ShiftLeft

Lund University

Cross-Cutting Concerns in Declarative Program Analysis

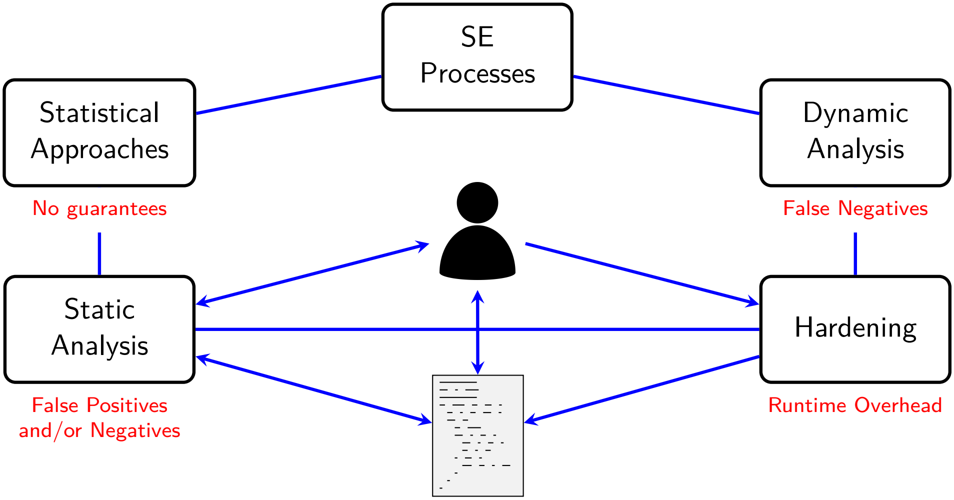
Alexandru Dura
Idriss Riouak
Anton Risberg Alaküla
Christoph Reichenbach

Görel Hedin
Niklas Fors
Emma Söderberg
Erik Präntare

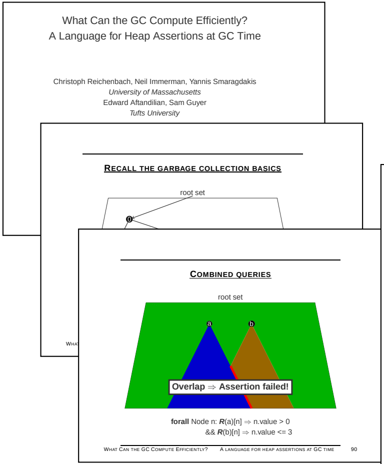
WASP | Wallenberg AI, Autonomous Systems and Software Program

Datalog for Program Analysis: Strengths

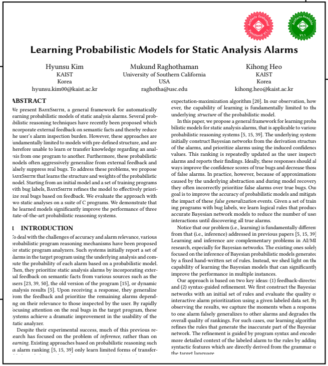
No Free Lunches in Security

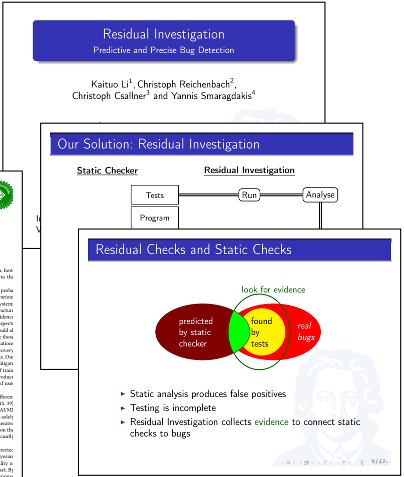


Integration: Beyond Individual Techniques



Static Analysis
+
Statistics





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 sprintsFn
  .method("(
  .callIn
  .whereNot(
(printfFns ++ sprintsFns)}).l
```

- 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_U
|
- (((n) + ((d) - 1)) / (d))
+ DIV_ROUND_UP(n,d)
)
```

- AST Structure detection
- Syntactic rewriting

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

Datalog Queries

```
Reachable(?to
  Reachable(
  Instruction
  VirtualMeth                              se),
  VarPointsTo
  HeapAllocat
  VirtualMeth                              c, ?sname),
  VirtualMeth                              c, ?descriptor),
  basic.MethodLookup(?sname, ?descriptor, ?heaptype,
      ?toMethod).
```

- Logic programming
- Recursion for graph traversal

DOOP, [https://github.com/plast-lab/doop/]

Reference Attribute Grammars (JastAdd)

```
inh Decl                                   st args);
inh Decl                                   rgs);
inh Decl                                   st args);

eq Block.                                  args) {
  if (!lo                                  ))
    retu
  return lookupMethod(name, args);
}
```

- 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

```java
public class MissingDefault ... { ...
  public Description matchSwitch(SwitchTree tree, VisitorState state) {
    Type switchType = ASTHelpers.getType(tree.getExpression());
    if (switchType.asElement().getKind() == ElementKind.ENUM) {
      return NO_MATCH;
    }
    Optional<? extends CaseTest> maybeDefault =
        tree.getCases().stream().filter(c -> c.getExpression() == null).findFirst();
    if (!maybeDefault.isPresent()) { ...
      return description.build();
    } ... }}
```

(**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

```java
public class MissingCasesInEnumSwitch ... { ...
  public Description matchSwitch(SwitchTree tree, VisitorState state) {
    Type switchType = ASTHelpers.getType(tree.getExpression());
    if (switchType.asElement().getKind() != ElementKind.ENUM) {
      return Description.NO_MATCH;
    }
    if (tree.getCases().stream().anyMatch(c -> c.getExpression() == null)) {
      return Description.NO_MATCH;
    }
    ImmutableSet<String> handled =
        tree.getCases().stream()
            .map(CaseTree::getExpression)
            .filter(IdentifierTree.class::isInstance)
            .map(e -> ((IdentifierTree) e).getName().toString())
            .collect(toImmutableSet());
    Set<String> unhandled = Sets.difference(
        ASTHelpers.enumValues(switchType.asElement()), handled);
    if (unhandled.isEmpty()) {
      return Description.NO_MATCH;
    }
    return buildDescription(tree).setMessage(buildMessage(unhandled)).build();
  } ... }
```

Only check the Enum case here

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

Checker #1: **Error Prone**

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

Declarative specification

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

```
public class ASTSwitchStatement ... { ...
public boolean hasDefaultCase() {
  for (ASTSwitchLabel label : this) {
    if (label.isDefault()) {
      return true;
    }}
  return false;
}
public boolean isExhaustiveEnumSwitch() {
  ASTExpression expression = getTestedExpression();
  if (expression.getType() == null) {
    return false;
  }
  if (Enum.class.isAssignableFrom(expression.getType())) {
    Set<String> constantNames = EnumUtils.getEnumMap(
        (Class< extends Enum>) expression.getType()).keySet();
    for (ASTSwitchLabel label : this) {
      constantNames.remove(label.getFirstDescendantOfType(ASTName.class).getImage());
    }
    return constantNames.isEmpty();
  }
  return false;
} ... }
```

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

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

Checker #2: **PMD**

(**Task 1**) Filter by AST node type

```
from SwitchStmt switch, EnumType enum, EnumConstant missing
where
  switch.getExpr().getType() = enum and
  missing.getDeclaringType() = enum and
  not switch.getAConstCase().getValue() = missing.getAnAccess() and
  not exists(switch.getDefaultCase())
select switch
```

(**Task 3**) Does there exist a missing value?

(**Task 2**) Lacking `default`?

`enum` switches only

Checker #3: **CodeQL**

Checker #4: **ExtendJ**

```
aspect Shared_SwitchDefault {
  inh SwitchStmt Case.enclosingSwitchStmt();
  eq Program.getChild().enclosingSwitchStmt() = null;
  eq SwitchStmt.getChild().enclosingSwitchStmt() = this;

  coll HashSet<ConstCase> SwitchStmt.validConstCases() root SwitchStmt;
  ConstCase contributes this
    when enclosingSwitchStmt() != null
         && typeProblems().isEmpty() && nameProblems().isEmpty()
    to SwitchStmt.validConstCases()
    for enclosingSwitchStmt();

  syn boolean SwitchStmt.isFullyMatchedEnum() = getExpr().type().isEnumDecl()
    && validConstCases().size() ==
       ((EnumDecl)getExpr().type()).enumConstants().size();
}

aspect JDL_SwitchDefault {
  SwitchStmt contributes error("Missing default case")
    when defaultCase() == null && !isFullyMatchedEnum()
    to CompilationUnit.javadlProblems();
}
```

(**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

```
SwitchWithDefault(s) :- s [switch ( #_ ) { .. default: .. }] .
Switch(s) :- s [switch ( #_ ) { .. }] .
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 1**) Find switch statements

(**Task 2**) Is Switch but not SwitchWithDefault

(**Task 3**) Find missing enum members

Add location information

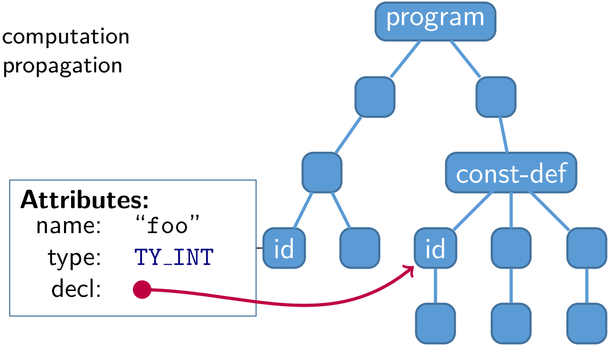
Checker #5: **JavaDL**

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 → parent
  - **Inherited**: ancestor → descendants (automatic forwarding)
  - **Collection**: descendant → 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

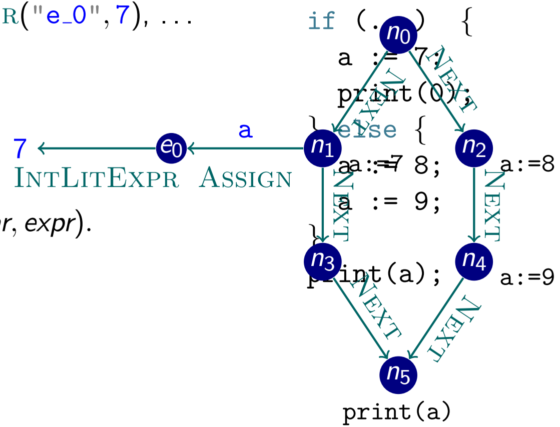
Input Facts

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).

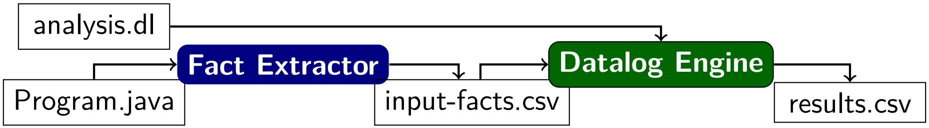
Derived Facts

MAYREACH$(z, var, expr)$:- ASSIGN$(z, var, expr)$.
MAYREACH$(z, var, expr)$:-
    $\neg$ASSIGN$(z, var, _)$,
    MAYREACH$(x, var, expr)$,
    NEXT(x, z).

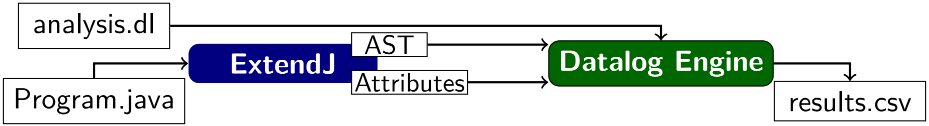


Program Analysis with Datalog

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]

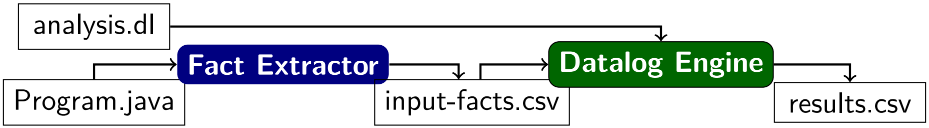
- Use (parts of) ExtendJ as analyser frontend
- Pattern matching on Java source code
  - Expose tree structure, access to subtree root (`r`):

    ```
    r  [#e + 0]
    ```
  - Automatically derive: pattern grammar ← ExtendJ grammar
  - Highly ambiguous grammar
- Expose additional information from ExtendJ:

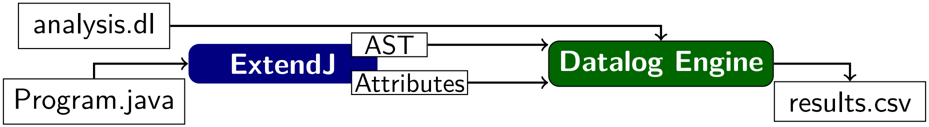
$\mathrm{TYPE}(n, \tau)$	n has type τ (AST node representing the class/type)
$\mathrm{DECL}(n_1, n_2)$	Declaration site for n_1
$\mathrm{INT}(n, i)$	Integer value i of n, if n is int literal
$\mathrm{ID}(n, s)$	String representation of identifier, if n is identifier
$\mathrm{SRC}(n, \ldots)$	Source location (line, column, file)
$\mathrm{SUCC}(n, m)$	CFG successor/predecessor

Program Analysis with Datalog

Common architecture (Doop, CodeQL etc.):



JavaDL:



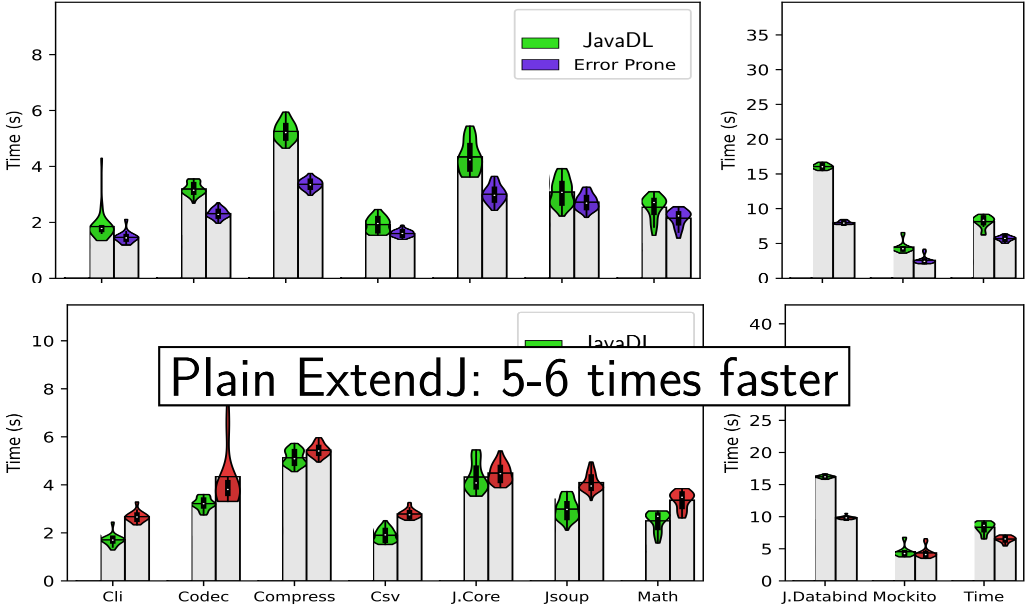
- Doop etc.: hand-written fact extractor, manually aligned with Datalog code
- JavaDL: integrated with JastAdd parser, attributes
  - Syntax-to-Datalog mapping derived automatically
  - Can also export (most) ExtendJ attributes if needed

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_CONVENTIOI 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



Plain ExtendJ: 5-6 times faster

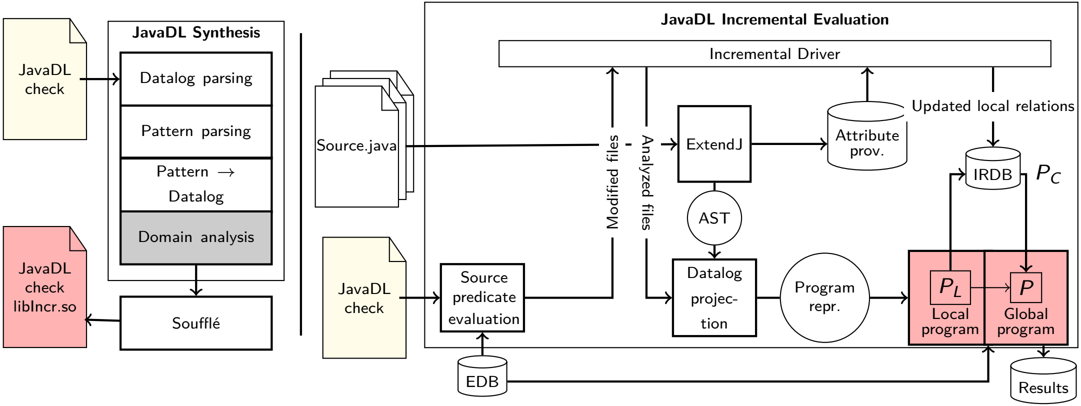
Incremental Evaluation in JavaDL

- Scenario: Running bug checkers during Continuous Integration
- Typical programs can consist of hundreds / thousands of source files
- For most commits, few (if any) change

$\Rightarrow$ Reuse earlier results?

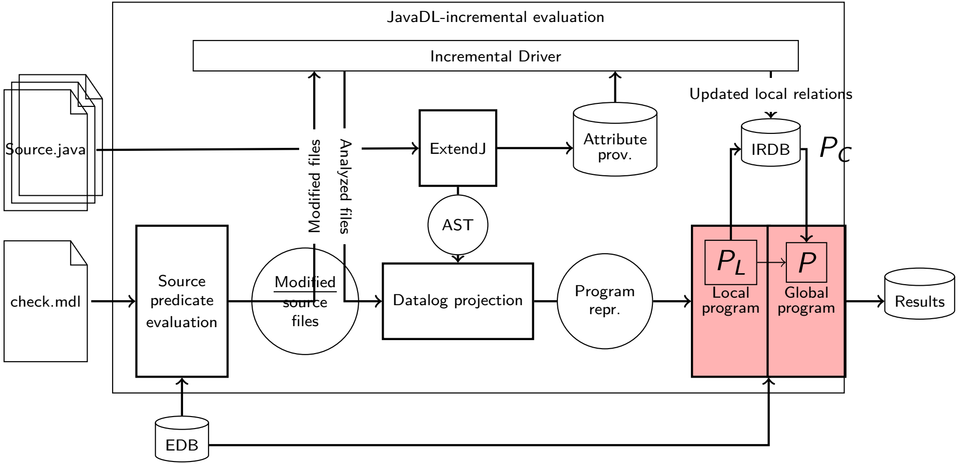
  - Incremental at file level

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

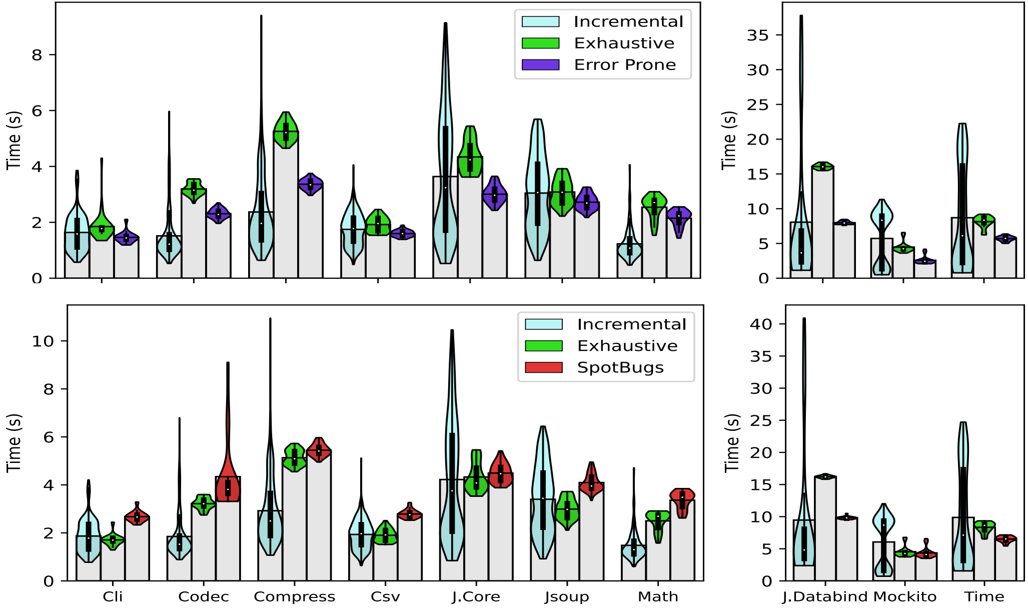
```
// Local rule
NewStringL(t, f, l, c, u) :-
        n [new String( #v )] , TYPE( #v , t),
        SRC(n, l, c, _, _, f), u = cu(n).
OUTPUT('NewStringL).

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

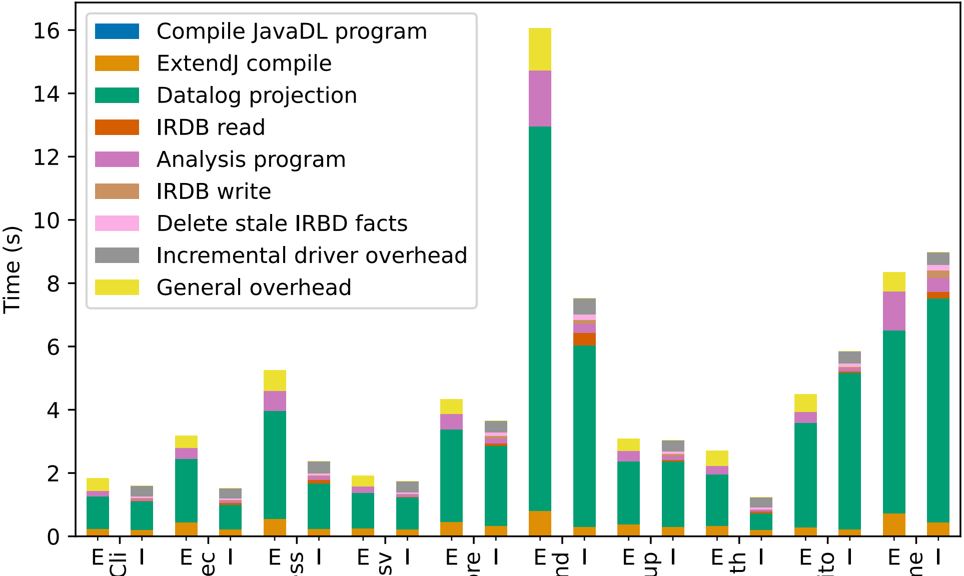
```
INPUT('StringClassC).
StringClass(s) :- StringClassL(s, _).
StringClass(s) :- StringClassC(s, _).
INPUT('NewStringC).
NewString(t, f, l, c) :-
        NewStringL(t, f, l, c, _).
NewString(t, f, l, c) :-
        NewStringC(t, f, l, c, _).

// Global rule
BadNewString(f, l, c) :-
   NewString(t, f, l, c), StringClass(t).
```

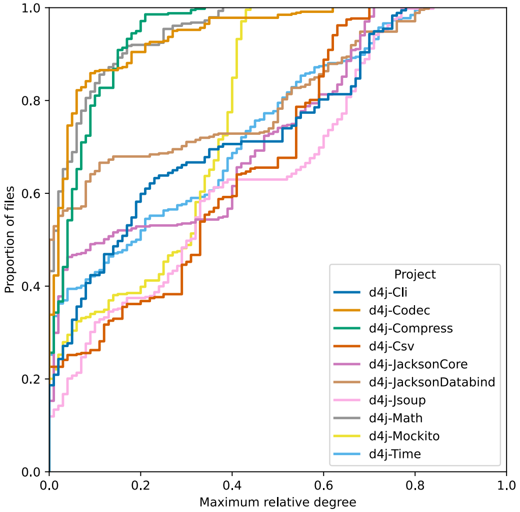
Performance

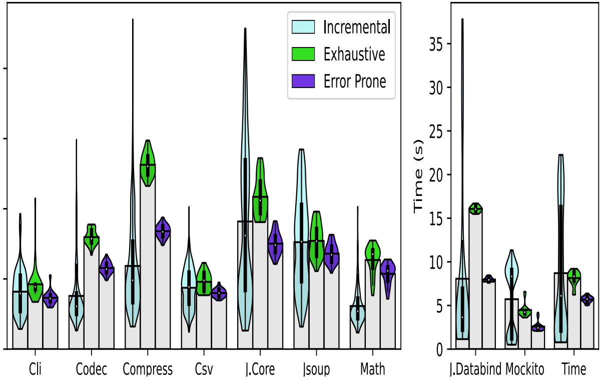


Where the Time Goes (Error Prone)



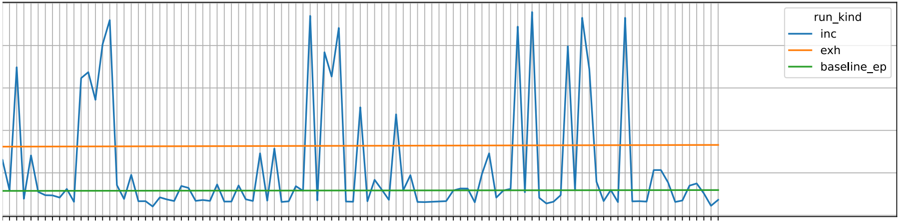
Connectivity



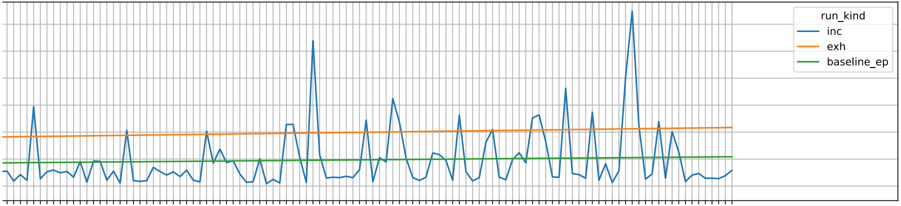


Incrementality: Execution times per commit

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 Präntare: "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 priorisation
- **Challenges**:
  - Expressivity vs. Ability to Reflect
  - Provenance vs. Explainability
  - Pattern Matching vs. "semantically equivalent code"?