



LUND
UNIVERSITY

ShiftLeft

Cross-Cutting Concerns in Declarative Program Analysis

Alexandru Dura
Idriss Riouak
Anton Risberg Alaküla
Christoph Reichenbach

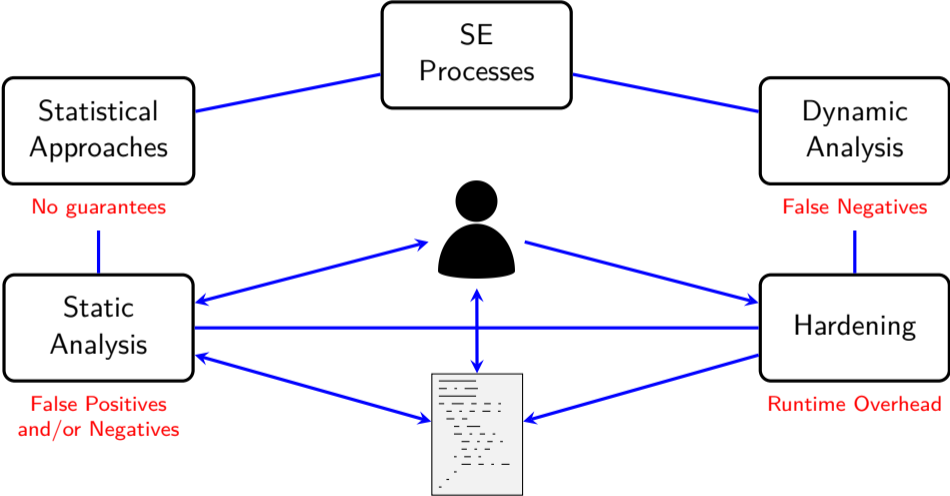
Görel Hedin
Niklas Fors
Emma Söderberg
Erik Präntare

WASP | WALLEBERG AI,
AUTONOMOUS SYSTEMS
AND SOFTWARE PROGRAM



Datalog for Program Analysis: Strengths

No Free Lunches in Security



Integration: Beyond Individual Techniques

Static Analysis + Statistics

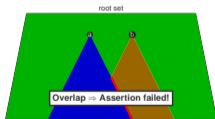
What Can the GC Compute Efficiently? A Language for Heap Assertions at GC Time

Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis
University of Massachusetts
Edward Athandilian, Sam Guyer
Tufts University

RECALL THE GARBAGE COLLECTION BASICS



COMBINED QUERIES



forall Node n: $R(a)[n] \Rightarrow n.value > 0$
 $\&\& R(b)[n] \Rightarrow n.value <= 3$

WHAT CAN THE GC COMPUTE EFFICIENTLY? A LANGUAGE FOR HEAP ASSERTIONS AT GC TIME 30

Learning Probabilistic Models for Static Analysis Alarms

Hyunsu Kim, KAST, Korea, hyunsu.kim@kast.ac.kr
 Mukund Raghothaman, University of Southern California, USA, raghothu@usc.edu
 Kihong Ho, KAST, Korea, kihong_ho@kast.ac.kr

ABSTRACT
 In recent literature, a general framework for automatically learning probabilistic models of static analysis alarms. Several probabilistic reasoning techniques have recently been proposed which incorporate external feedback on semantic facts and thereby reduce the user's alarm inspection burden. However, these approaches are fundamentally limited to models with pre-defined structure, and are therefore unable to learn or transfer knowledge regarding an analysis from one program to another. Furthermore, these probabilistic models often aggressively generalize from external feedback and thereby suppress real bugs. To address these problems, we propose a framework that learns the structure and weights of the probabilistic model. Starting from an initial model and a set of training programs with bug labels, *LearnStructure* refines the model to effectively prioritize real bugs based on feedback. We evaluate the approach with two static analyses on a suite of C programs. We demonstrate that the learned models significantly improve the performance of these state-of-the-art probabilistic reasoning systems.

1 INTRODUCTION
 In line with the challenges of accuracy and alarm relevance, various probabilistic program reasoning mechanisms have been proposed to static program analyzers. Such systems initially report a set of alarms in the target program using the underlying analysis and compute the probability of each alarm based on a probabilistic model. Then, they prioritize static analysis alarms by incorporating external feedback on semantic facts from various sources such as the user [23, 39, 36], the old version of the program [11], or dynamic analysis results [2]. Upon receiving a response, they generalize from the feedback and prioritize the remaining alarms depending on their relevance to those inspected by the user. By rapidly iterating attention on the real bugs in the target program, these systems achieve a dramatic improvement in the usability of the static analyzer.

Despite their experimental success, much of this previous research has focused on the problem of inference, rather than on learning. Existing approaches based on probabilistic reasoning such as alarm ranking [3, 15, 39] only have limited focus of transfer-

operation maintenance algorithms [28]. In our observation, however, the capability of learning is fundamentally limited to the underlying structure of the probabilistic model.

In this paper, we propose a general framework for learning probabilistic models for static analysis alarms, that is applicable to various probabilistic reasoning systems [3, 15, 39]. The underlying system initially constructs Bayesian networks from the destination structure of the alarms, and prioritizes alarms using the induced confidence values. This ranking is repeatedly updated as the user inspects alarms and reports findings. Ideally, these responses should do ways improve the confidence scores of true bugs and decrease those of false alarms. In practice, however, because of approximations caused by the underlying abstraction and during model recovery they often incorrectly prioritize false alarms over true bugs. Our goal is to improve the accuracy of probabilistic models and mitigate the impact of these false generalization events. Given a set of training programs with bug labels, we learn logical rules that produce accurate Bayesian network models to reduce the number of user interactions until discovering all true alarms.

Notice that our problem (i.e., learning) is fundamentally different from that (i.e., inference) addressed in previous papers [3, 15, 39]. Learning and inference are complementary problems in AI/ML research, especially for Bayesian networks. The existing work is focused on the inference of Bayesian probabilistic models generated by a third hand-written set of rules. Instead, we shed light on the capability of learning the Bayesian models that can significantly improve the performance on multiple instances.

Our approach is based on true bug labels (1) feedback-directed and (2) system-guided refinement. We first construct the Bayesian networks with an initial set of rules and evaluate the quality of interactive alarm prioritization using a given labeled data set. By observing the results, we capture the moments when a response to one alarm likely generalizes to other alarms and degrades the overall quality of ranking. For each case, our learning algorithm refines the rules that generate the inaccurate part of the Bayesian networks. The refinement is guided by program syntax and domain-related context of the labeled alarms to the rules by adding syntactic features which are directly derived from the grammar of the program language.

Residual Investigation

Predictive and Precise Bug Detection

Kaituo Li¹, Christoph Reichenbach²,
 Christoph Csallner³ and Yannis Smaragdakis⁴

Our Solution: Residual Investigation

Static Checker **Residual Investigation**

```

    graph LR
        subgraph SC [Static Checker]
            T[Tests]
            P[Program]
        end
        subgraph RI [Residual Investigation]
            R[Run]
            A[Analyse]
        end
        T --> R
        P --> R
        R --> A
    
```

Residual Checks and Static Checks

- Static analysis produces false positives
- Testing is incomplete
- Residual Investigation collects evidence to connect static checks to bugs

Dynamic Analysis + Hardening

Static + Dynamic Analysis

Integration: Challenges

- Features external to analysis? (dynamic data, design docs, ...)
- Learning and adaptation?
- Explainability?
- Scalability?
 - Demand-driven or incremental evaluation
 - Differential Analysis
 - Trading off precision vs. efficiency (widening, context sensitivity)

	CWEs
Weaknesses	399
Categories	40
Views	0
Total	439

...

	Analysis	Learning	Explanation	Scalability
Input not sanitised				
Null pointer deref				
Array out-of-bounds				
...				

Cross-Cutting Features

Vision: Transparent Analyses

- Separation:
 - Expose building blocks of analyses
 - “Small” computations
 - No side effects
 - Dependencies known (?)
 - Generic “weaving” mechanism for cross-cutting concerns
 - Re-use evaluation results
 - Trace provenance
 - Inject context information
 - Back-propagate user feedback
- Similar architectures:
 - Program Query Language (Martin et al.)
 - Attribute Grammars (Knuth)
 - IFDS / IDE (Horwitz, Reps, Sagiv)
 - OPAL (Helm et al.)

Declarative Approaches to Program Analysis

Code Property Graphs (Joern)

```
{val printf  
  .method("  
  .callIn  
  .whereNot(  
val sprintf  
  .method("  
  .callIn  
  .whereNot(  
(printfFns ++ sprintfFns)}.1
```

- AST, CFG, PDG traversal
- Filter by edge annotation

Joern language [<https://queries.joern.io>]

Syntactic Patterns (Coccinelle)

```
@ haskernel @  
@@  
#include <linux/kernel.h>  
@ depends on  
expression n,  
@@  
(  
- (((n) + (d  
+ DIV_ROUND_UP  
|  
- (((n) + ((d) - 1)) / (d))  
+ DIV_ROUND_UP(n,d)  
)
```

- AST Structure detection
- Syntactic rewriting

Coccinelle, [<https://coccinelle.gitlabpages.inria.fr>]

Datalog Queries

```
Reachable(?t  
  Reachable(  
  Instruction  
  VirtualMet  
  VarPointsTo  
  HeapAllocat  
  VirtualMet  
  VirtualMet  
  basic.MethodLookup(?sname, ?descriptor, ?heaptype,  
    ?toMethod).
```

- Logic programming
- Recursion for graph traversal

DOOP, [<https://github.com/plast-lab/door/>]

Reference Attribute Grammars (JastAdd)

```
inh Decl  
inh Decl  
inh Decl  
eq Block  
if (!l  
  return  
  return  
}
```

- Implicit AST propagation
- Overlay CFG on AST
- Embedded Java code (no visible side effects)

JastAdd, [<https://jastadd.org>]

Practicality of Declarative Approaches

- **Usability?**

- Is it reasonable to ask developers to work with declarative DSLs?

- **Efficiency?**

- Are declarative approaches fast enough?

- **Effective Transparency?**

- Do they actually simplify cross-cutting concerns?

Building Practical Static Analysers?

Example analysis:

```
enum Direction { N, S, E, W };  
...  
switch (dir) {  
case N: ...  
case S: ...  
case E: ...  
// Missing switch case or default!  
}
```

- **Task 1:** find `switch`
- **Task 2:** check `default`
- **Task 3:** check completeness

```

1 public class MissingDefault ... { ...
2   public Description matchSwitch(SwitchTree tree, VisitorState state) {
3     Type switchType = ASTHelpers.getType(tree.getExpression());
4     if (switchType.asElement().getKind() == ElementKind.ENUM) {
5       return NO_MATCH;
6     }
7     Optional<? extends CaseTest> maybeDefault =
8       tree.getCases().stream().filter(c -> c.getExpression() == null).findFirst();
9     if (!maybeDefault.isPresent()) { ...
10      return description.build();
11    } ... }

```

(Task 1) Callback: AST visitor invokes method on switch statements

Don't check the Switch/Enum case here

(Task 2) Default case is encoded as case with null expression: scan for it

```

1 public class MissingCasesInEnumSwitch ... { ...
2   public Description matchSwitch(SwitchTree tree, VisitorState state) {
3     Type switchType = ASTHelpers.getType(tree.getExpression());
4     if (switchType.asElement().getKind() != ElementKind.ENUM) {
5       return Description.NO_MATCH;
6     }
7     if (tree.getCases().stream().anyMatch(c -> c.getExpression() == null)) {
8       return Description.NO_MATCH;
9     }
10    ImmutableSet<String> handled =
11      tree.getCases().stream()
12        .map(CaseTree::getExpression)
13        .filter(IdentifierTree.class::isInstance)
14        .map(e -> ((IdentifierTree) e).getName().toString())
15        .collect(toImmutableSet());
16    Set<String> unhandled = Sets.difference(
17      ASTHelpers.enumValues(switchType.asElement()), handled);
18    if (unhandled.isEmpty()) {
19      return Description.NO_MATCH;
20    }
21    return buildDescription(tree).setMessage(buildMessage(unhandled)).build();
22  } ... }

```

Only check the Enum case here

(Task 3) Compute set of case handlers via stream processing, compare against expectations

```
1 <rule name="SwitchStmtsShouldHaveDefault" ... > ...
2     //SwitchStatement[@DefaultCase = false() and @ExhaustiveEnumSwitch = false()]
3     ...
4 </rule>
```

Declarative specification

(Task 1) XPath expression informs AST visitor to trigger callbacks if it visits switch statement

```
1 public class ASTSwitchStatement ... { ...
2 public boolean hasDefaultCase() {
3     for (ASTSwitchLabel label : this) {
4         if (label.isDefault()) {
5             return true;
6         }
7     }
8     return false;
9 }
10 public boolean isExhaustiveEnumSwitch() {
11     ASTExpression expression = getTestedExpression();
12     if (expression.getType() == null) {
13         return false;
14     }
15     if (Enum.class.isAssignableFrom(expression.getType())) {
16         Set<String> constantNames = EnumUtils.getEnumMap(
17             (Class< extends Enum>) expression.getType()).keySet();
18         for (ASTSwitchLabel label : this) {
19             constantNames.remove(label.getFirstDescendantOfType(ASTName.class).getImage());
20         }
21         return constantNames.isEmpty();
22     }
23     return false;
24 }
25 }
```

(Task 2) Loop through case labels, check for default

(Task 3) Create set of all enum constants, gradually remove entries, check if empty

(Task 1) Filter by AST node type

```
1 from SwitchStmt switch, EnumType enum, EnumConstant missing
2 where
3   switch.getExpr().getType() = enum and
4   missing.getDeclaringType() = enum and
5   not switch.getAConstCase().getValue() = missing.getAnAccess() and
6   not exists(switch.getDefaultCase())
7 select switch
```

(Task 3) Does there exist a missing value?

(Task 2) Lacking default?

enum switches only

Checker #3: CodeQL

```

1 aspect Shared_SwitchDefault {
2   inh SwitchStmt Case.enclosingSwitchStmt();
3   eq Program.getChild().enclosingSwitchStmt() = null;
4   eq SwitchStmt.getChild().enclosingSwitchStmt() = this;
5
6   coll HashSet<ConstCase> SwitchStmt.validConstCases() root SwitchStmt;
7   ConstCase contributes this
8     when enclosingSwitchStmt() != null
9       && typeProblems().isEmpty() && nameProblems().isEmpty()
10    to SwitchStmt.validConstCases()
11    for enclosingSwitchStmt();
12
13   syn boolean SwitchStmt.isFullyMatchedEnum() = getExpr().type().isEnumDecl()
14     && validConstCases().size() ==
15     ((EnumDecl) getExpr().type()).enumConstants().size();
16 }
17
18 aspect JDL_SwitchDefault {
19   SwitchStmt contributes error("Missing-default-case")
20     when defaultCase() = null && !isFullyMatchedEnum()
21     to CompilationUnit.javadlProblems();
22 }

```

(Task 1a) switch AST node announces itself to its children

Collect all well-typed case branches for switch

(Task 3) Check # branches vs # enum constructors

(Task 2) Check default via attribute

(Task 1b) Check all switch nodes

Checker #4: ExtendJ

```
SWITCHWITHDEFAULT(s) :- s [switch (#_) { .. default: .. }].
```

```
SWITCH(s) :- s [switch (#_) { .. }].
```

(Task 1) Find switch statements

```
CASEONENUM(s, e, d) :- s [switch (#v) { .. case #c : .. }],
```

```
    TYPE(#v, e), e [enum #_ { .. }],
```

```
    DECL(#c, d).
```

```
// variable #m can be either enum member or enum constant,  
// use ID to discriminate between the two
```

```
ENUMMEMBER(e, #m) :- e [enum #_ { .., #m, .. ; .. }],
```

```
    ID(#m, _).
```

```
SWITCHWITHOUTENUMMEMBER(s, e) :- CASEONENUM(s, e, _),
```

```
    ENUMMEMBER(e, m),
```

```
    NOT(CASEONENUM(s, e, m)).
```

```
SWITCHONALLENUMMEMBERS(s) :- CASEONENUM(s, e, _),
```

```
    NOT(SWITCHWITHOUTENUMMEMBER(s, e)).
```

```
SWITCHWITHOUTDEFAULT(s) :- SWITCH(s),
```

```
    NOT(SWITCHWITHDEFAULT(s)),
```

```
    NOT(SWITCHONALLENUMMEMBERS(s)).
```

```
SWITCHWITHOUTDEFAULTDETAIL(l, c, file) :- SWITCHWITHOUTDEFAULT(s),
```

```
    SRC(s, l, c, _, _, file),
```

```
    GT(l, 0).
```

(Task 2) Is SWITCH but not SWITCHWITHDEFAULT

(Task 3) Find missing enum members

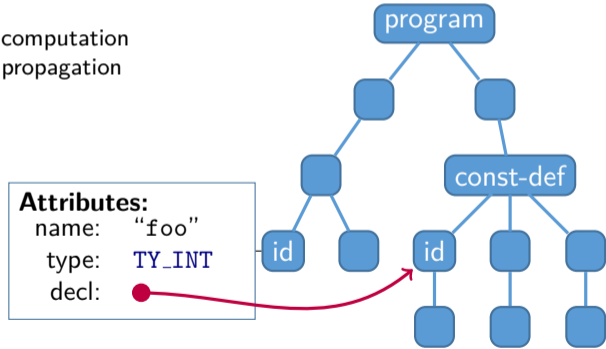
Add location information

Bug Patterns in JastAdd / ExtendJ via Reference Attribute Grammars

with **Idriss Riouak**, **Anton Risberg Alaküla**, **Niklas Fors**, and **Görel Hedin**

Reference Attribute Grammars

- Attributes adorn AST nodes with:
 - Values
 - References
 - Equations
 - Explicit computation
 - Implicit propagation



Varieties of Attributes

■ Propagation:

- **Synthesised**: children \rightarrow parent
- **Inherited**: ancestor \rightarrow descendants (automatic forwarding)
- **Collection**: descendant \rightarrow nearest ancestor (with aggregation)

■ Categories of Attributes:

- **Value**
- **Reference**
- **Parameterised** (method-like)
- **Nonterminal** (*synthetic* AST fragment)

■ Evaluation Options:

- **Cached** (eval at most once)
- **Circular** (fixpoint computation)
- **Concurrent**

JastAdd and ExtendJ

■ JastAdd

- Reference Attribute Grammar (RAG) system based on Java
- Equations can contain arbitrary Java code
- Aspect-like composition mechanisms
- **OO-style equation inheritance**

■ ExtendJ

- **Java 8** compiler implemented in JastAdd
- **RAGs enable Data Flow Analysis**

JastAdd: Cross-Cutting Concerns

- Demand-driven evaluation
- Incremental evaluation [Söderberg, Hedin: “Incremental evaluation of reference attribute grammars using dynamic dependency tracking”, 2012]
- Support for target language extensions

JavaDL (MetaDL[Java])

with **Alexandru Dura** and **Emma Söderberg**

[Alexandru Dura, Christoph Reichenbach, Emma Söderberg: 'JavaDL: Automatically Incrementalizing Java Bug Pattern Detection', ECOOP 2021]

Program Analysis in Datalog

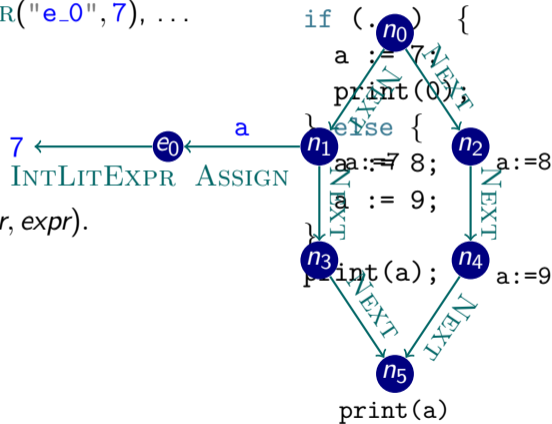
Input Facts
NEXT("n_0", "n_1"), NEXT("n_0", "n_2"), ...
ASSIGN("n_1", "a", "e_0"), INTLITEXPR("e_0", 7), ...

PATH(x, y) :- NEXT(x, y).

PATH(x, z) :- PATH(x, y), PATH(y, z).

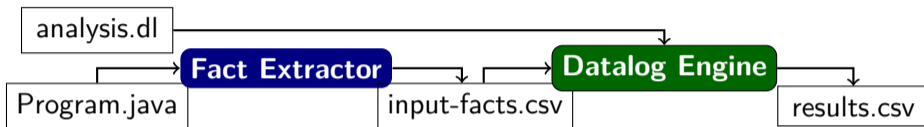
MAYREACH(z, var, expr) :- ASSIGN(z, var, expr).

MAYREACH(z, var, expr) :-
 ¬ASSIGN(z, var, _),
 MAYREACH(x, var, expr),
 NEXT(x, z).

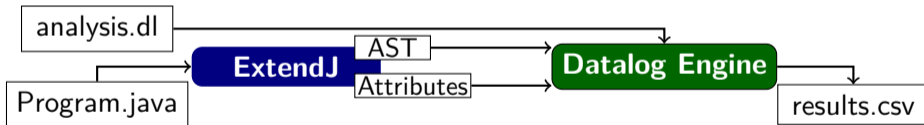


Program Analysis with Datalog

Common architecture (Doop, CodeQL etc.):



JavaDL:



- Doop etc.: hand-written fact extractor, manually aligned with Datalog code
- JavaDL: integrated with JastAdd parser, attributes
 - Syntax-to-Datalog mapping derived automatically
 - Can also export (most) ExtendJ attributes if needed

MetaDL[X]

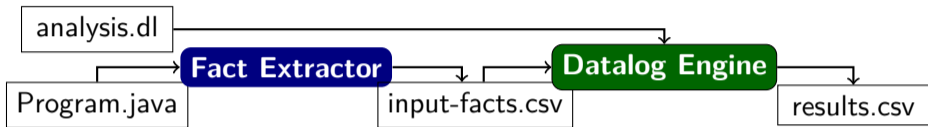
- MetaDL: Datalog with syntactic patterns
 - for Datalog
 - JavaDL: plus patterns for Java
 - Clog: plus patterns for C

JavaDL = MetaDL[Java]

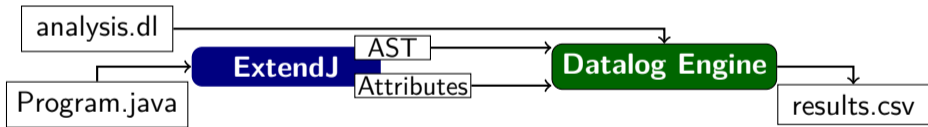
- Use (parts of) ExtendJ as analyser frontend
- Pattern matching on Java source code
 - Expose tree structure, access to subtree root (r):
 r [#e + 0]
 - Automatically derive: pattern grammar \leftarrow ExtendJ grammar
 - Highly ambiguous grammar
- Expose additional information from ExtendJ:
 - TYPE**(n, τ) n has type τ (AST node representing the class/type)
 - DECL**(n_1, n_2) Declaration site for n_1
 - INT**(n, i) Integer value i of n , if n is int literal
 - ID**(n, s) String representation of identifier, if n is identifier
 - SRC**(n, \dots) Source location (line, column, file)
 - SUCC**(n, m) CFG successor/predecessor

Program Analysis with Datalog

Common architecture (Doop, CodeQL etc.):



JavaDL:



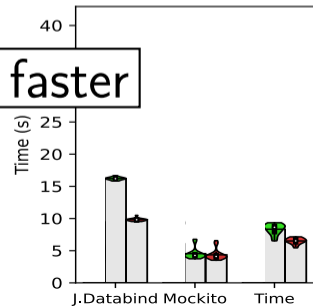
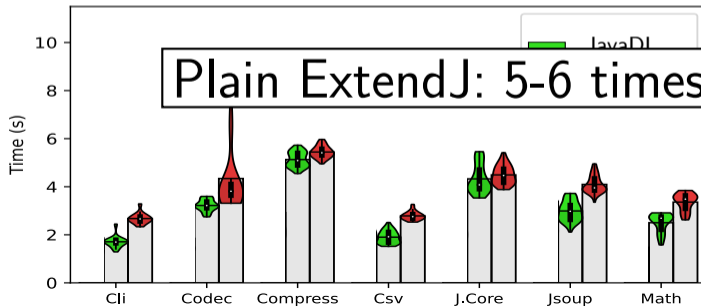
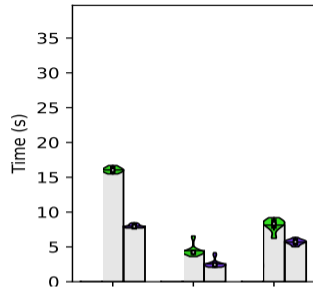
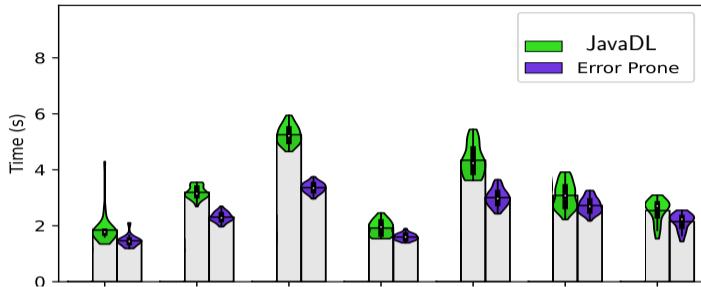
- Doop etc.: hand-written fact extractor, manually aligned with Datalog code
- JavaDL: integrated with JastAdd parser, attributes
 - Syntax-to-Datalog mapping derived automatically
 - Can also export (most) ExtendJ attributes if needed

Bug Checkers Overview

Static Checker Framework					JavaDL		
	Bug Pattern	ID	LOC	Notes	LOC	#Rules	Attrs
Error Prone	Covariant equals()	NonOverridingEquals	116	fix: +16	15	9	D
	Boxed Primitive Constructor	BoxedPrimitiveConstructor	115	fix: +114	9	3	D
	Missing @Override	MissingOverride	82	fix: +2	48	30	D
	Complex Operator Precedence	OperatorPrecedence, Unnecessary-Parentheses	99	fix: +39	37	37	
	Useless Type Parameter	TypeParameterUnusedInFormals	108		27	18	D
	== with reference	ReferenceEquality	97		88	48	D,T
SpotBugs	Covariant equals()	EQ_ABSTRACT_SELF	541	share 18	15	9	D
	Field never written to	UWF_UNWRITTEN_FIELD, UWF_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD	1032	share 14	51	47	D
	Missing switch default	SF_SWITCH_NO_DEFAULT	289	share 4	21	8	D,T
	Expose internal representation	EI_EXPOSE_REP, MS_EXPOSE_REP, EI_EXPOSE_REP2, EI_EXPOSE_STATIC_REP2	138		29	20	D
	Naming convention violation	NM_METHOD_NAMING_CONVENTION, NM_FIELD_NAMING_CONVENTION, NM_CLASS_NAMING_CONVENTION	499	share 12	17	10	
	Boxed primitive constructor	DM_NUMBER_CTOR, DM_STRING_CTOR	1415	share 52	9	3	D,T

Attr: ExtendJ Attributes used; D: DECL, T: TYPE

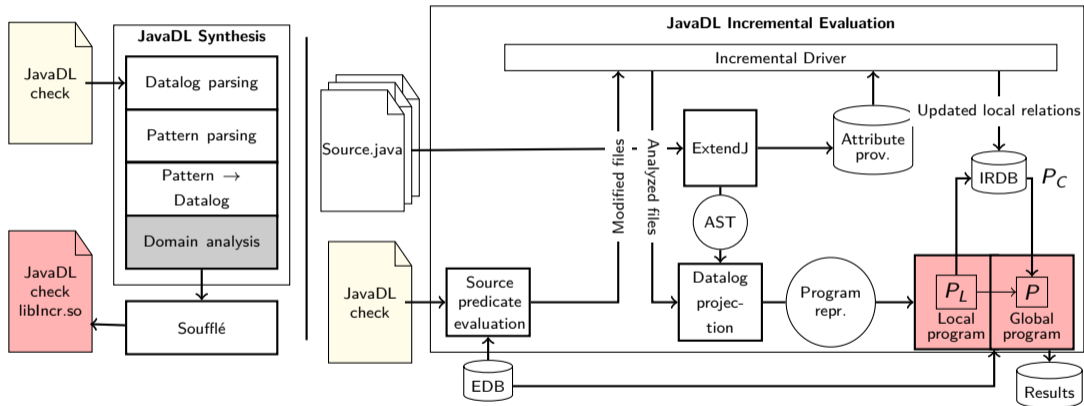
Performance



Incremental Evaluation in JavaDL

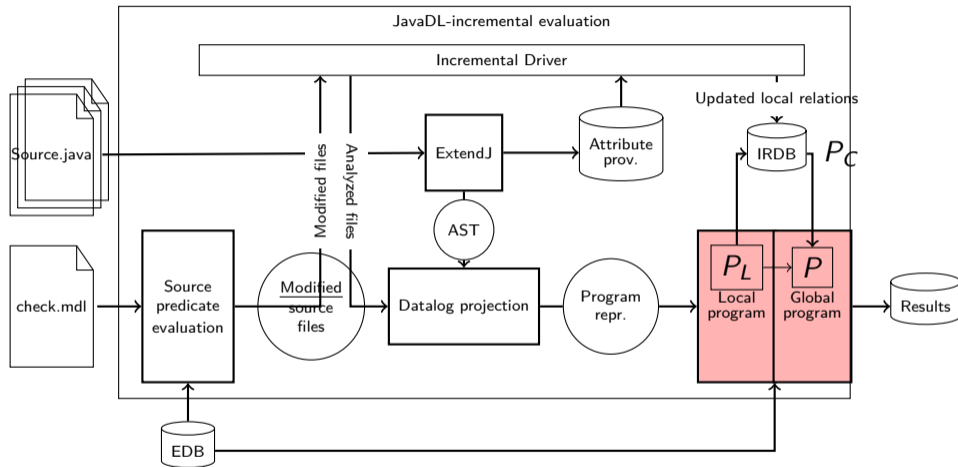
- Scenario: Running bug checkers during Continuous Integration
 - Typical programs can consist of hundreds / thousands of source files
 - For most commits, few (if any) change
- ⇒ Reuse earlier results?
- Incremental at file level

Incremental Evaluation in JavaDL



- Incremental at file level

Incremental JavaDL



- Incrementalise at file level
- Track: which file requires reanalysing which file
- Separate: local vs. global parts of analysis
- Challenges:

JavaDL: Automatic Rule Split

User Spec

```
NEWSTRING(t, f, l, c) :- n [new String( #v )], TYPE( #v , t), SRC(n, l, c, -, -, f).  
STRINGCLASS(s) :- s [.. class String { .. }],  
SRC(s, -, -, -, -, "java/lang/String.class").  
BADNEWSTRING(f, l, c) :- NEWSTRING(t, f, l, c), STRINGCLASS(t).
```

MetaDL IR After Rewriting

```
// Local rule
```

```
NEWSTRINGL(t, f, l, c, u) :-  
    n [new String( #v )], TYPE( #v , t),  
    SRC(n, l, c, -, -, f), u = cu(n).  
OUTPUT('NEWSTRINGL).
```

```
// Local rule
```

```
STRINGCLASSL(s, u) :- s [.. class String { .. }],  
SRC(s, -, -, -, -, "java/lang/String.class"), u = cu(s).  
OUTPUT('STRINGCLASSL).
```

```
INPUT('STRINGCLASSC).
```

```
STRINGCLASS(s) :- STRINGCLASSL(s, -).
```

```
STRINGCLASS(s) :- STRINGCLASSC(s, -).
```

```
INPUT('NEWSTRINGC).
```

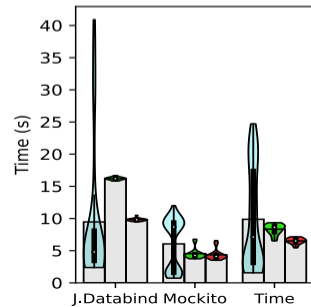
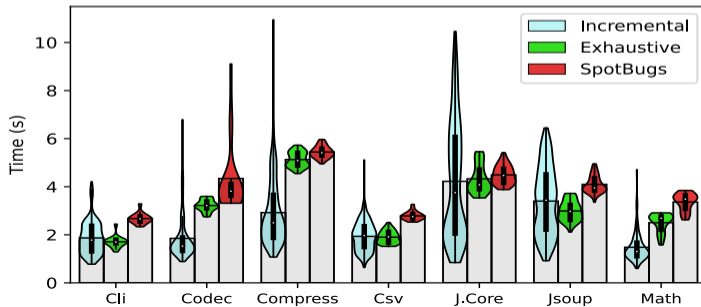
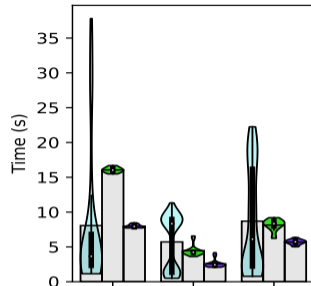
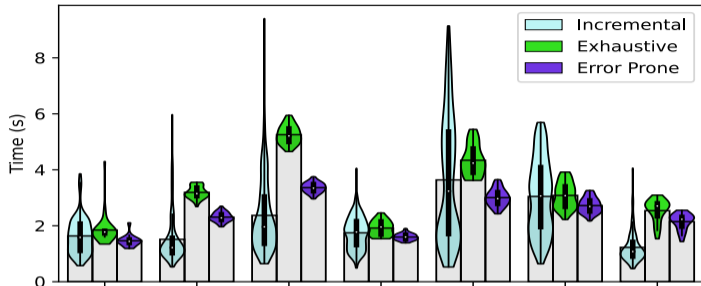
```
NEWSTRING(t, f, l, c) :-  
    NEWSTRINGL(t, f, l, c, -).
```

```
NEWSTRING(t, f, l, c) :-  
    NEWSTRINGC(t, f, l, c, -).
```

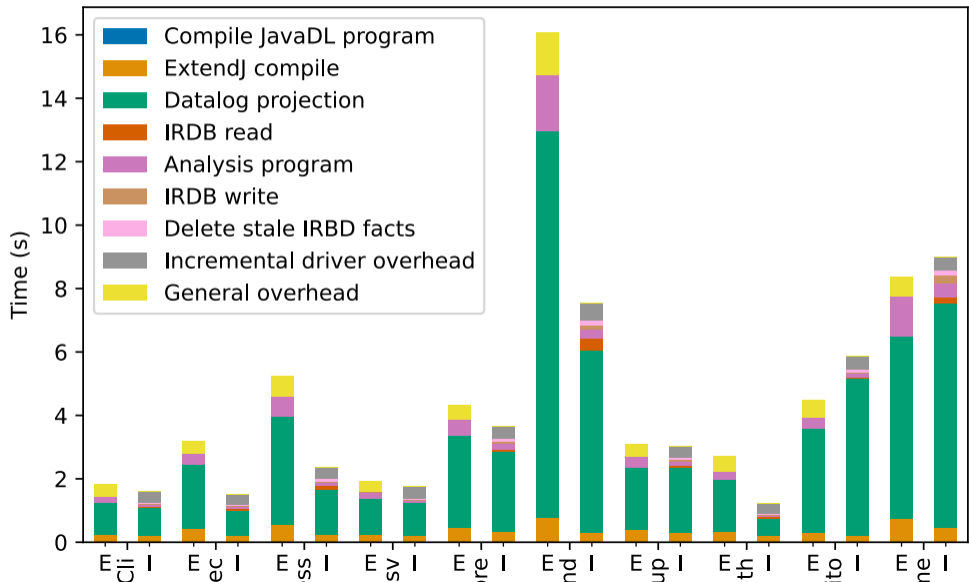
```
// Global rule
```

```
BADNEWSTRING(f, l, c) :-  
    NEWSTRING(t, f, l, c), STRINGCLASS(t).
```

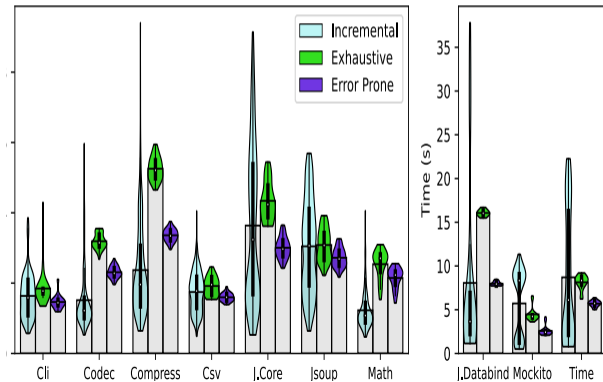
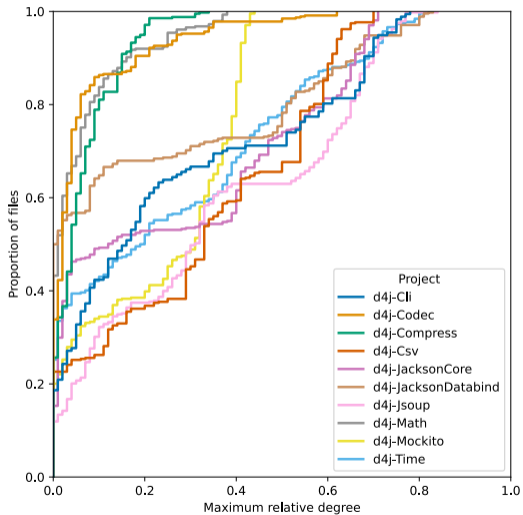
Performance



Where the Time Goes (Error Prone)

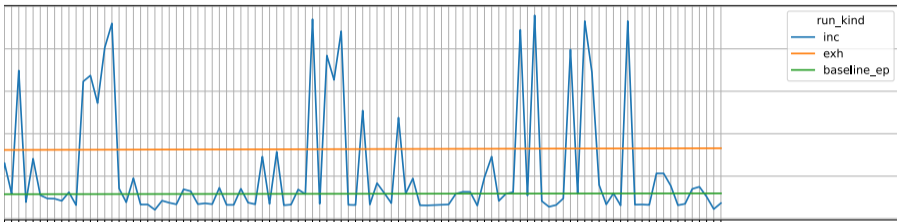


Connectivity

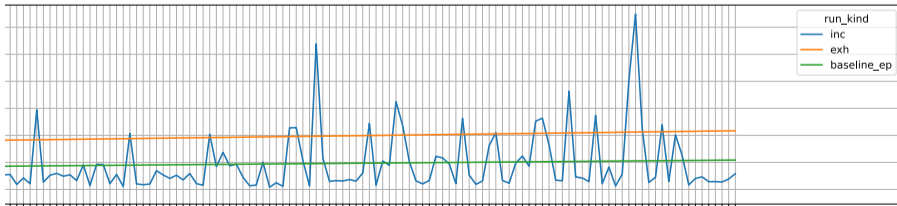


Incrementality: Execution times per commit

Benchmark: **Jackson Databind**



Benchmark: **Math**



Summary

- JavaDL: Syntactic Pattern Matching + Datalog
- Based on:
 - ExtendJ
 - Soufflé
- Competitive performance against state-of-the-practice chckers
 - Can't beat hand-written JastAdd checkers due to cost of copying, though
- **Incremental evaluation:**
 - Can outperform exhaustive evaluation
 - Automatic incrementalisation (file-level)
 - Automatic dependency tracking across Datalog + ExtendJ
- Analysis language effective, but room for improvement:
 - Careful balancing of semantic vs. syntactic matching
 - Named instead of positional predicate arguments
 - Static AST node type analysis, quality-of-life tooling
 - Simplifications in syntax, built-in predicates

MetaDL: Cross-Cutting Concerns

- Integrating external data sources
 - Clog (MetaDL[C]): partly delegates to Clang AST pattern matcher library (on demand)
- Speed / precision trade-off
 - Erik Prántare: “Decoupling Context Sensitivities From Program Analyses” (MSc thesis)
- Incremental evaluation
 - JavaDL (MetaDL[Java]): Source file-level granularity

Review: Cross-Cutting Challenges

- Features external to analysis? (dynamic data, design docs, ...)
- Learning and adaptation?
- Explainability?
- Scalability?
 - Demand-driven or incremental evaluation
 - Differential Analysis
 - Trading off precision vs. efficiency (widening, context sensitivity)
- ...

Conclusions

- Declarative languages simplify bug pattern descriptions (vs. imperative)
- Practical (effectiveness, execution time, code size)
- Some Approaches:
 - Syntactic Patterns: situationally effective
 - Reference Attribute Grammars: AST / graph perspective; top-down eval
 - Datalog: Relational perspective; bottom-up eval
- Transparent Analysis can enable:
 - Incrementalisation / Demand evaluation
 - Explainability
 - Machine learning-based prioritisation
- **Challenges:**
 - Expressivity vs. Ability to Reflect
 - Provenance vs. Explainability
 - Pattern Matching vs. “semantically equivalent code”?