Java Code Analysis with Database and Domain Specific Language

www.huawei.com

Presenter：Pan Linjie or 潘临杰 or денис

Organization：Huawei Cloud

Date：2023-10-09

HUAWEI

Personal Profile

- Name: Pan Linjie or 潘临杰 or денис
- Organization: Huawei Cloud
- Work place: Beijing, China
- Education:
  - 2009——2013 Beihang University (BUAA) Bachelor
  - 2013——2016 Beihang University (BUAA) Master
  - 2016——2021 Institute of Software, Chinese Academy of Sciences (ISCAS) Doctor
- Research interests: Static Analysis, Software Testing, Constraint Solving
- Personal page: https://github.com/pangeneral

Content

How to Perform Static Analysis?

Start from an Example

An Example of check rule

- 1. Define a check rule: *Case* should end with break or *$FALL-THROUGH$* comment
- 2. Implement the rule: write a checker
- 3. Scan the code

Right case

```
switch (label) {
    case 0:
    case 1:
        System.out.println("1");
        // $FALL-THROUGH$
    case 2:
        System.out.println("2");
        // $FALL-THROUGH$
    case 3:
        System.out.println("3");
        break;
    default:
        System.out.println("Default case!");
}
```

Wrong case

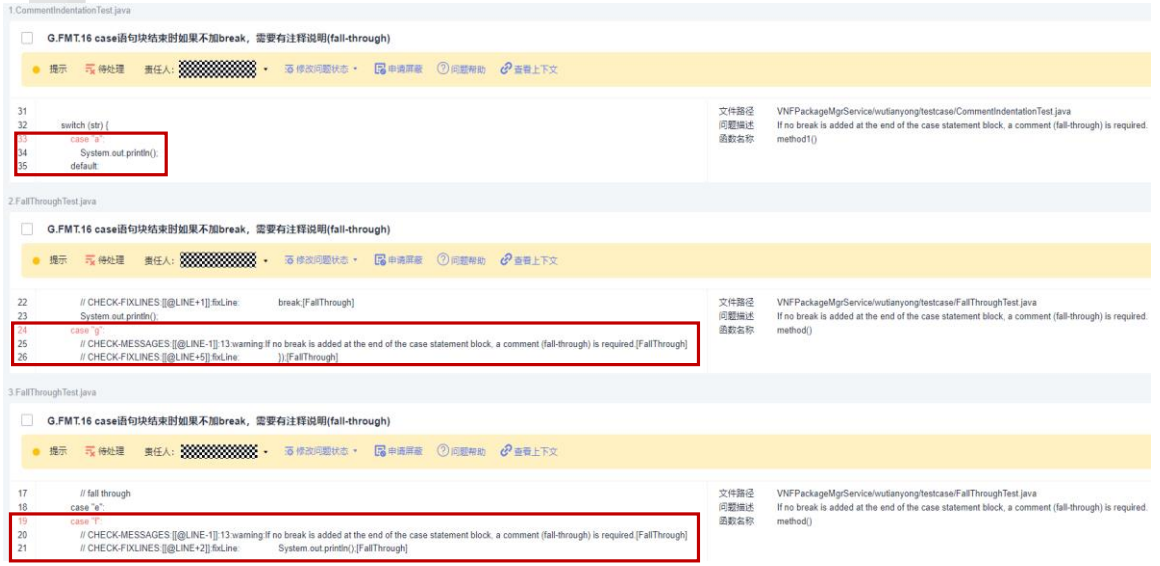
```
switch (label) {
    case 0:
    case 1:
        System.out.println("1");
    case 2:
        System.out.println("2");
    case 3:
        System.out.println("3");
        break;
    default:
        System.out.println("Default case!");
}
```

An Example about checker

- 1. Define a check rule: *Case* should end with break or *$FALL-THROUGH$* comment
- 2. Implement the rule: write a checker
- 3. Scan the code

```java
public class FallThroughFixer extends JavaGenericDefectFixer {
    public List<DefectInstance> detectDefectsForFileModel(JavaClass javaClass) {
        CompilationUnit compilationUnit = javaClass.getCompilationUnit();
        compilationUnit.accept(new ASTVisitor() {
            @Override
            public boolean visit(SwitchStatement node) {
                checkFallThrough(node.statements(), javaClass);
                return super.visit(node);
            }
        });
        return getCurrentFileDefectInstances();
    }

    private void checkFallThrough(List<ASTNode> switchStts, JavaClass javaClass) {
        Map<SwitchCase, List<ASTNode>> caseGroup = caseGroup(switchStts);
        CompilationUnit compilationUnit = javaClass.getCompilationUnit();
        SwitchCase lastCase = caseGroup.keySet().stream()
                .max(Comparator.comparingInt(node -> compilationUnit.getLineNumber(node.getStartPosition())))
                .orElse(null);
        caseGroup.forEach((switchCase, nodes) -> {
            if (CollectionUtils.isEmpty(nodes)) {
                return;
            }
            boolean hasBreak = nodes.stream().anyMatch(this::hasBreak);
            if (!hasBreak && !switchCase.equals(lastCase)) {
                ASTNode lastNode = nodes.get(nodes.size() - 1);
                int endLineNum = compilationUnit.getLineNumber(lastNode.getStartPosition() + lastNode.getLength());
                String potentialCmt = getCurrentFileLines().get(endLineNum);
                if (potentialCmt.trim().startsWith("//") || potentialCmt.trim().startsWith("/*")) {
                    return;
                }
                int bugLineNum = compilationUnit.getLineNumber(switchCase.getStartPosition());
                int bugColumnNum = compilationUnit.getColumnNumber(switchCase.getStartPosition()) + 1;
                DefectInstance defectInstance = createADummyDefectInstance(bugLineNum, bugLineNum, bugColumnNum);
                defectInstance.getContext().add("lastNode", lastNode);
                defectInstance.getContext().add("javaClass", javaClass);
                saveDefectInstance(defectInstance);
            }
        });
    }
```

```java
    private boolean hasBreak(ASTNode node) {
        AtomicBoolean hasBreak = new AtomicBoolean(false);
        node.accept(new ASTVisitor() {
            @Override
            public boolean visit(BreakStatement node) {
                hasBreak.set(true);
                return super.visit(node);
            }
        });
        return hasBreak.get();
    }
    private Map<SwitchCase, List<ASTNode>> caseGroup(List<ASTNode> switchStts) {
        Map<SwitchCase, List<ASTNode>> caseGroup = new HashMap<>();
        for (int i = 0; i < switchStts.size(); i++) {
            ASTNode stt = switchStts.get(i);
            if (!(stt instanceof SwitchCase)) {
                continue;
            }
            SwitchCase switchCase = (SwitchCase) stt;
            int index = i + 1;
            List<ASTNode> list = new ArrayList<>();
            ASTNode nextNode = null;
            while (!(nextNode instanceof SwitchCase) && index < switchStts.size()) {
                nextNode = switchStts.get(index);
                if (!(nextNode instanceof SwitchCase)) {
                    list.add(nextNode);
                }
                index++;
            }
            caseGroup.put(switchCase, list);
        }
        return caseGroup;
    }
}
```



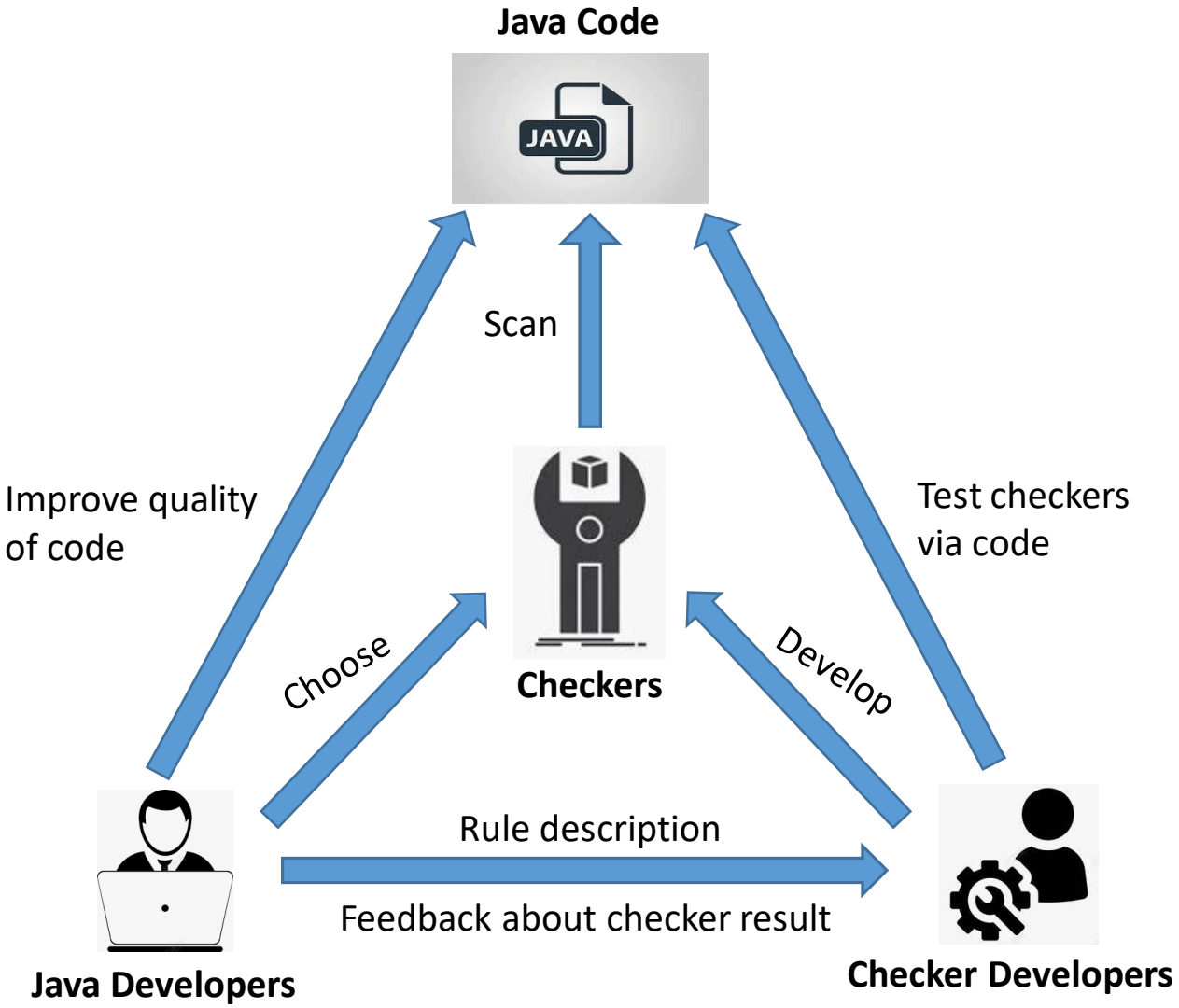
1. Relatively complex checker logic compared with simple rule description
2. Operation on nodes of abstract syntax tree: JavaClass, ASTNode, SwitchStatement

Static Analysis in Industry

- **Targets** of analysis checkers: Java code
  - Huge scale
  - Complex logic
  - Harmful bugs and vulnerabilities
- **Users** of analysis checkers: Java developers
  - Fast iterative development
  - Locate the potential bugs and vulnerabilities
  - Quick feedback about the result of static analysis

No FP! No FN!

- **Owners** of analysis checkers: checker developers
  - Various requirements
  - Customized check rule
  - Develop and modify check rules timely



Simplify the development of checkers

Domain-Specific Language (DSL)

- General-Purpose Language (GPL): JAVA, C++
- Common DSL: Maven, Gradle, HTML, SQL
- Our domain: static analysis
  - Program is very complex
    - Rice theorem: any non-trivial semantic property of a language which is recognized by a Turing machine is **undecidable**
    - The scene of analysis is various
  - Different scenes are similar
    - Fixed program elements: class, method, statement, expression, variable
    - Obtain program elements and perform analysis logic
- The goal of DSL-based analysis
  - Simplify development of analysis rule
  - Reduce duplicate analysis

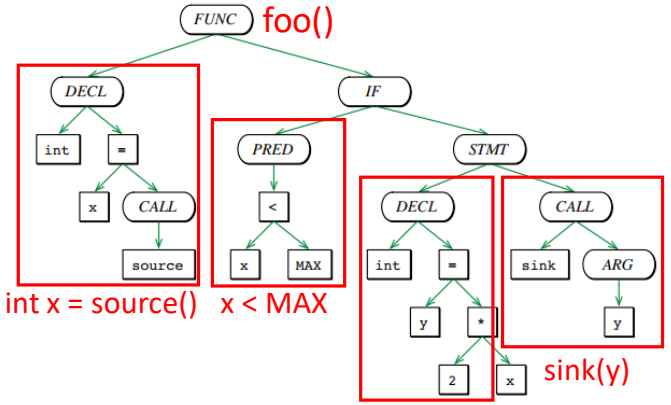
```java
public class FallThroughFixer extends JavaGenericDefectFixer {
    public List<DefectInstance> detectDefectsForFileModel(JavaClass javaClass) {
        CompilationUnit compilationUnit = javaClass.getCompilationUnit();
        compilationUnit.accept(new ASTVisitor() {
            @Override
            public boolean visit(SwitchStatement node) {
                checkFallThrough(node.statements(), javaClass);
                return super.visit(node);
            }
        });
        return getCurrentFileDefectInstances();
    }

    private void checkFallThrough(List<ASTNode> switchStts, JavaClass javaClass) {
        Map<SwitchCase, List<ASTNode>> caseGroup = caseGroup(switchStts);
        CompilationUnit compilationUnit = javaClass.getCompilationUnit();
        SwitchCase lastCase = caseGroup.keySet().stream()
                .max(Comparator.comparingInt(node -> compilationUnit.getLineNumber(node.getStartPosition())))
                .orElse(null);
        caseGroup.forEach((switchCase, nodes) -> {
            if (CollectionUtils.isEmpty(nodes)) {
                return;
            }
            boolean hasBreak = nodes.stream().anyMatch(this::hasBreak);
            if (!hasBreak && !switchCase.equals(lastCase)) {
                ASTNode lastNode = nodes.get(nodes.size() - 1);
                int endLineNum = compilationUnit.getLineNumber(lastNode.getStartPosition() + lastNode.getLength());
                String potentialCmt = getCurrentFileLines().get(endLineNum);
                if (potentialCmt.trim().startsWith("//") || potentialCmt.trim().startsWith("/*")) {
                    return;
                }
                int bugLineNum = compilationUnit.getLineNumber(switchCase.getStartPosition());
                int bugColumnNum = compilationUnit.getColumnNumber(switchCase.getStartPosition()) + 1;
                DefectInstance defectInstance = createADummyDefectInstance(bugLineNum, bugLineNum, bugColumnNum);
                defectInstance.getContext().add("lastNode", lastNode);
                defectInstance.getContext().add("javaClass", javaClass);
                saveDefectInstance(defectInstance);
            }
        });
    }
```

Find caseBlock cb where
cb.lastStatement is not or(break, fallthrough)

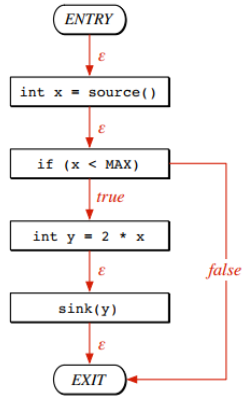
How to represent code?

Abstraction of Code

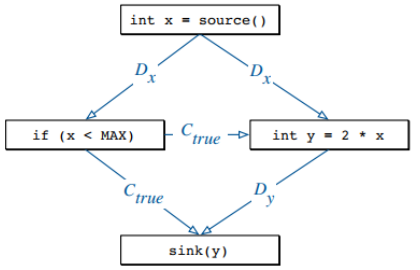
```
void foo()                          1
{                                   2
    int x = source();               3
    if (x < MAX)                    4
    {                               5
        int y = 2 * x;              6
        sink(y);                    7
    }                               8
}                                   9
```



(a) Abstract syntax tree (AST)



(b) Control flow graph (CFG)



(c) Program dependence graph (PDG)

- Representation of code
  - Abstract syntax tree: syntactic structure
  - Control flow graph: execution path
  - Program dependence graph: data and control dependence

About Code Property Graph

- Code property graph: the combination of AST, CFG and PDG [1]

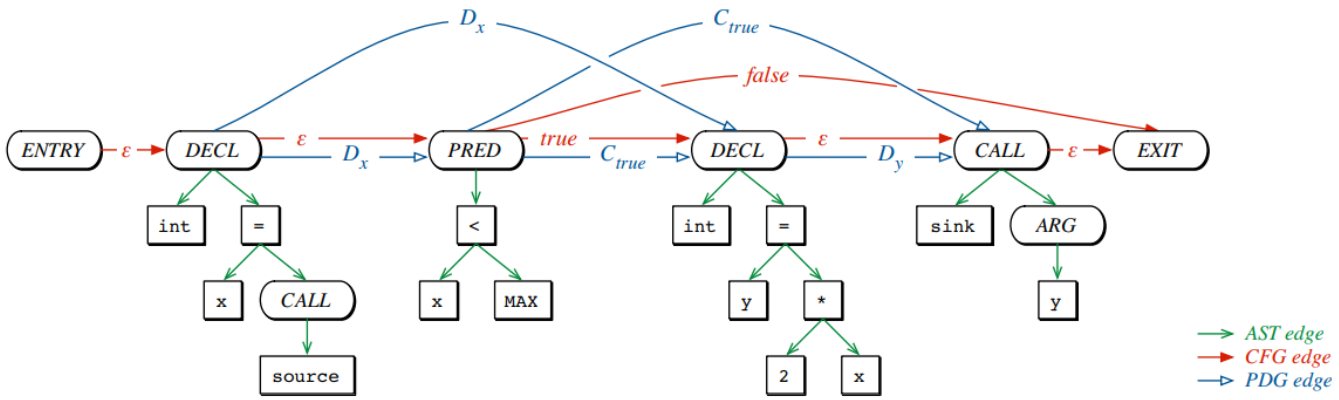
Definition 3. A *code property graph* is a property graph $G = (V, E, \lambda, \mu)$ constructed from the AST, CFG and PDG of source code with

- $V = V_A$,
- $E = E_A \cup E_C \cup E_P$,
- $\lambda = \lambda_A \cup \lambda_C \cup \lambda_P$ and
- $\mu = \mu_A \cup \mu_E$,

where we combine the labeling and property functions with a slight abuse of notation.

Gather information of code together

Extend CPG to contain more information



2014 IEEE Symposium on Security and Privacy

Modeling and Discovering Vulnerabilities with Code Property Graphs

Fabian Yamaguchi*, Nico Golde†, Daniel Arp* and Konrad Rieck*
*University of Göttingen, Germany
†Qualcomm Research Germany

***Abstract*—The vast majority of security breaches encountered today are a direct result of insecure code. Consequently, the protection of computer systems critically depends on the rigorous identification of vulnerabilities in software, a tedious and error-prone process requiring significant expertise. Unfortunately, a single flaw suffices to undermine the security of a system and thus the sheer amount of code to audit plays into the attacker's cards. In this paper, we present a method to effectively mine large amounts of source code for vulnerabilities. To this end, we introduce a novel representation of source code called a *code property graph* that merges concepts of classic program analysis, namely abstract syntax trees, control flow graphs and program dependence graphs, into a joint data structure. This comprehensive representation enables us to elegantly model templates for common vulnerabilities with *graph traversals* that, for instance, can identify buffer overflows, integer overflows, format string vulnerabilities, or memory disclosures. We implement our approach using a popular graph database and demonstrate its efficacy by identifying 18 previously unknown vulnerabilities in the source code of the Linux kernel.**

***Keywords*—Vulnerabilities; Static Analysis; Graph Databases**

I. Introduction

The security of computer systems fundamentally depends on the quality of its underlying software. Despite a long series of research in academia and industry, security vulnerabilities regularly manifest in program code, for example as failures to account for buffer boundaries or as insufficient validation of input data. Consequently, vulnerabilities in software remain one of the primary causes for security breaches today. For example, in 2013 a single buffer overflow in a universal plug-and-play library rendered over 23 million routers vulnerable to attacks from the Internet [26]. Similarly, thousands of users currently fall victim to web-based malware that exploits different flaws in the Java runtime environment [29].

The discovery of software vulnerabilities is a classic yet challenging problem of security. Due to the inability of a program to identify non-trivial properties of another program, the generic problem of finding software vulnerabilities is undecidable [33]. As a consequence, current means for spotting security flaws are either limited to specific types of vulnerabilities or build on tedious and manual auditing. In particular, securing large software projects, such as an operating system kernel, resembles a daunting task, as a single flaw may undermine the security of the entire code base. Although some classes of vulnerabilities reoccurring throughout the software landscape exist for a long time, such as buffer overflows and format string vulnerabilities, automatically detecting their incarnations in specific software projects is often still not possible without significant expert knowledge [16].

As a result of this situation, security research has initially focused on statically finding specific types of vulnerabilities, such as flaws induced by insecure library functions [6], buffer overflows [45], integer overflows [40] or insufficient validation of input data [18]. Based on concepts from software testing, a broader detection of vulnerabilities has then been achieved using dynamic program analysis, ranging from simple fuzz testing [e.g., 38, 42] to advanced taint tracking and symbolic execution [e.g., 2, 35]. While these approaches can discover different types of flaws, they are hard to operate efficiently in practice and often fail to provide appropriate results due to either prohibitive runtime or the exponential growth of execution paths to consider [16, 21]. As a remedy, security research has recently started to explore approaches that assist an analyst during auditing instead of replacing her. The proposed methods accelerate the auditing process by augmenting static program analysis with expert knowledge and can thereby guide the search for vulnerabilities [e.g., 39, 43, 44].

In this paper, we continue this direction of research and present a novel approach for mining large amounts of source code for vulnerabilities. Our approach combines classic concepts of program analysis with recent developments in the field of graph mining. The key insight underlying our approach is that many vulnerabilities can only be adequately discovered by jointly taking into account the structure, control flow and dependencies of code. We address this requirement by introducing a novel representation of source code denoted as *code property graph*. This graph combines properties of abstract syntax trees, control flow graphs and program dependence graphs in a joint data structure. This comprehensive view on code enables us to elegantly model templates for common vulnerabilities using *graph traversals*. Similar to the query in a database, a graph traversal passes over the code property graph and inspects the code structure, the control flow, and the data dependencies associated with each node. This joint access to different code properties enables crafting concise templates for several types of flaws and thereby helps to audit large amounts of code for vulnerabilities.

We implement our approach using a popular graph database and demonstrate its practical merits by designing graph traversals for several well-known vulnerability types, such as buffer overflows, integer overflows, format string vulnerabilities, or memory disclosures. As a show case, we analyze the source code of the Linux kernel—a large and well-audited code base. We find that almost all vulnerabilities reported for

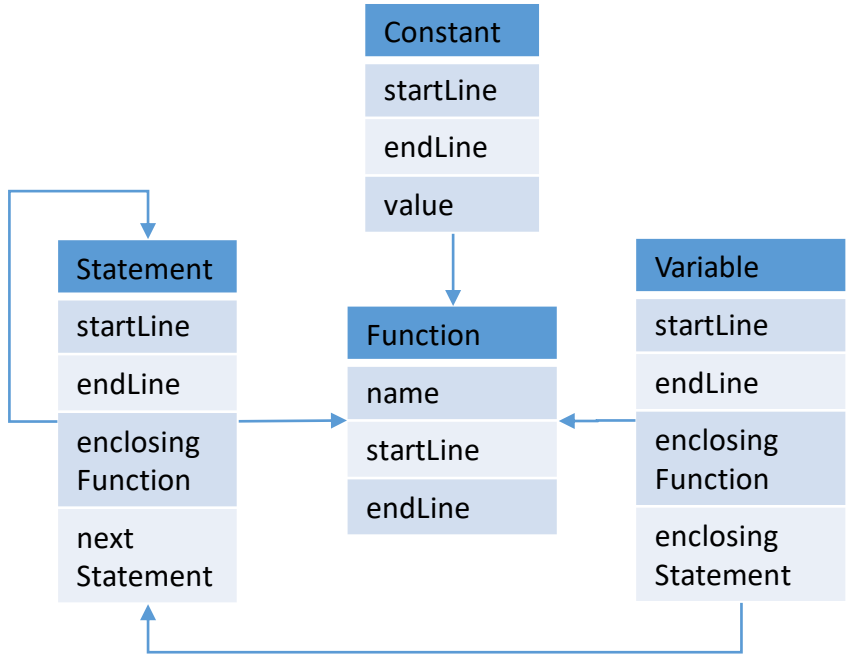
DOI 10.1109/SP.2014.44

How to save code?

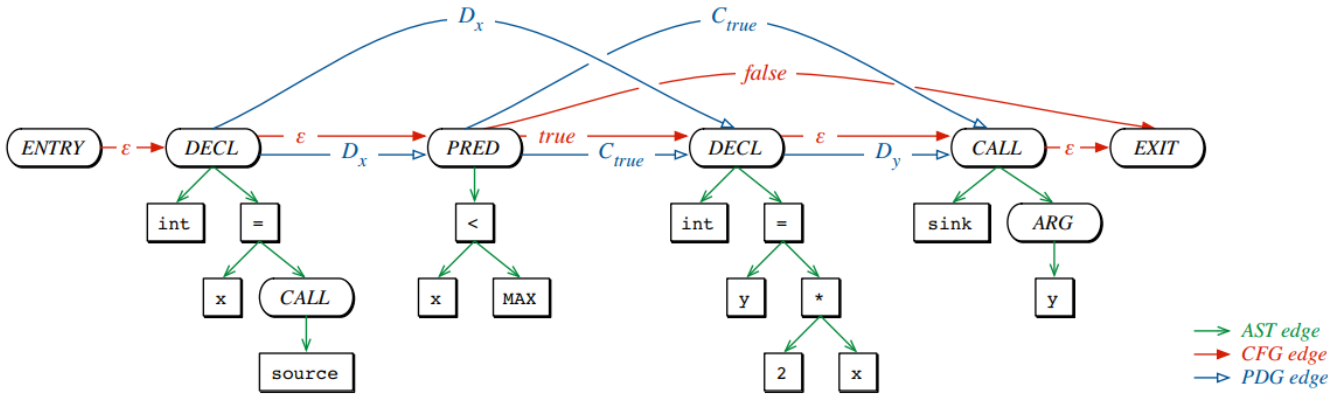
[1] Fabian Yamaguchi, Nico Golde, Daniel Arp, Konrad Rieck: Modeling and Discovering Vulnerabilities with Code Property Graphs. S&P 2014. https://ieeexplore.ieee.org/document/6956589

Relational Database VS Graph Database

```
1  void foo()
2  {
3      int x = source();
4      if (x < MAX)
5      {
6          int y = 2 * x;
7          sink(y);
8      }
9  }
```



	Relational Database	Graph Database
Data organization	Table and relation	Node and edge
Strength	Filter structured data	Process relation

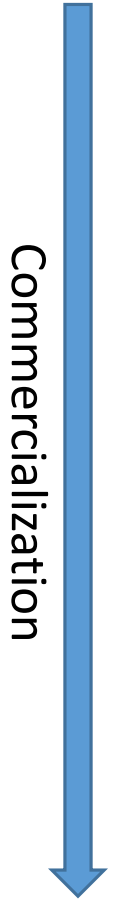


- Which one is better?

How many classes do we have? Is there a path from a statement to another?

Relation is more important than data

Common Graph Database



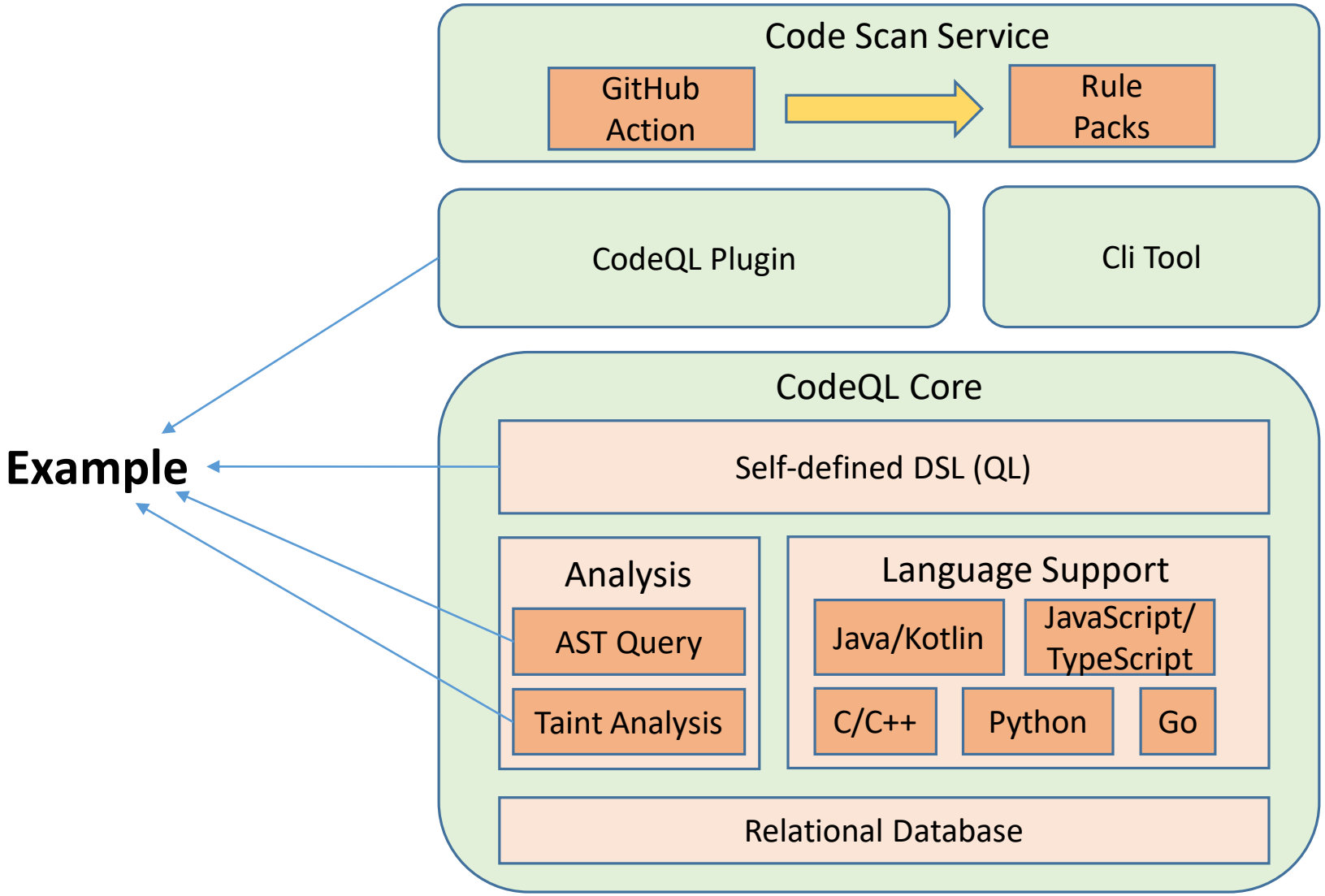
- **TinkerGraph**：light、easy to configure、support Gremlin、no transaction、open-source
- **OverflowDB**：memory friendly、open-source
- JanusGraph：support huge graph、concurrent transaction、open-source of code
- TigerGraph：concurrent graph algorithmbig data process、license for commercial application
- Amazon Neptune：High performance、multiple levels of security、Must access via virtual private cloud
- Neo4j：master-slave topology cluster、real-time schema update、license for commercial application

Content

Tools using Database and DSL

- CodeQL: relational database with self-defined DSL
- Joern: graph database with Gremlin
- Codyze: graph database with Gremlin

Framework of CodeQL



An Example of Query Analysis via CodeQL

- Rule: find all empty blocks
- Structure of checker
  - Metadata
  - Import CodeQL libraries
  - CodeQL class and predicates definition
  - CodeQL variable declaration
  - CodeQL query logic
  - Entrance of query: select clause
- Query kind
  - Alert (@kind problem): highlight issues in specific code locations
  - Path (@kind path-problem): describe the flow of information between a source and a sink

Not Java, Not Java, Not Java

```
1  /**
2   * @name Empty block
3   * @kind problem
4   * @problem.severity warning
5   * @id java/example/empty-block
6   */
7
8  import java
9  import semmle.code.java.dataflow.DataFlow
10
11 from BlockStmt b
12
13 where b.getNumStmt() = 0
14
15 select b, "This is an empty block."
```

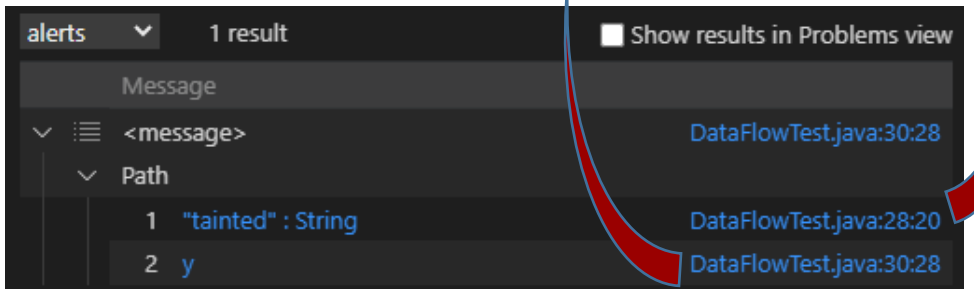
Self-defined DSL called QL

An Example of Taint Analysis via CodeQL

- Import dependencies
- Inherit TaintTracking::Configuration
- Define source and sink
  - Source: string contant “tainted”
  - Sink: the first argument of function call
- Define taint condition
- Code under analysis

```
27      public void dataflowTest() {
28          String x = "tainted";
29          String y = "still" + x;
30          System.out.println(y);
31      }
32  }
33
```

- Analysis result




```
/**
 * @kind path-problem
 */

import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.TaintTracking
import DataFlow::PathGraph

class TaintConf extends TaintTracking::Configuration {
    Quick Evaluation: TaintConf
    TaintConf() {
        this = "Taint Configuration"
    }

    Quick Evaluation: isSource
    override predicate isSource(DataFlow::Node source) {
        source.asExpr().(StringLiteral).getValue() = "tainted"
    }

    Quick Evaluation: isSink
    override predicate isSink(DataFlow::Node sink) {
        exists(Call call |
            sink.asExpr() = call.getArgument(0)
        )
    }
}

from TaintConf config,
DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "<message>"
```

Taint Analysis with Sanitizer and Additional Flow

- Source: string contant ”source”
- Sink: the first argument of function call println
- **Sanitizer: the first argument of function call sanitize**
- **Self-defined flow: from the first argument to the second of function call extraFlow**

```java
public void dataflowTest() {
    // source
    String source = "source";
    String directTainted = "direct" + source;
    String taintedByExtraFlow = "ExtraFlow";
    this.extraFlow(source, taintedByExtraFlow);
    // sink point 1
    System.out.println(taintedByExtraFlow);
    this.sanitize(source);
    String notTaintedByExtraFlow = "notTainted";
    this.extraFlow(source, notTaintedByExtraFlow);
    // sink point 2
    System.out.println(directTainted);
    System.out.println(notTaintedByExtraFlow); // not sink
}

1 usage
public void sanitize(String str) {
}

2 usages
public void extraFlow(String first, String second) {
    System.out.printf("{} is tainted by {}", second, first);
}
```

```
class TaintConf extends TaintTracking::Configuration {
    Quick Evaluation: TaintConf
    TaintConf() {
        this = "Taint Configuration"
    }

    Quick Evaluation: isSource
    override predicate isSource(DataFlow::Node source) {
        source.asExpr().(StringLiteral).getValue() = "source"
    }

    Quick Evaluation: isSink
    override predicate isSink(DataFlow::Node sink) {
        exists(MethodAccess ma |
            sink.asExpr() = ma.getArgument(0) and
            ma.getMethod().getName() = "println"
        )
    }

    Quick Evaluation: isSanitizer
    override predicate isSanitizer(DataFlow::Node node) {
        exists(MethodAccess ma |
            ma.getArgument(0) = node.asExpr() and
            ma.getMethod().getName() = "sanitize"
        )
    }

    Quick Evaluation: isAdditionalTaintStep
    override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node node2) {
        exists(MethodAccess ma |
            ma.getArgument(0) = node1.asExpr() and
            ma.getArgument(1) = node2.asExpr() and
            ma.getMethod().getName() = "extraFlow")
    }
}
```

alerts — 2 results — Show results in Problems view

Message	
<message>	DataFlowTest.java:35:28
Path	
1 "source" : String	DataFlowTest.java:30:25
2 taintedByExtraFlow	DataFlowTest.java:35:28
<message>	DataFlowTest.java:40:28
Path	
1 "source" : String	DataFlowTest.java:30:25
2 directTainted	DataFlowTest.java:40:28

Taint Analysis with State

- Source
  - 1. string constant ”sourceCleaned”, initialized with state ”**cleaned**”
  - 2. string constant ”sourceNotCleaned”, initialized with state ”**notCleaned**”
- Sink: the first arugment of function call println with state ”**cleaned**”, “**notCleaned**” or ”**taintFromCleaned**”
- Sanitizer: the first argument of sanitize with state “**cleaned**”
- Self-defined flow: Taint the second argument, initialized with state “**taintFromCleaned**”, if the first argument has the state “**cleaned**”

Clean the taint

```
28      public void dataflowTest() {
29          // sourceCleaned
30          String sourceCleaned = "sourceCleaned";
31          // sourceNotCleaned
32          String sourceNotCleaned = "sourceNotCleaned";
33          String tainted = "";
34          String notTainted = "";
35
36          // no additional flow
37          this.extraFlow(sourceNotCleaned, notTainted);
38          // additional flow
39          this.extraFlow(sourceCleaned, tainted);
40          this.sanitize(sourceNotCleaned);
41          this.sanitize(sourceCleaned);
42
43          System.out.println(sourceCleaned);
44          System.out.println(sourceNotCleaned); // sink point
45          System.out.println(notTainted);
46          System.out.println(tainted); // sink point
47      }
48
            2 usages
49          public void sanitize(String str) {}
50
            2 usages
51          public void extraFlow(String first, String second) {
52              System.out.printf("{} is tainted by {}", second, first);
53          }
54      }
```

```
class TaintConf extends TaintTracking::Configuration {
    Quick Evaluation: TaintConf
    TaintConf() {
        this = "Taint Configuration"
    }

    Quick Evaluation: isSource
    override predicate isSource(DataFlow::Node source, DataFlow::FlowState state) {
        (source.asExpr().(StringLiteral).getValue() = "sourceCleaned" and state = "cleaned") or
        (source.asExpr().(StringLiteral).getValue() = "sourceNotCleaned" and state = "notCleaned")
    }

    Quick Evaluation: isSink
    override predicate isSink(DataFlow::Node sink, DataFlow::FlowState state) {
        exists(MethodAccess ma |
            sink.asExpr() = ma.getArgument(0) and
            ma.getMethod().getName() = "println" and
            (state = "cleaned" or state = "notCleaned" or state = "taintFromCleaned")
        )
    }

    Quick Evaluation: isSanitizer
    override predicate isSanitizer(DataFlow::Node node, DataFlow::FlowState state) {
        exists(MethodAccess ma |
            ma.getArgument(0) = node.asExpr() and
            ma.getMethod().getName() = "sanitize" and
            state = "cleaned"
        )
    }

    override predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::FlowState state1,
            DataFlow::Node node2, DataFlow::FlowState state2) {
        exists(MethodAccess ma |
            ma.getArgument(0) = node1.asExpr() and
            ma.getArgument(1) = node2.asExpr() and
            ma.getMethod().getName() = "extraFlow" and
            state1 = "cleaned" and state2 = "taintFromCleaned")
    }
}
```

alerts | 2 results | Show results in Problems view

Message	
<message>	DataFlowTest.java:44:28
Path	
1 "sourceNotCleaned" : String	DataFlowTest.java:32:35
2 sourceNotCleaned	DataFlowTest.java:44:28
<message>	DataFlowTest.java:46:28
Path	
1 "sourceCleaned" : String	DataFlowTest.java:30:32
2 sourceCleaned : String	DataFlowTest.java:39:24
3 tainted : String	DataFlowTest.java:39:39
4 tainted	DataFlowTest.java:46:28

CodeQL Plugin **only** available in VSCode

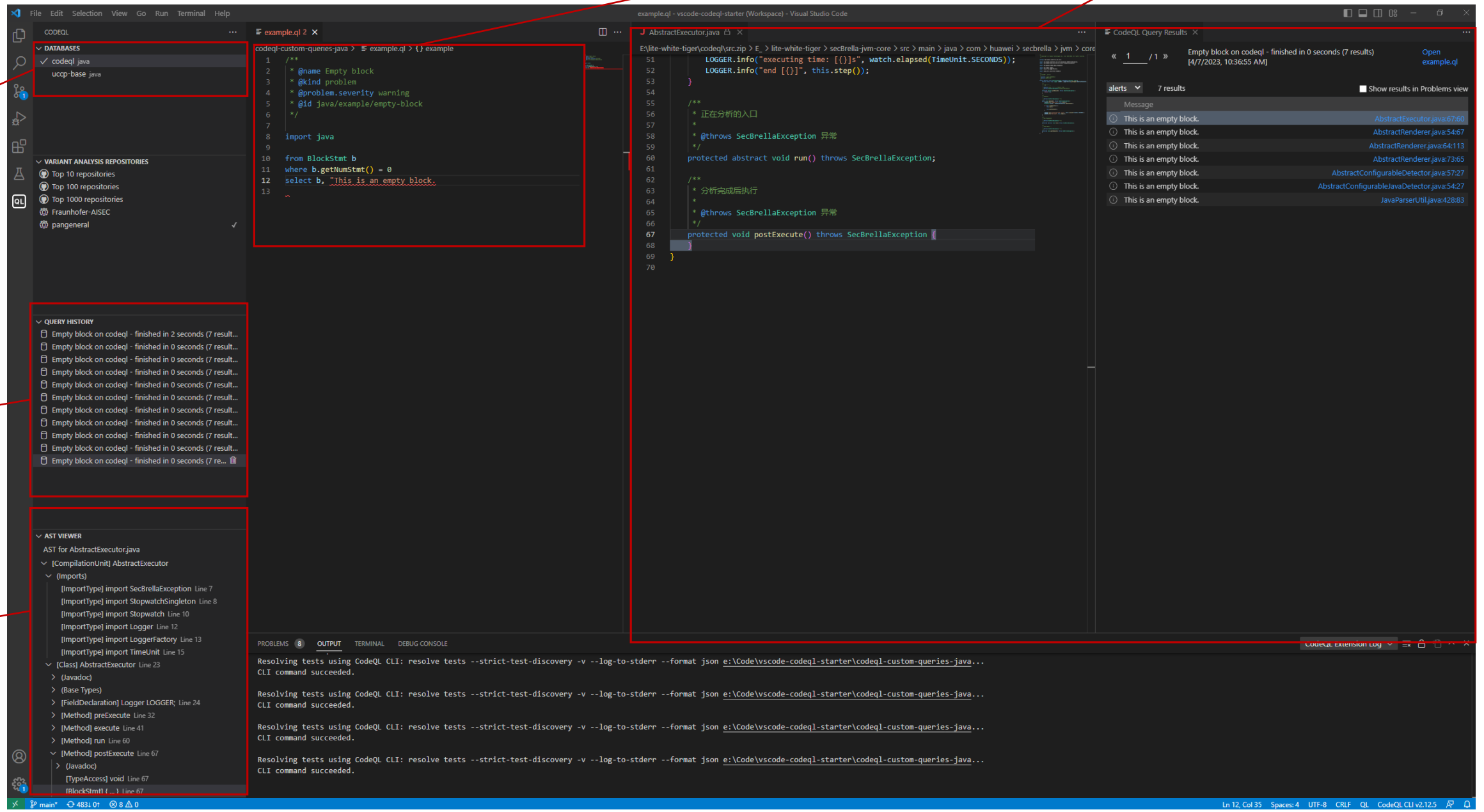
DSL Code (Checker)

View analysis result

Import database under analysis

Query history

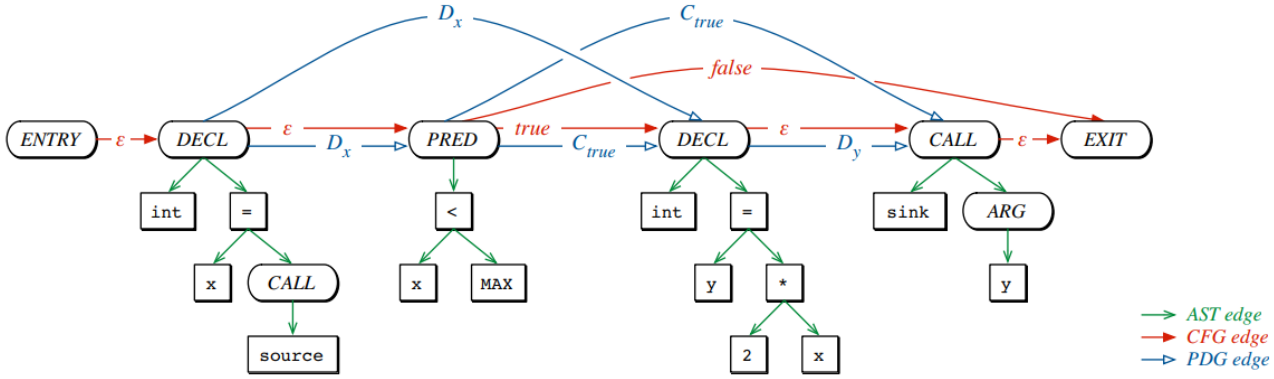
View AST of code under analysis

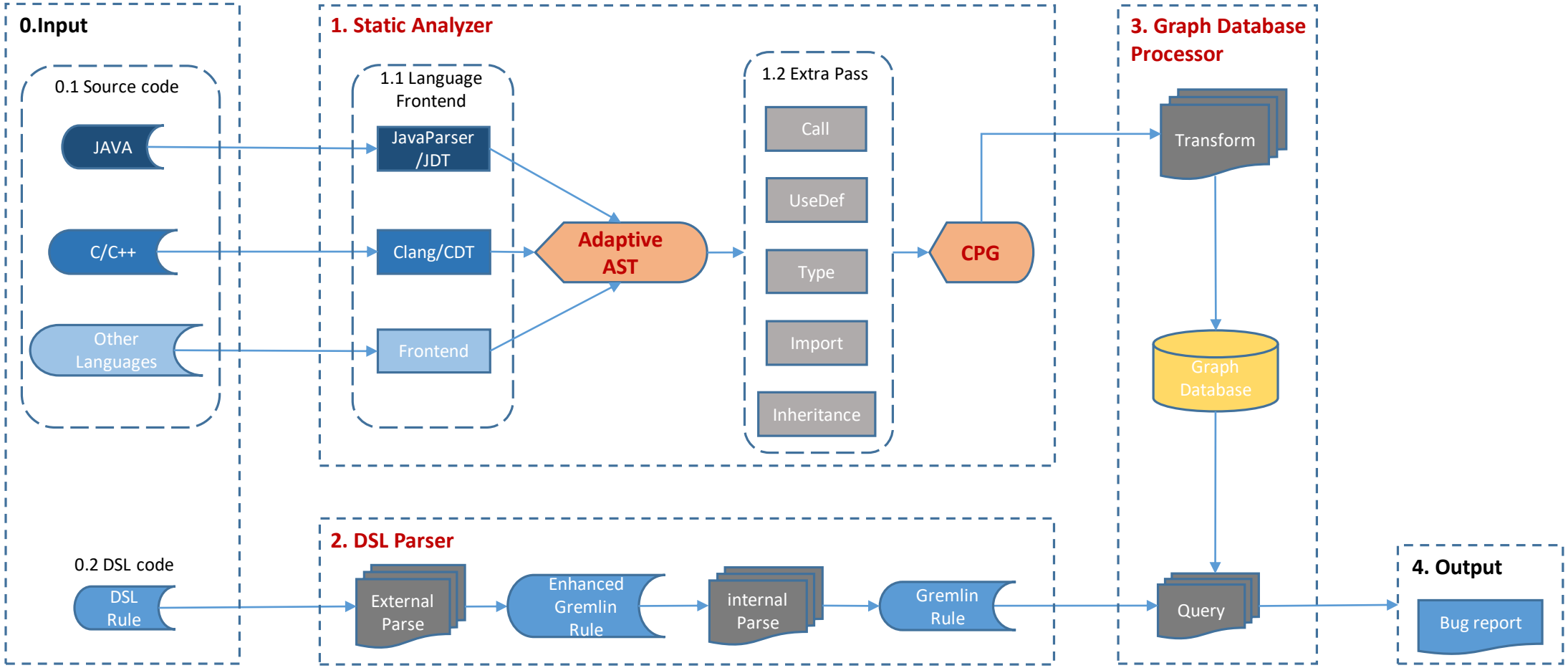


Content

Workflow of Our Analysis Engine

- Graph database with self-defined DSL
- Adaptive AST and CPG
- DSL translation





Adaptive AST and CPG

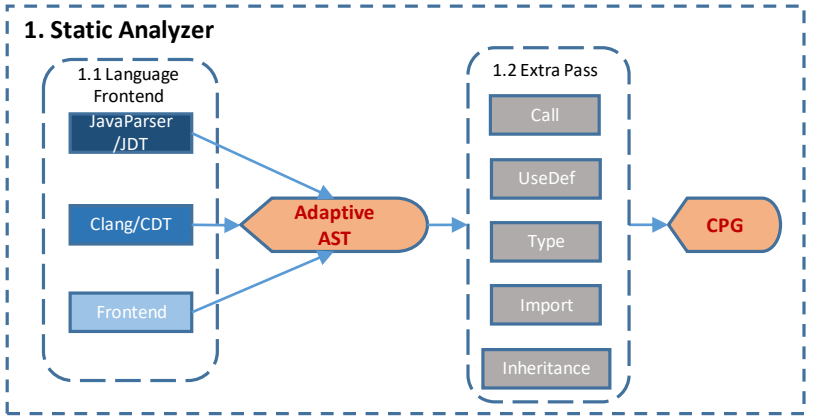
- Java AST Schema
  - Class Declaration
  - Function Declaration
  - Variable Declaration
  - **Import Declaration**
  - …
- C++ AST Schema
  - Class Declaration
  - Function Declaration
  - Variable Declaration
  - **Pointer Type**
  - …

Is it Unified Schema?

- Adaptive AST Schema
  - Class Declaration
  - Function Declaration
  - Variable Declaration
  - **Import Declaration**
  - **Pointer Type**
  - …

Python Schema

Typescript Schema

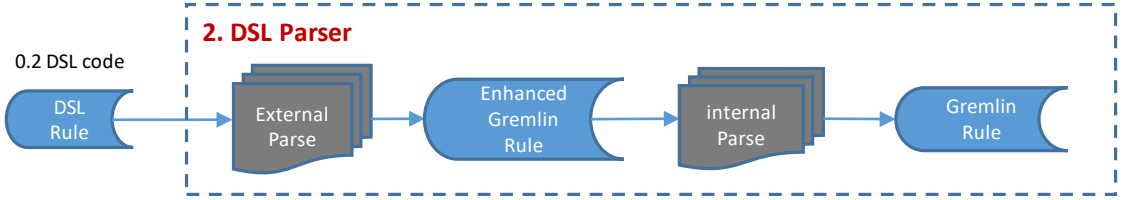


- Code Property Graph
  - AST Nodes and edges
  - **Control flow edges**
  - **Data flow edges**
  - **Additional edges**
    - **Function Call -> Function Declaration**
    - **Variable Usage -> Variable Declaration**
    - **Field Access -> Field Declaration**
    - **Type -> Super Type**
    - **Expression/Variable -> Type**

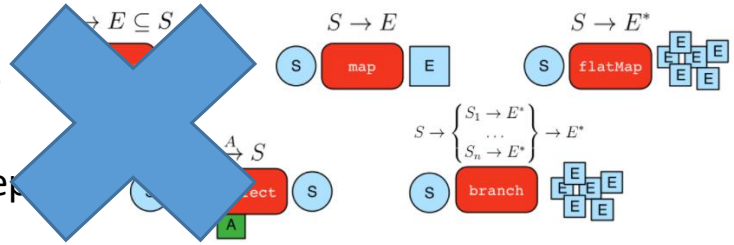
Why do we need extra pass?

1. Build AST file by file
2. Inter-file edges
3. Postprocess after building AST

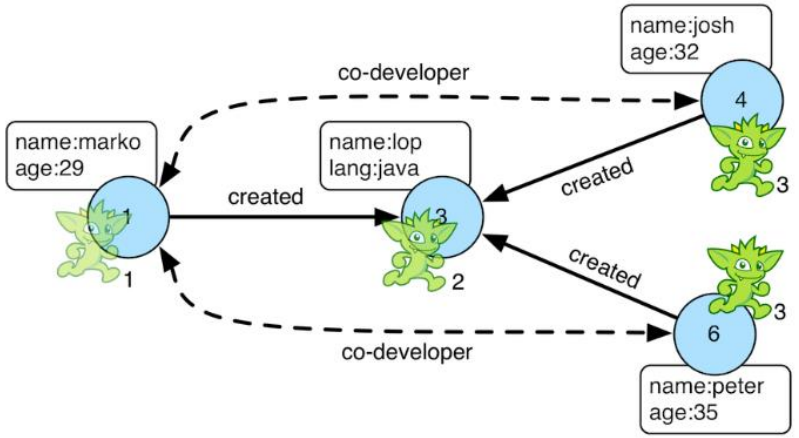
Introduction to Gremlin Query



- Start steps: V() and E() are for query
  - addE(): A ge to start the traversal
  - addV(): Ad rtex to start the traversal
  - inject(): I rts itrary objects to start the traversal
  - **E(): Reads edges from the graph to start the traversal**
  - **V(): Reads vertices from the graph to start the traversal**
- Five kinds of general step
  - MapStep
  - FlatMapStep
  - FilterStep
  - SideEffectStep
  - BranchStep



- End steps
  - hasNext(): deter es whether t e are available results
  - next(): return esult
  - next(n): retu st
  - tryNext(): compo ext()
  - toList(): return all
  - toSet(): return
  - toBulkSet(): ted set
  - fill(collection): results in ded collection
  - iterate(): generate side effects without returning real result



- Query Example

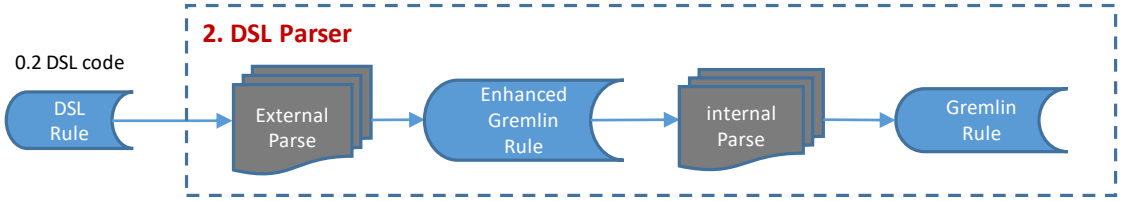
1.Find the co-developers of marko

```
g.V().has("name", "marko").out("created").in("created")
  .has("name", P.neq("marko")).toList()
```

2.Find all developers older than 30

```
g.V().has("lang", "java").in("created")
  .where(__.has("age", P.gt(30))).toSet()
```

DSL Translation



- Define our own DSL
  - Grammar of self-defined DSL

```
queryStmt: rootNodeAttr Satisfy condExpr Semicolon;

rootNodeAttr: (Node | NodeAttr) (Point NodeAttr)* (As alias | alias)?;

condExpr:  (AND | OR) LeftBracket  condExpr (Comma condExpr)+ RightBracket
        | NOT LeftBracket condExpr RightBracket
        | condition;
```

  - Lexer of self-defined DSL

```
Node: 'functionCall' | 'functionDeclaration';

NodeAttr: 'function' | 'base' | 'body'
        | 'arguments' | 'rootBase'
        | 'enclosingClass';

Semicolon: ';';
Point: '.';
```

- Enhanced Gremlin (APIs for JAVA)

```
@GremlinDsl(traversalSource = QueryTraversalSourceDsl")
public interface QueryTraversalDsl<S, E> extends GraphTraversal.Admin<S, E> {
    default GraphTraversal<S, Vertex> function() {
        return this.outE(Elabel.CALL_EDGE).inV();
    }

    default GraphTraversal<S, Vertex> functionCall() {
        return (GraphTraversal<S, Vertex>) this.hasLabel(P.within(LabelGroup.FUNCTION_CALL_LABEL));
    }

    default GraphTraversal<S, Vertex> functionDeclaration() {
        return (GraphTraversal<S, Vertex>) hasLabel(P.within(LabelGroup.FUNCTION_LABEL));
    }
}
```

- Original Gremlin

Example of Multi-Layered DSL

- No debug code
  - It is a Function
  - Its name starts with ”debug”
  - It has only one parameter
  - The type of parameter is java.util.List”

- Original Gremlin

```
List<Vertex> vertexList = (List<Vertex>) this.graphTraversal
    .hasLabel(P.within(AbstractQuery.functionLabel))
    .has(VKey.NAME, TextP.startingWith("debug"))
    .where(__.outE(ELabel.AST_EDGE).has(EKey.RELATION,
            EValue.FUNC_DECL_PARAMETER)
        .inV().count().is(1))
    .where(__.outE(ELabel.AST_EDGE).has(EKey.RELATION,
            EValue.FUNC_DECL_PARAMETER)
        .inV().outE(ELabel.AST_EDGE).has(EKey.RELATION,
            EValue.NODE_TYPE)
        .inV().has("name", "java.util.List"))
    .toList();
```

encapsulate

- Enhanced Gremlin

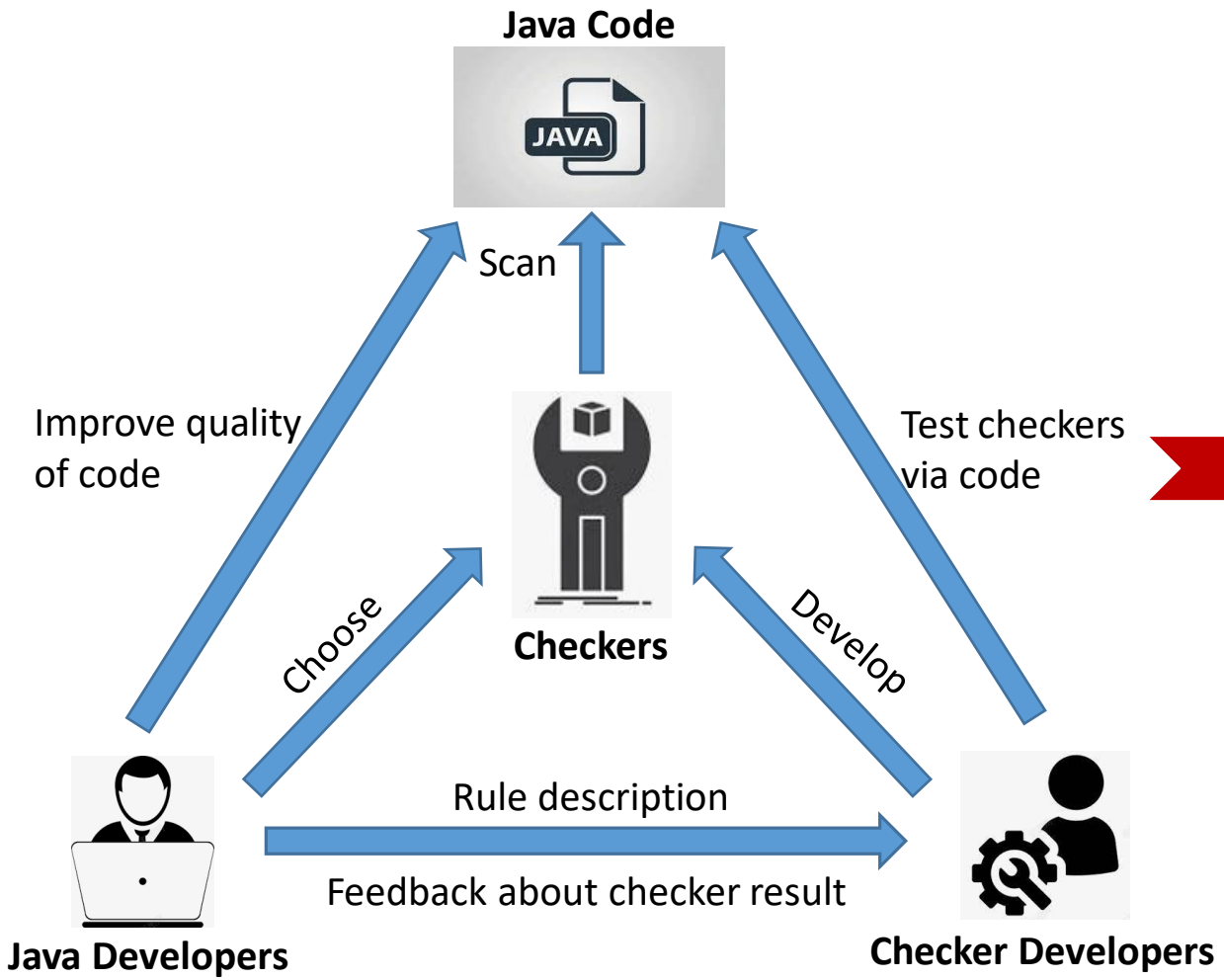
```
public List<Vertex> leftOverDebug() {
    return graphTraversal
        .function()
        .has("name", TextP.startingWith("debug"))
        .where(__.argument().count().is(1))
        .where(__.argument().type()
                .has("name", "java.util.List"))
        .toList();
}
```

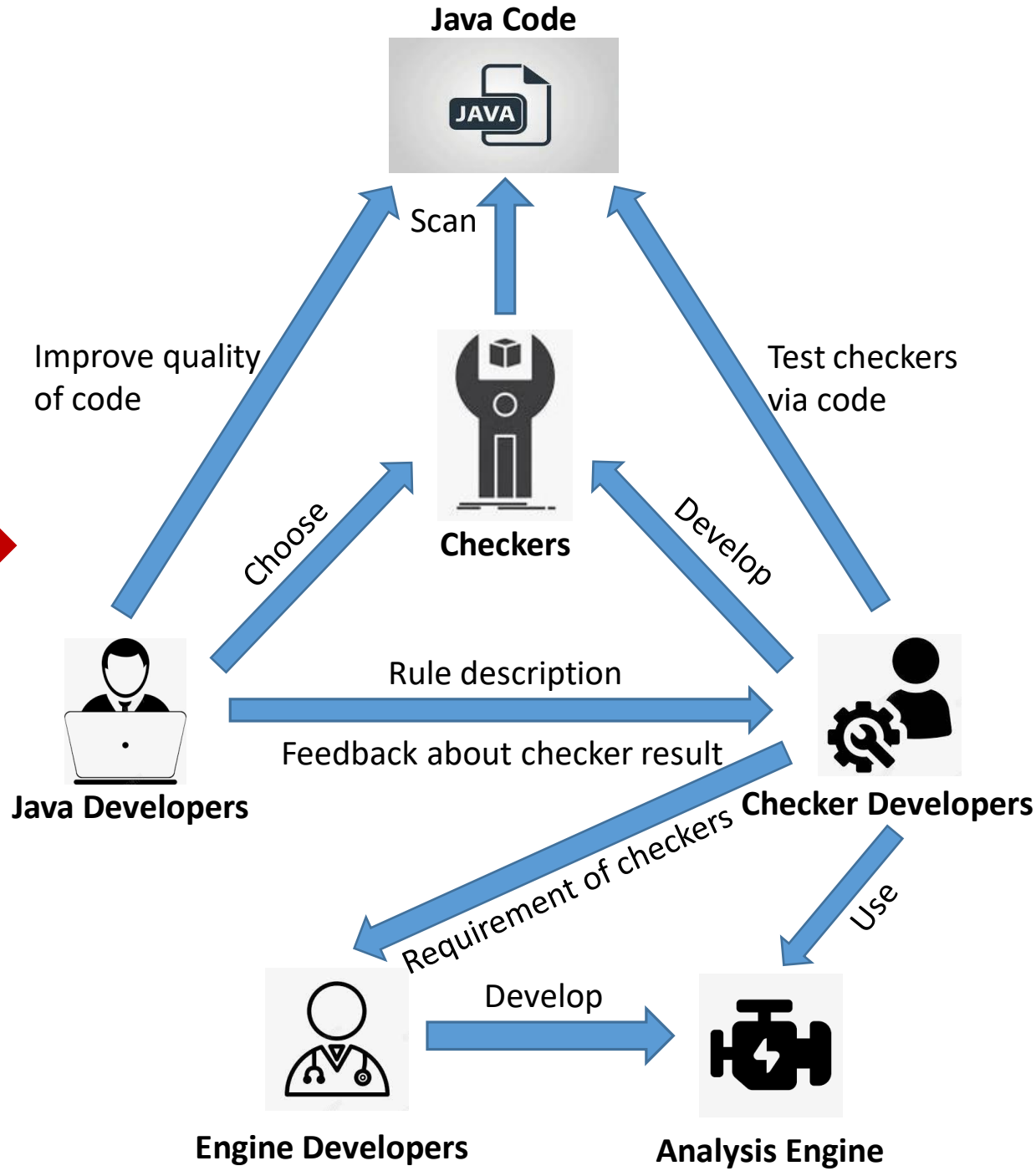
encapsulate

- Self-defined DSL

```
functionDeclaration where
    and(name startWith "debug",
        arguments.size() == 1,
        arguments[0].type.name == "java.util.List");
```

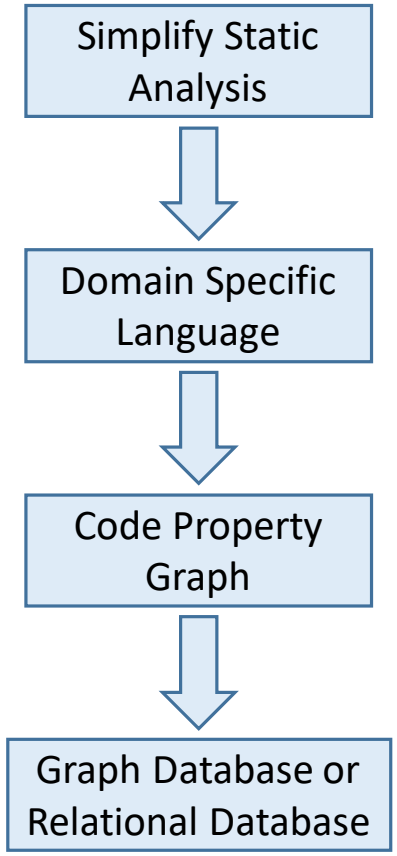
Future of Static Analysis



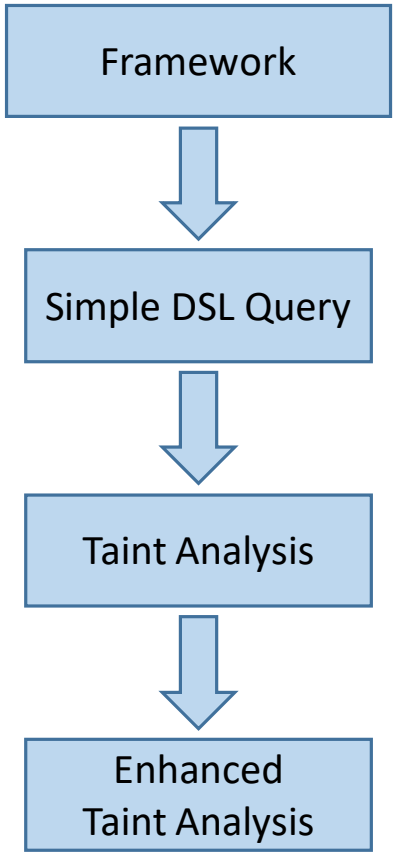


Summary

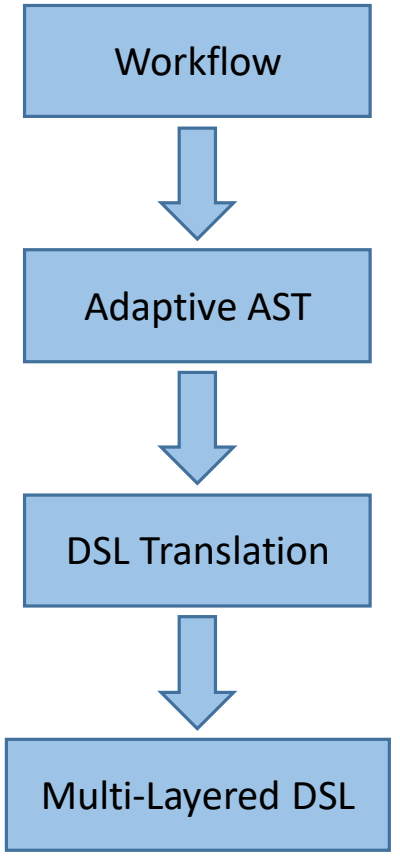
- Background



- CodeQL



- Our work



Thank You!

Спасибо!

谢谢！