin and Eva Magnusson. 2003. JastAdd: An Aspect-oriented Compiler Construction System. *Sci. Comput. Program.* 47, 1 (April 2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0

Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. 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 (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 184–196. https://doi.org/10.1145/3368089.3409765

Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously reasoning about programs using differential Bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 561–575. https://doi.org/10.1145/3314221.3314616

Shan Shan Huang, David Zook, and Yannis Smaragdakis. 2008. Domain-Specific Languages and Program Generation with Meta-AspectJ. *ACM Trans. Softw. Eng. Methodol.* 18, 2, Article 6 (Nov. 2008), 32 pages. https://doi.org/10.1145/1416563.1416566

N. Immerman. 1999. *Descriptive Complexity*. Springer New York. http://books.google.de/books?id=kWSZ0OWnupkC

Doug Janzen and Kris De Volder. 2003. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*. 178–187.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440. https://doi.org/10.1145/2610384.2628055

Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and automatic composition of language extensions to C: the ableC extensible language framework. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. https://doi.org/10.1145/3138224

Lennart C.L. Kats, Martin Bravenboer, and Eelco Visser. 2008. Mixing Source and Bytecode: A Case for Compilation by Normalization. *SIGPLAN Not.* 43, 10 (Oct. 2008), 91–108. https://doi.org/10.1145/1449955.1449772

Lennart C. L. Kats, Karl T. Kalleberg, and Eelco Visser. 2011. Interactive Disambiguation of Meta Programs with Concrete Object Syntax. In *Software Language Engineering*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 327–336. https://doi.org/10.1007/978-3-642-19440-5_22

Donald E. Knuth. 1968. Semantics of context-free languages. *Theory of Computing Systems* 2, 2 (1 June 1968), 127–145. https://doi.org/10.1007/BF01692511

Ondřej Lhoták and Laurie Hendren. 2004. Jedd: a BDD-based relational extension of Java. *ACM SIGPLAN Notices* 39, 6 (2004), 158–169. https://doi.org/10.1145/996893.996861

Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* 28, 2, Article 7 (Feb. 2019), 50 pages. https://doi.org/10.1145/3295739

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: a Declarative Language for Fixed Points on Lattices. *ACM SIGPLAN Notices* 51, 6 (2016), 194–208. https://doi.org/10.1145/2980983.2908096

Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 365–383. https://doi.org/10.1145/1094811.1094840

Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. 2018. Continuous model validation using reference attribute grammars. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 70–82. https://doi.org/10.1145/3276604.3276616

Krishna Narasimhan, Christoph Reichenbach, and Julia Lawall. 2018. Cleaning up copy–paste clones with interactive merging. *Automated Software Engineering* 25, 3 (01 Sep 2018), 627–673. https://doi.org/10.1007/s10515-018-0238-5

Peter O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.

Jesper Öqvist and Görel Hedin. 2013. Extending the JastAdd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 147–152. https://doi.org/10.1145/2500828.2500843

Jesper Öqvist and Görel Hedin. 2017. Concurrent circular reference attribute grammars. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 151–162. https://doi.org/10.1145/3136014.3136032

Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179. https://doi.org/10.1002/spe.2346

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 722–735. https://doi.org/10.1145/3192366.3192417

Christoph Reichenbach. 2021. Software Ticks Need No Specifications. In *Proceedings of the 43rd International Conference on Software Engineering: New Ideas and Emerging Results Track*. IEEE - Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/ICSE-NIER52604.2021.00021 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE 2021 ; Conference date: 23-05-2021 Through 29-05-2021.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61. https://doi.org/10.1145/199448.199462

Thomas W Reps. 1995. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*. Springer, 163–196. https://doi.org/10.1007/978-1-4615-2207-2_8

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. 2021. A Precise Framework for Source-Level Control-Flow Analysis. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021*. IEEE Computer Society.

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2015.76

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th Int. Conf. on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

Elizabeth Scott. 2008. SPPF-style parsing from Earley recognisers. *Electronic Notes in Theoretical Computer Science* 203, 2 (2008), 53–67. https://doi.org/10.1016/j.entcs.2008.03.044

Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. 2013. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming* 78, 10 (2013), 1809 – 1827. https://doi.org/10.1016/j.scico.2012.02.002

Emma Söderberg and Görel Hedin. 2011. Automated Selective Caching for Reference Attribute Grammars. In *Software Language Engineering*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–21. https://doi.org/10.1007/978-3-642-19440-5_2

Emma Söderberg and Görel Hedin. 2012. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Technical Report 98. Lund University. LU-CS-TR:2012-249, ISSN 1404-1200.

SpotBugs community 2021. SpotBugs 4.4.1 Bug Descriptions. Retrieved 10 September 2021 from https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-Based Program Analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276509

J. D. Ullman. 1989. Bottom-up Beats Top-down for Datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '89)*. Association for Computing Machinery, New York, NY, USA, 140–149.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., Riverton, NJ, USA, 214–224. https://doi.org/10.1145/1925805.1925818

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25 (2020), 1419–1457. Issue 2. https://doi.org/10.1007/s10664-019-09750-5

Eelco Visser. 2002. Meta-Programming with Concrete Object Syntax. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*. Springer-Verlag, Berlin, Heidelberg, 299–315. https://doi.org/10.1007/3-540-45821-2_19

Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher-Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*. 131–145. https://doi.org/10.1145/73141.74830

David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–35. https://doi.org/10.1145/3379446