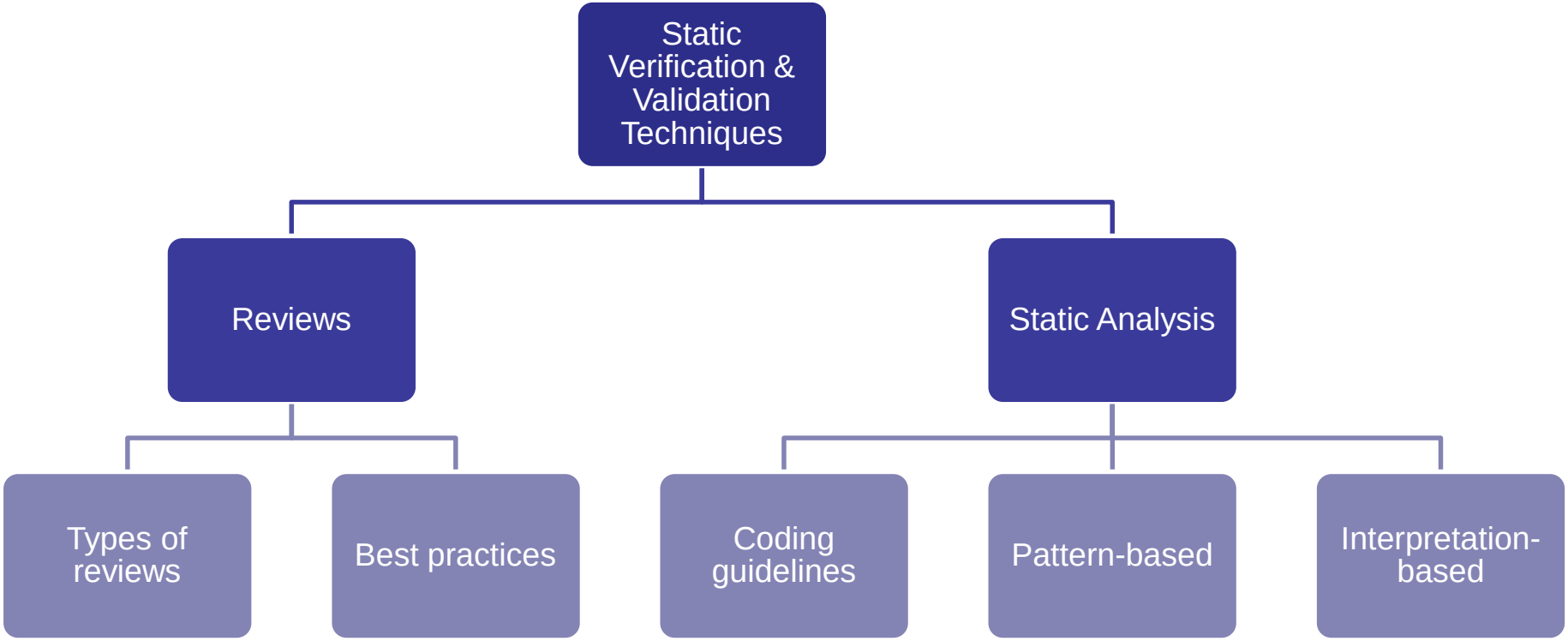
# Static Verification & Validation Techniques
## Software Analysis

**Daniel Varró ♦ McGill University**
***daniel.varro@mcgill.ca***

Other sources: I. Majzik, Z. Micskei

# Overview


Static Verification & Validation Techniques
Reviews
Static Analysis
Types of reviews
Best practices
Coding guidelines
Pattern-based
Interpretation-based

# Table of Contents

# CODING GUIDELINES

# Coding Guidelines

**Coding guidelines = set of rules giving recommendations on style (formatting, naming, structure) and best programming practices (constructs, architecture)**

- **Main categories:**
  - Industry/domain-specific (automotive, avionics, railway)
  - Platform-specific (Java, C, C++, ...)
  - Organization-specific (Google, CERN, etc.)

Src: I. Majzik, Z. Micskei: Systems and Software Verification course

# Industry-specific: MISRA C

- **MISRA: Motor Industry Software Reliability Association**
  - Focus on reliability, safety, security
  - Tools: PolySpace, SonarQube, Coverity, ...
- **Guidelines: 143 rules + 16 directives**
  - **Rules**:
    - Impose requirements on the source code which are complete, unambiguous and independent of
      any process, documentation or functional requirement
    - Analysis tools can check compliance with rules
      - Can be decidable (always yes/no) or undecidable
  - **Directives**:
    - Not defined wrt the source code alone
    - Impose requirements on processes, documentation or functional requirements
    - May be a degree of subjective judgement

# Sample Rules in MISRA C

**Unreachable code**: cannot execute thus it does not affect program behavior

- **Rules 2.x: Unused code**
  - 2.1: A project shall not contain unreachable code
  - 2.2: There shall be no dead code
- **Rules 5.x: Identifiers**
  - 5.5: Identifiers shall be distinct from macro names
  - 5.6: A typedef name shall be a unique identifier
- **Rules 13.x: Side effects**
  - 13.5: The right hand operand of a logical && or || operator shall not contain persistent side effects
- **Rules 15.x:**
  - 15.1: A goto statement shall not be used
  - 15.7: All if … else if constructs shall be terminated with an else statement

**Dead code**: If an operation is reachable but removing the operation does not affect program behavior

# Platform-specific: .Net

- **Framework Design Guidelines (C#)**
  - Focus on platform and API development
- **Categories:**
  - Naming, type design, member design, extensibility, exceptions, usage, common design patterns
  - „Do”, „Consider”, „Avoid”, „Do not”
- **Tool: StyleCop**

- **Examples**:
  - CONSIDER making base classes abstract even if they don’t contain any abstract members.
  - DO NOT provide abstractions unless they are tested by developing several concrete implementations and APIs consuming the abstractions.
  - DO choose carefully between an abstract class and an interface when designing an abstraction.
  - AVOID defining operator overloads, except in types that should feel like primitive (built-in) types.
  - DO provide at least one concrete type that inherits from each abstract class that you ship.

https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/index

# Organization-specific: Google

- **Java Style Guide:**
  - Source file basics
  - Source file structure
  - Formatting
  - Naming
  - Programming practices
  - Javadoc

- **Examples:**
  - Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are broken and they must be fixed. (vs. MISRA C)
  - Local variable names are written in *lowerCamelCase*
  - Braces are used with *if*, *else*, *for*, *do* and *while* statements, even when the body is empty or contains only a single statement.
  - Each statement is followed by a line break.

https://google.github.io/styleguide/javaguide.html

# How to enforce coding guidelines?

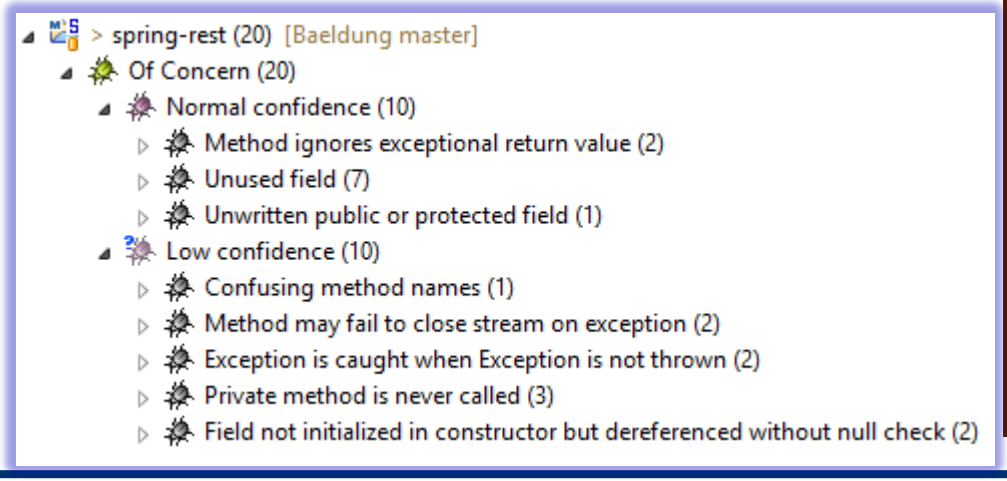
- **Tool support:**
  - Built-in base functionality in mandy IDEs
  - External tools
  - Part of continuous integration

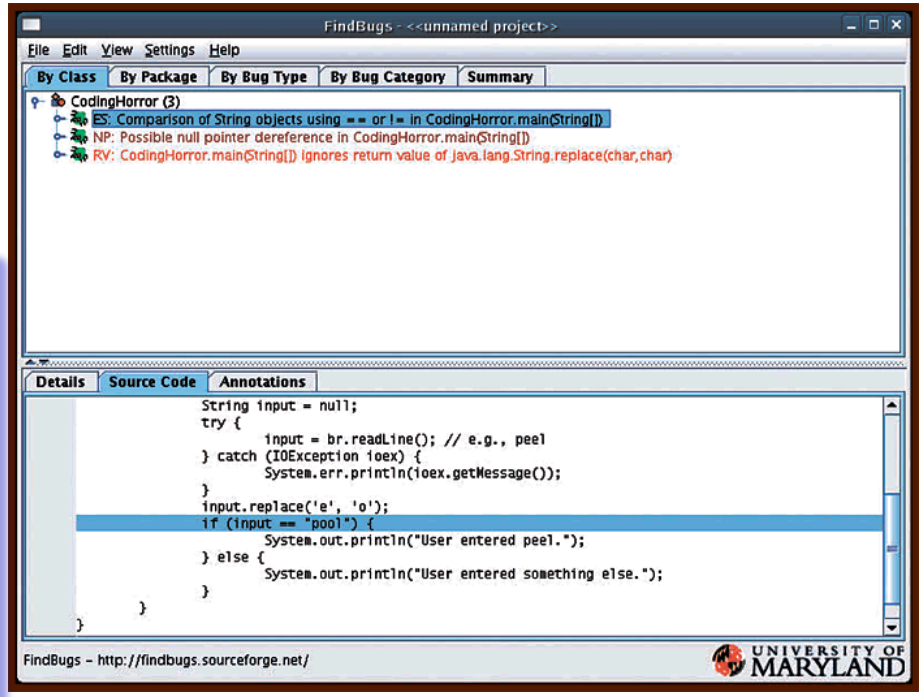
- **General advice:**
  - Always use a common guideline
  - Do not mix-and-match them as they may be contradictory
  - Minimum: use common formatting offered by IDE

# PATTERN-BASED STATIC ANALYSIS TOOLS

# Pattern-based static analysis

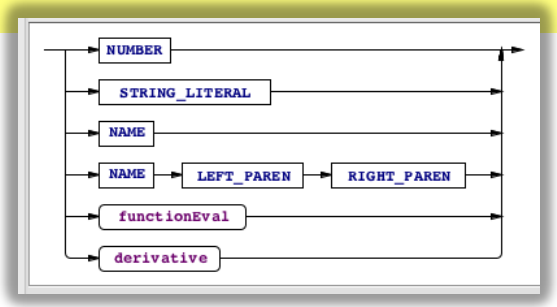
- **Automated analysis of software without code execution**
- **Scope**
  - Basic static properties with error patterns
  - E.g. Ignored return values, unused variable
- **Tools**
  - FindBugs, PMD, SonarQube, CheckStyle, ...



# Parsers: The Traditional Setup

Grammar
(EBNF)



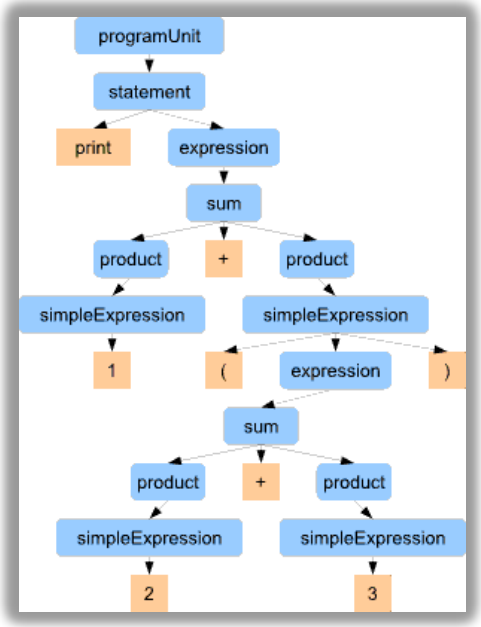


Source code of program

Parsing
(lexer + parser)

**Lexer**:
Source Code → Token List
with RegExp / Grammar

**Parser**:
Token List → AST
using Grammar

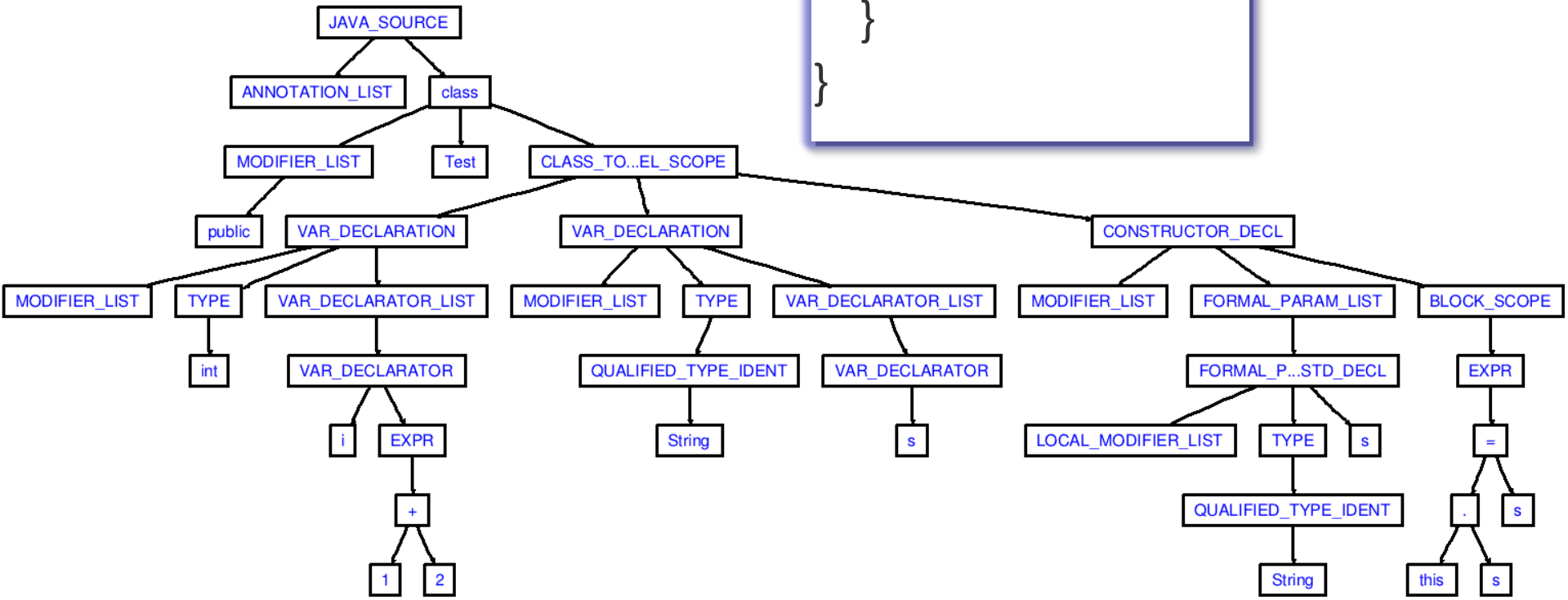



Abstract
syntax tree (AST)

Based on slides from Model Based Systems Engineering course at BME

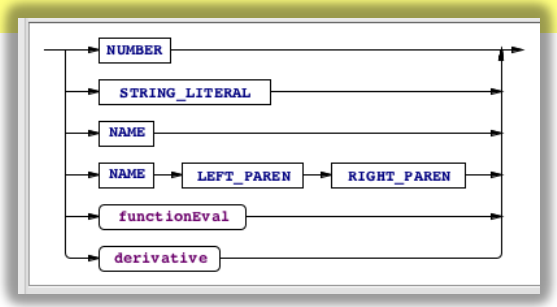
# A Java program and its AST

```
public class Test {
  int i = 1 + 2;
  String s;
  Test(String s) {
     this.s = s;
  }
}
```



# Parsers in Software Engineering Practice

Grammar

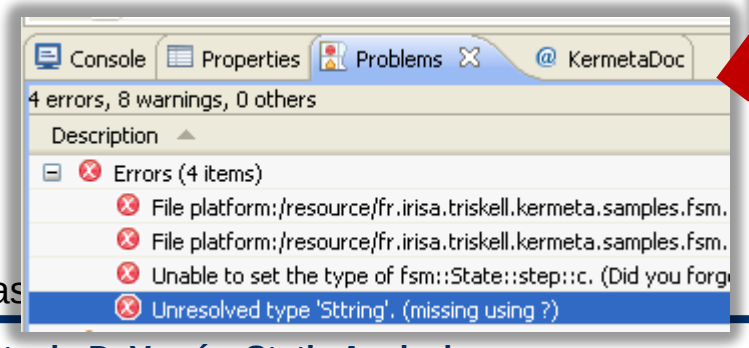


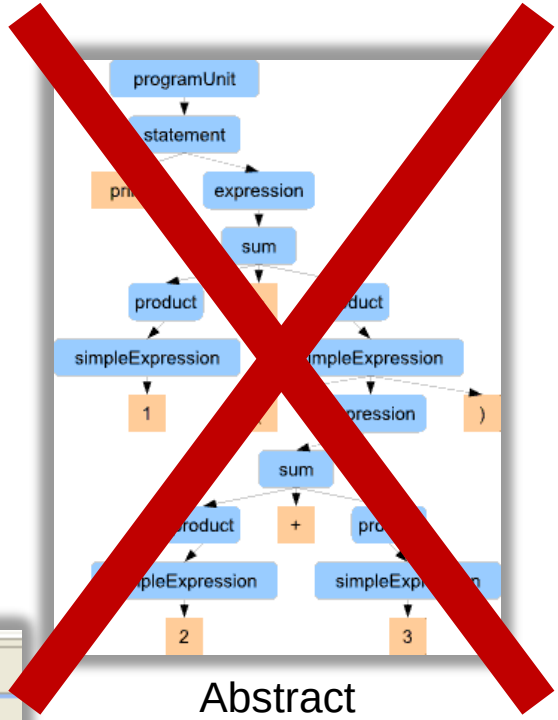


Source code of program

Parsing
(lexer + parser)

Error report




Abstract
syntax tree (AST)

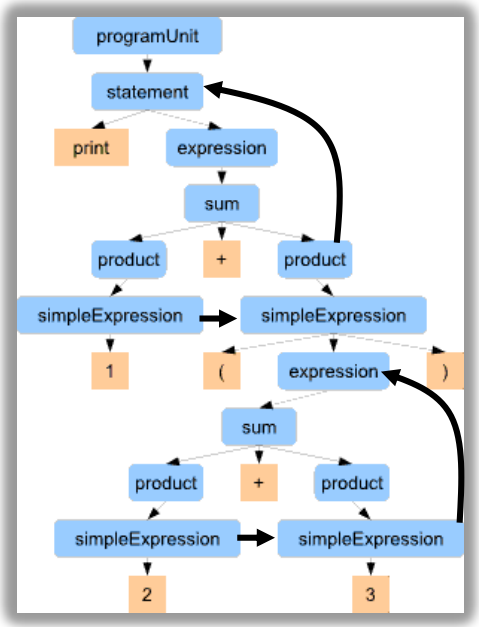
Error recovery parsing

Based on slides from Model Bas

# Views in Static Program Analysis

Source code of program

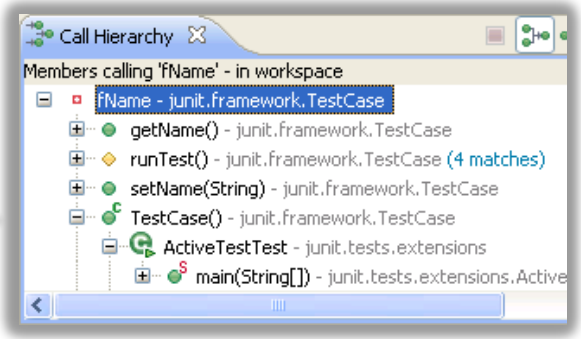
Parsing




AST: Abstract syntax tree

„Visitor"

Call graph



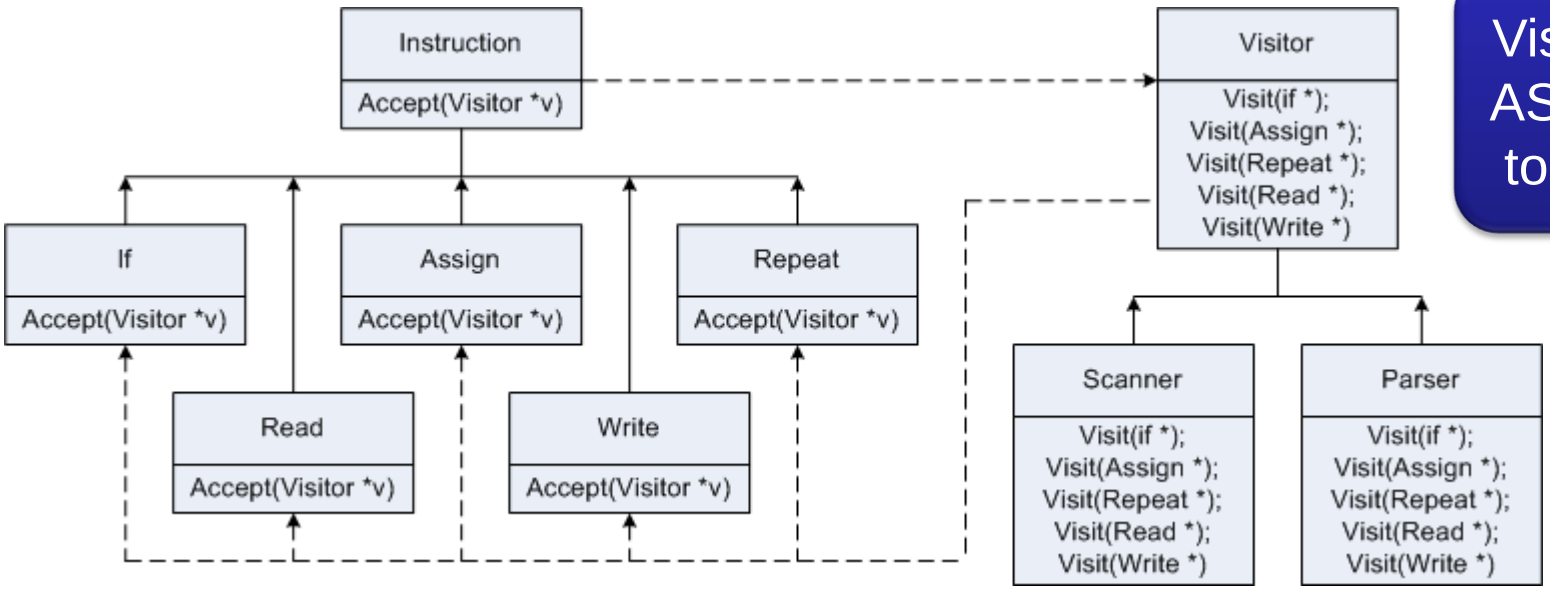

„Visitor"

Type hierarchy


Based on slides from Model Based Systems Engineering c

# Visitor Pattern in Static Analysis

- **Visitor pattern: Separate an algorithm from an object structure**
  - Each element class has an **"accept"** method that takes a visitor object as an argument
  - The visitor is an interface that has a different **"visit"** method for each element class
  - The "accept" method of an element class calls back the "visit" method for its class
  - Separate concrete visitor classes can perform some particular operations



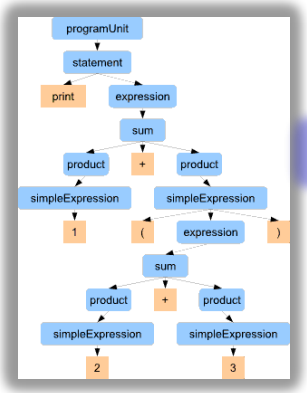

Visitors travers AST / ASG in a top-down way

http://alumni.cs.ucr.edu/~lgao/teaching/visitor.html

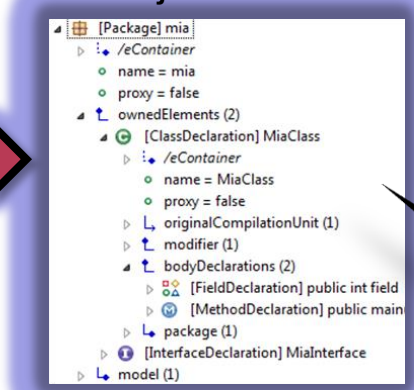
# ASG / DOM

Source code of program

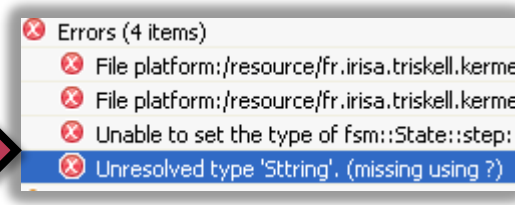
AST: Abstract syntax tree



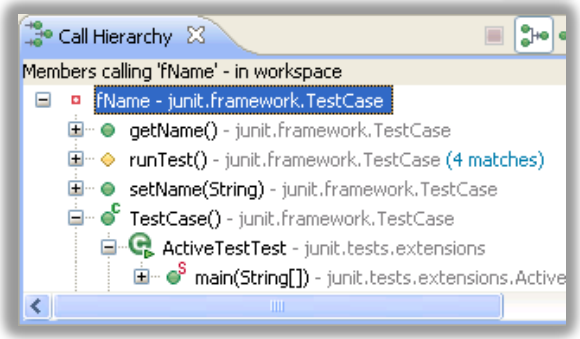

DOM: Document Object Model




Pattern-based static checks




Defined by a domain model




Call graph (View)


Type hierarchy (View)

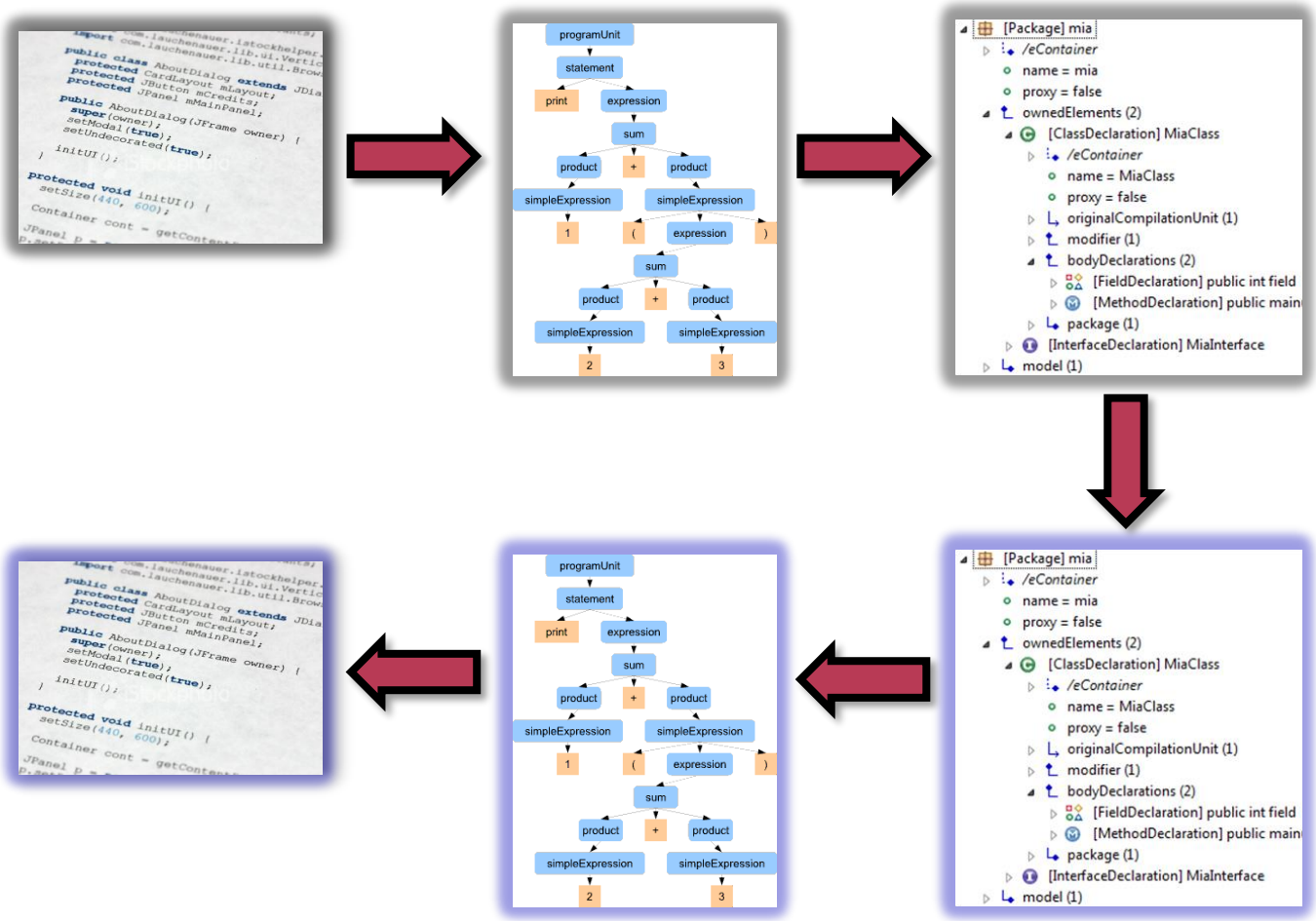
Based on slides from Model Based Systems Engineering course at BME

# Refactoring

Source code $_1$

AST $_1$

DOM$_1$

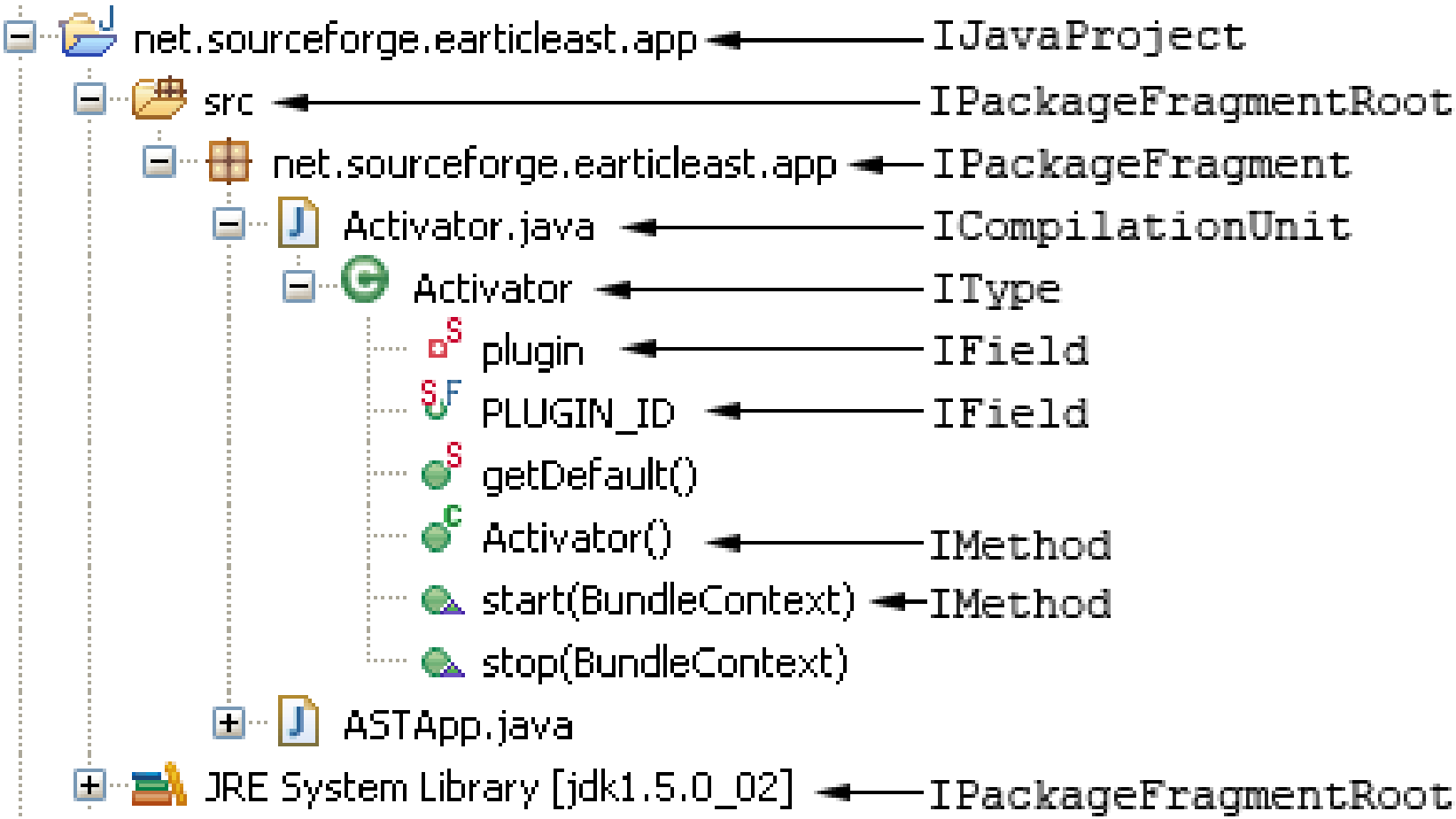



Source code $_2$

AST$_2$

DOM$_2$

# Back to DOM / ASG example




https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

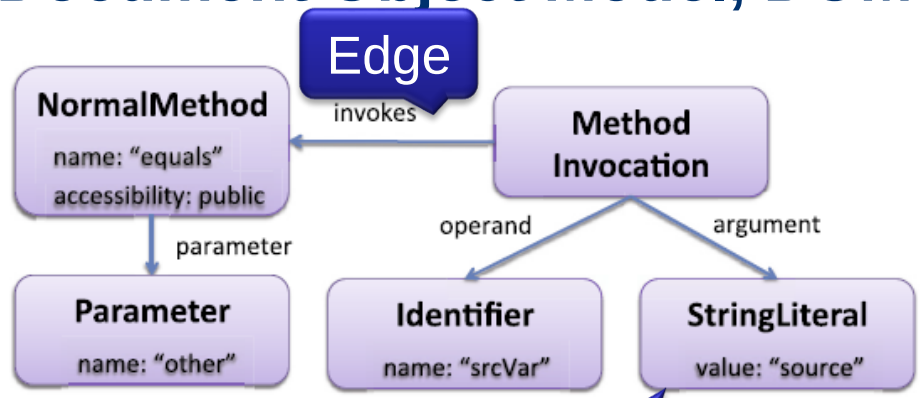
# Example: Abstract Syntax Graph

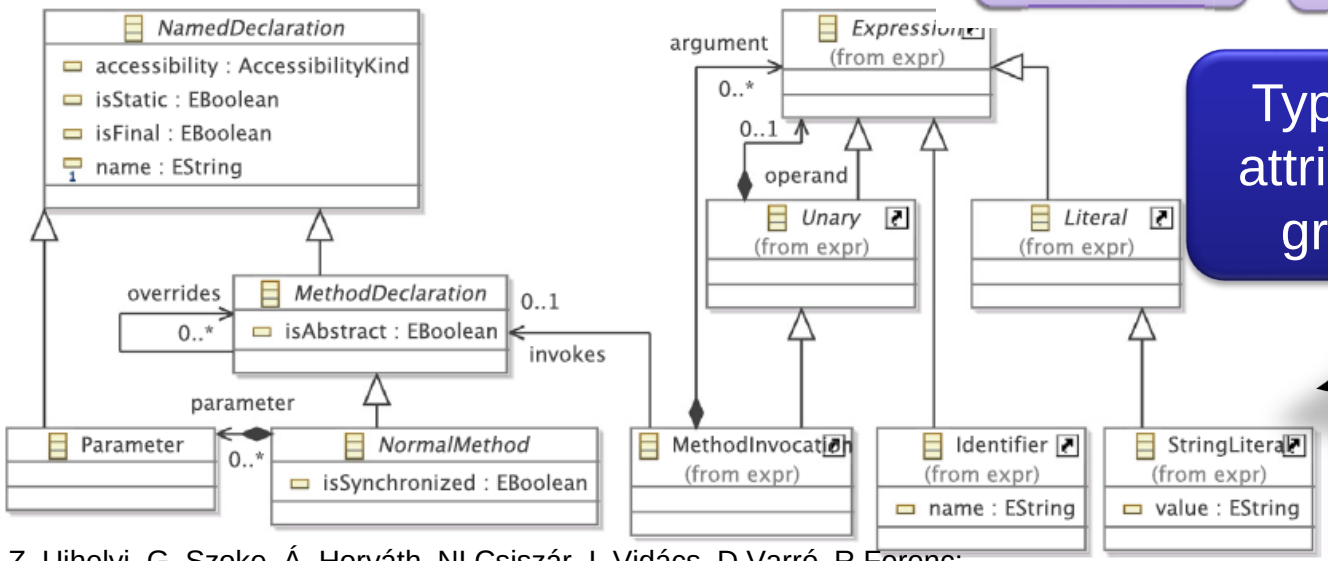
- **Java code snippet**

```
public boolean equals(Object other) {...}

...
// Code inside another method
// The variable 'srcVar' is defined locally
srcVar.equals("source");
...
```

- **Abstract Syntax Graph, ASG Document Object Model, DOM**




Typed & attributed graph

Domain model (metamodel) of Java ASG

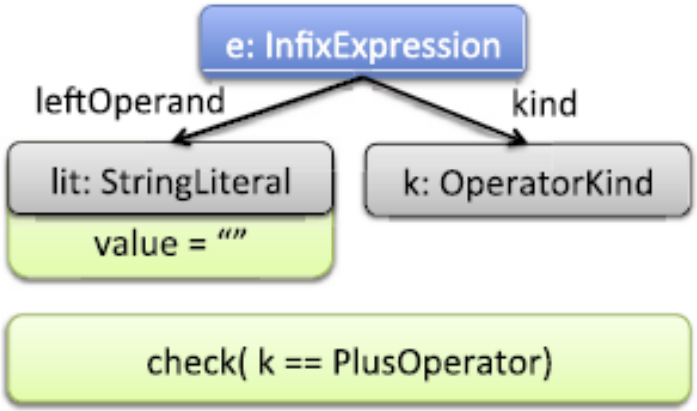
Z. Ujhelyi, G. Szoke, Á. Horváth, NI Csiszár, L Vidács, D Varró, R Ferenc: Performance comparison of query-based techniques for anti-pattern detection. Information & Software Technology 65: 147-165 (2015)

# Pattern-based Static Analysis

- **ASG: knowledge representation of program as typed and attributed graphs**
  - Nodes + Edges + Attributes
- **Find erroneous cases by graph pattern matching**
  - Find a small graph pattern in a large graph model
  - Match: complete mapping
    - graph pattern nodes → graph model nodes
    - graph pattern edges → graph model edges (compliant with node mapping)
  - No match → no violations

# Example: Graph patterns

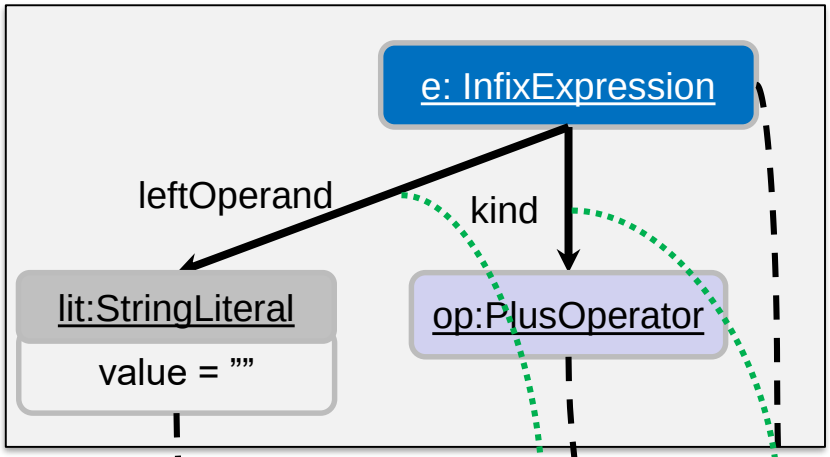
- **Concatenation to Empty String**
  - Nodes: `NodeType(var)`
    - e: InfixExpression
    - lit: StringLiteral
    - k: OperatorKind
  - Edges: `edgeType(from, to)`
    - leftOperand(e, lit)
    - kind(e, k)
  - Attribute checks:
    - lit.value == ""
    - k == PlusOperator




Z. Ujhelyi, G. Szoke, Á. Horváth, NI Csiszár, L Vidács, D Varró, R Ferenc:
Performance comparison of query-based techniques for anti-pattern detection. Information & Software Technology 65: 147-165 (2015)

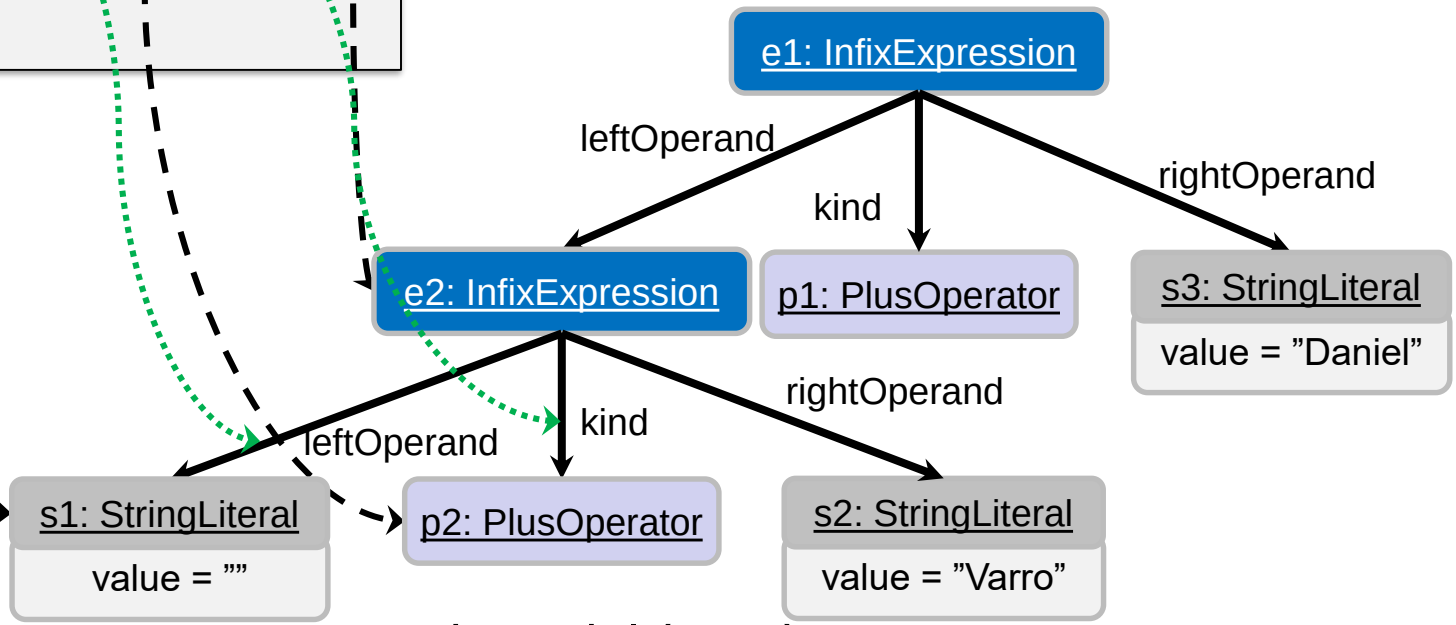
# Is there a match / violation? I

Graph Pattern




Match/Violation: YES!

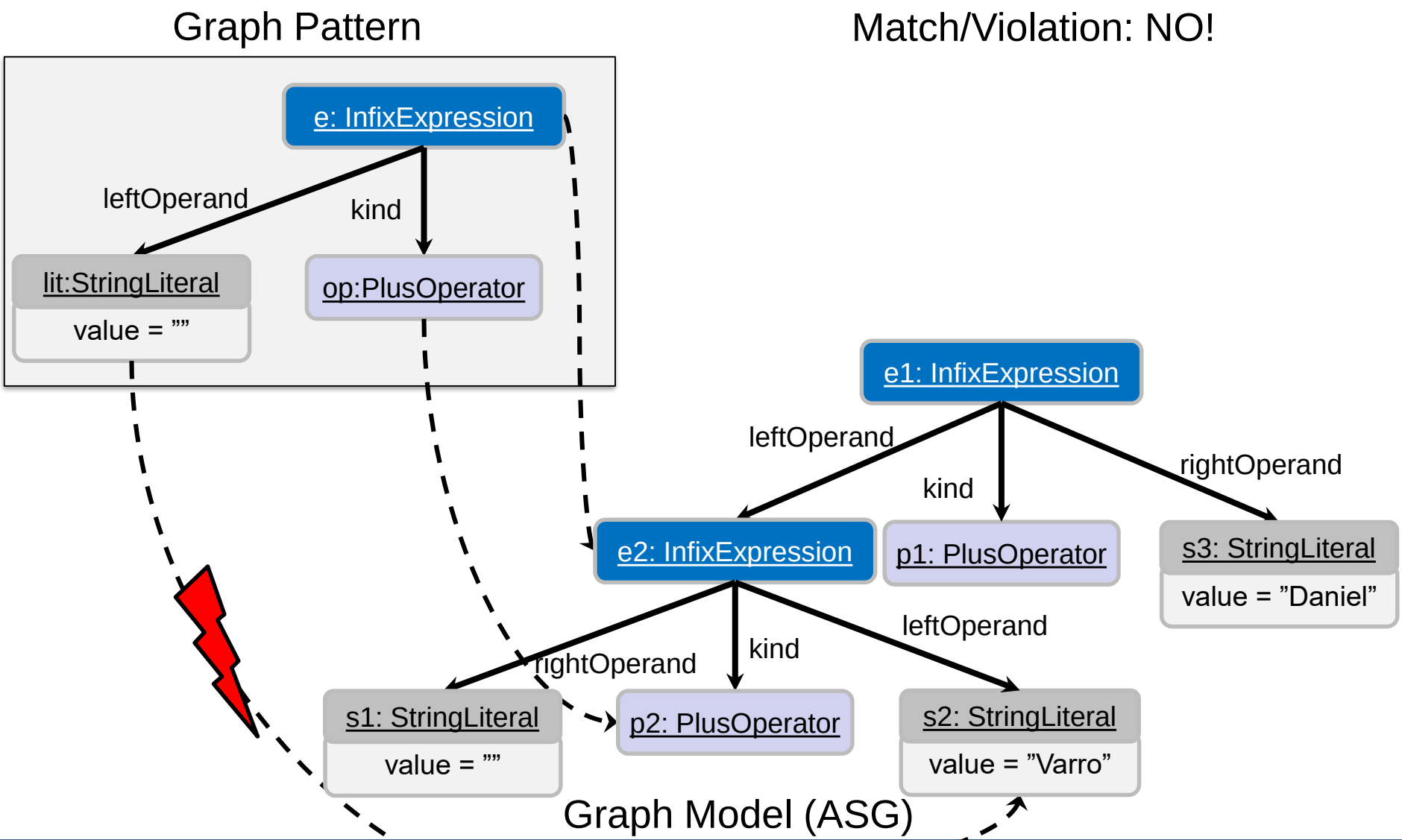
| Pattern | Model |
|---|---|
| e | e2 |
| op | p2 |
| lit | s1 |




Graph Model (ASG)

Pattern matching can traverse ASG in any order

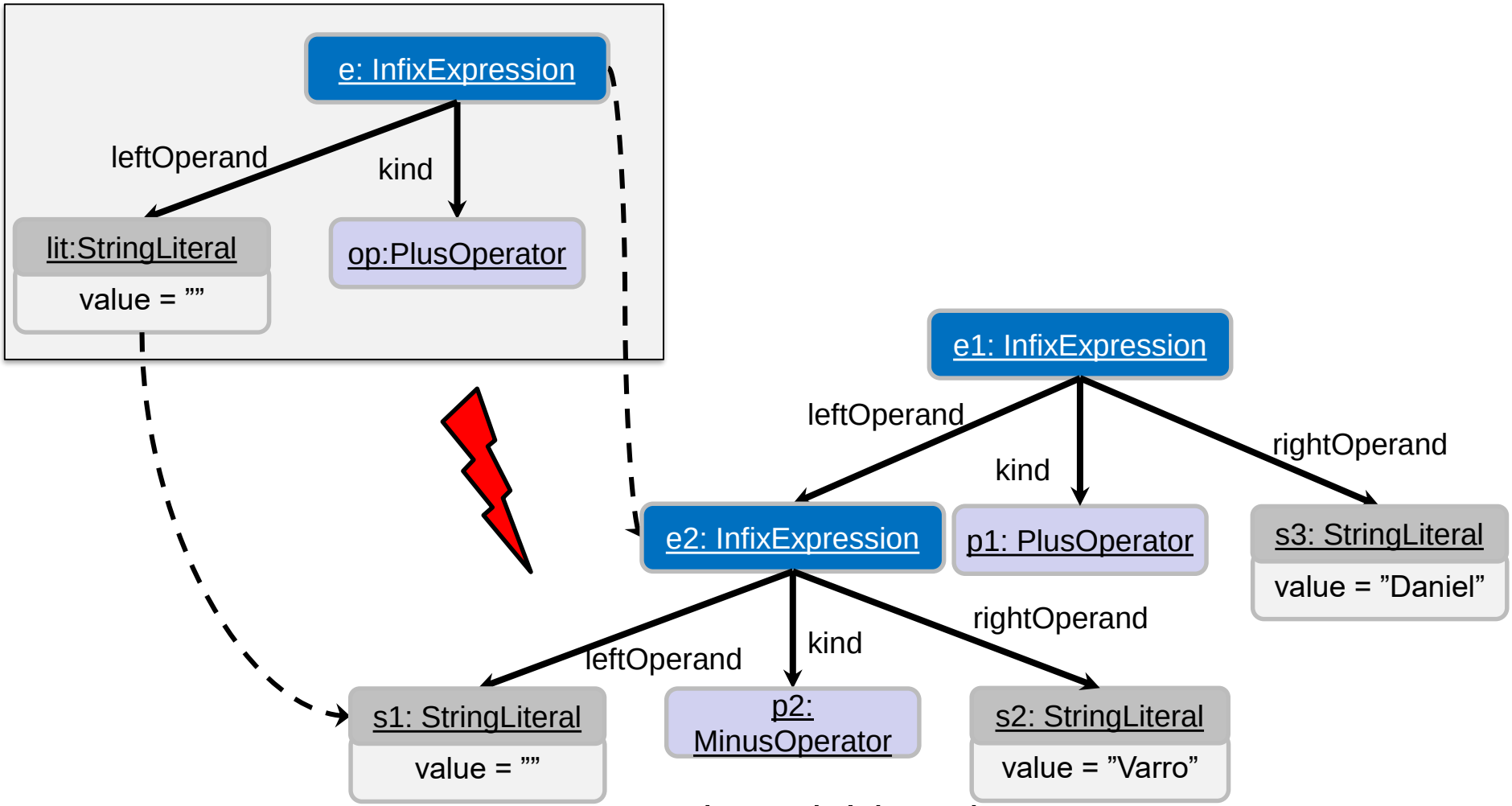
# Is there a match / violation? II



# Is there a match / violation? III

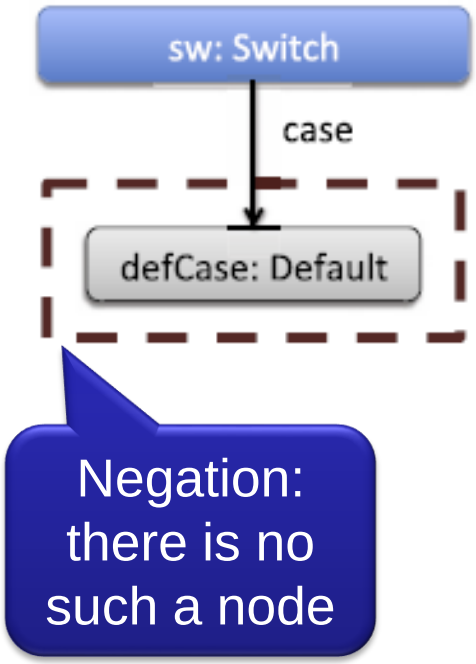
Graph Pattern

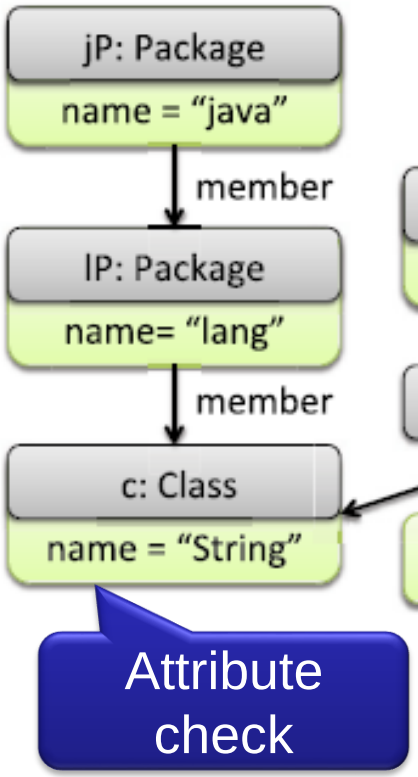
Match/Violation: NO!

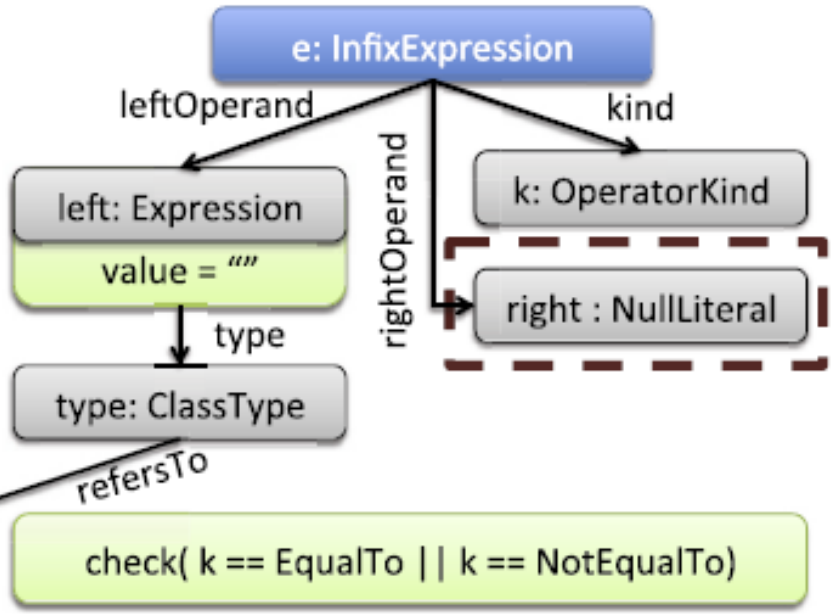



Graph Model (ASG)

# What do these patterns capture?



# Use of Lightweight Analyzers

- **Typically part of the continuous integration chain**
  - Automated their use
  - From the start of the project (Why?)
  - Perform check before/after each commit
  - Generate reports, send notifications (e.g. Slack)
- **Customize the tools**
  - Include/exclude issues of certain severity or category
  - Add cutstom rules
- **Review results carefully:**
  - **False positives** (false alarm)
    - Reported issue/defect would not cause a failure
  - **False negative**
    - Lack of errors does not mean correct software

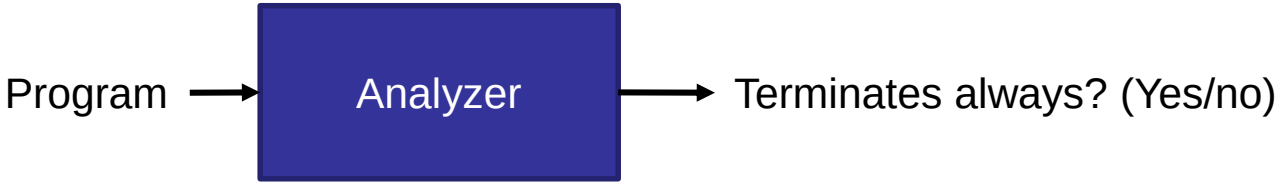
# ABSTRACT INTERPRETATION FOR STATIC ANALYSIS

# Testing vs. Static Analysis

- **Goal:**
  - Testing: investigates one run of the program
    - For a particular input + execution context
  - Static Analysis: reason about all runs of a program (without explicitly executing it)
    - For any input + any execution context
- **Outcome:**
  - Testing:
    - Pass / Fail
  - Static Analysis:
    - Safe / Error / Incomplete (Don't know)

# Complexity Issues

- **Can the analyzer prove that for any program P and input I, P will terminate or not? (Yes/No)**



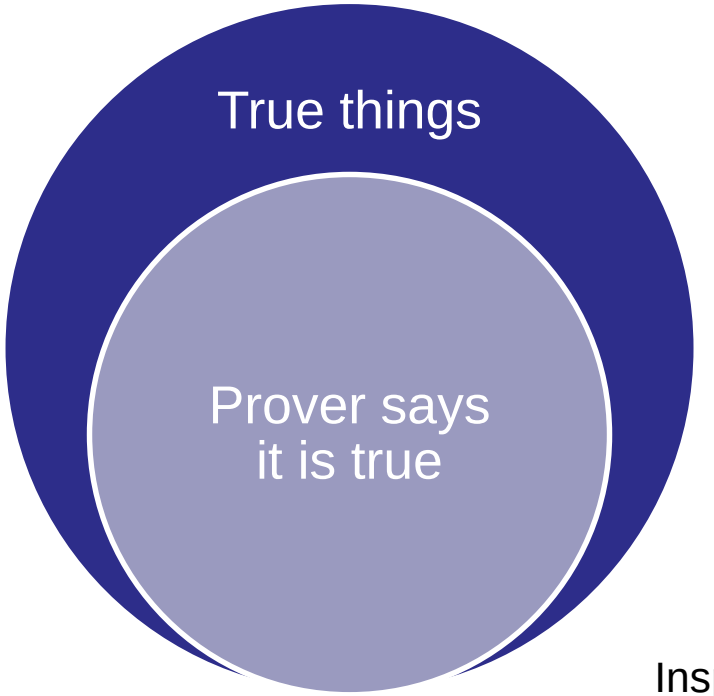

  - The Halting Problem (HP) ➔ undecidable
  - It is not possible to write an algorithm which provides a yes/no answer to all programs and all inputs
- **Most interesting propreties are undecidable:**
  - Deciding any of them would imply that we solve the HP
  - All array accesses are in bounds? (➔ buffer overruns)
  - Is a pointer dereferenced after it is freed?
  - Is an SQL query constructed from untrusted input?

Inspired by Univ. Maryland: Software Security course

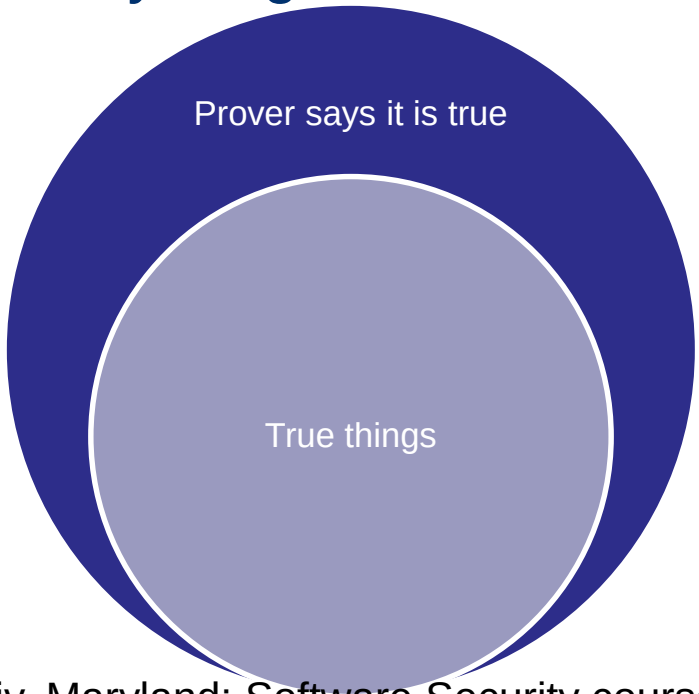
*Perfect* SA is impossible, but it *useful* SA is possible

# Soundness vs. Completeness

- **Soundness**
  - If prover says that P is true ➔ P is true
  - Trivially sound: SA says nothing




- **Completeness**
  - If P is true ➔ SA says that P is true
  - Trivially complete: say everything




Inspired by Univ. Maryland: Software Security course

# Sound + Complete SA

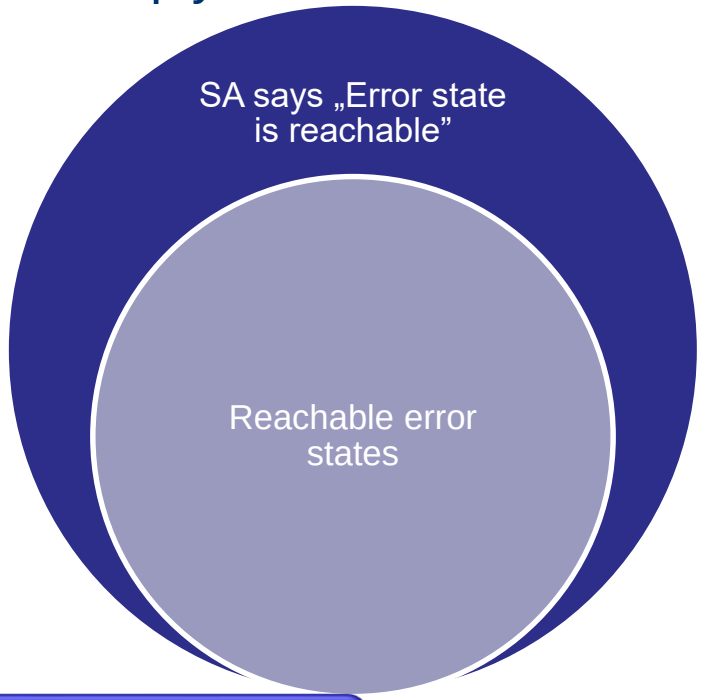
- **If SA says that P is true ⇔ P is true**
  - „Things I say are all the true things"
  - E.g. Systems that SA judges to be safe are all the safe systems

Such SA cannot exist for undecidable properties

SA proves all the true things and only true things

# Sound vs. Complete SA

- **Sound SA**
  - If SA says „program is error free" ➔ it is error really free
  - Alarms do not imply defects
- **Complete SA**
  - If program is claimed to be erroneous ➔ it is erroneous
  - Lack of error message does not imply error freedom


Useful SA are between soundness and completeness

# Designated properties of SAs

| | |
|---|---|
| **Precision:** | • Minimize the number of false alarms |
| **Scalable:** | • Capable of analyzing large programs |
| **Understandability:** | • Error reports should be interpreted by software engineers |

Intuition:

- A code that is hard to understand for humans is typically also hard to understand for programs (longer runtime OK)
- If programmers clean up code ➔ false alarms are reduced

# The Power of Abstraction

- **Example: Sign Analysis**
  - Arithmetic expressions:
    - Set of integers: $\mathbb{Z}$
    - Operators: +, ×
  - Goal make a judgement on the sign of an expression
    - S={ -, 0, +} : signs
    - Abstract integers: $\widehat{\mathbb{Z}} = \wp(S)$ (power set)
    - Operators:
  - **Abstraction function**: $\alpha$: $\mathbb{Z} \mapsto \widehat{\mathbb{Z}}$
    - $\alpha(z) = \{-\}\ if\ z < 0$
    - $\alpha(z) = \{0\}\ if\ z = 0$
    - $\alpha(z) = \{+\}\ if\ z > 0$
  - Note: $\alpha((-4) + 4) = 0$ but
    - $\alpha(-4) \oplus \alpha(4) = \{-, 0, +\}$

| a | b | a⊕b |
|---|---|---|
| { -,0} | { - } | { - } |
| { - } | { + } | { -,0,+ } |

| a | b | a⊗b |
|---|---|---|
| { +,0 } | { - } | { -, 0 } |
| { -,+ } | { 0 } | { 0 } |

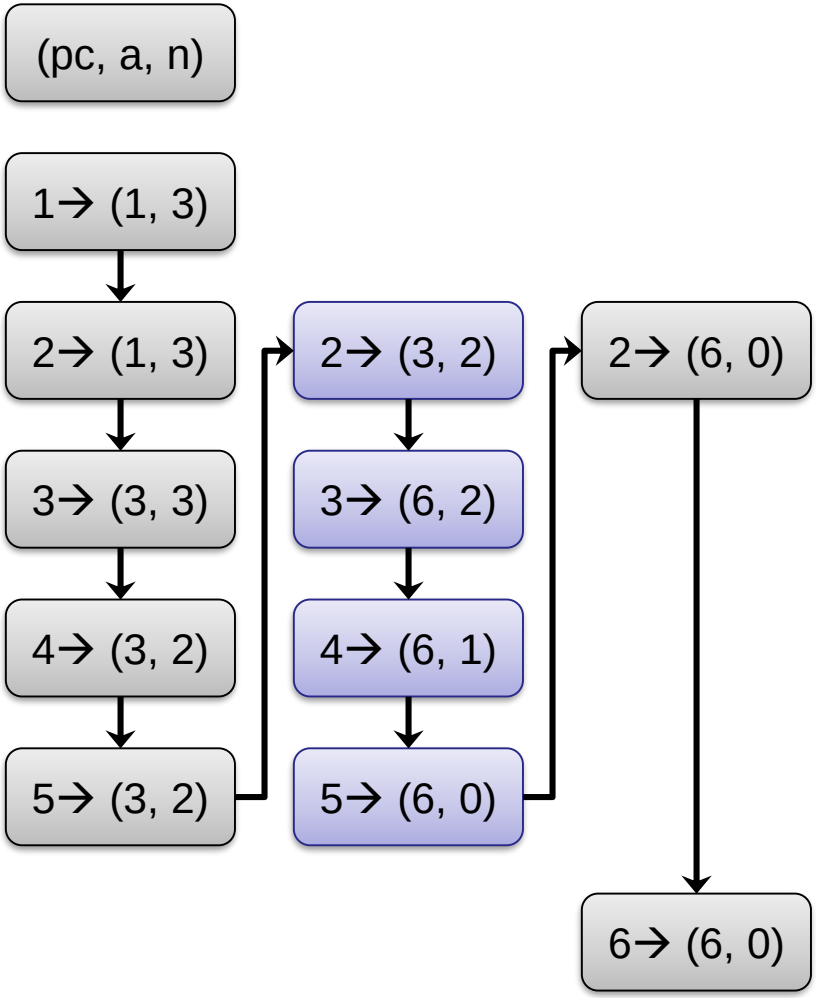
# Example: Factorial Program

```
<prog> ::= <stmt> ...
<stmt> ::= <label> :
  | goto <label> ;
        | <var> := <exp> ;
  | if <exp> goto <label> ;
<exp> ::= <exp> + <exp>
  | <exp> * <exp>
  | <exp> = <exp>
  | <int>
  | <var>
```

**Sample program** (factorial)

```
        a := 1 ;
top:    if n = 0 goto done ;
        a := a * n ;
        n := n + -1 ;
        goto top ;
done:
```

# Example: Concrete Evaluation

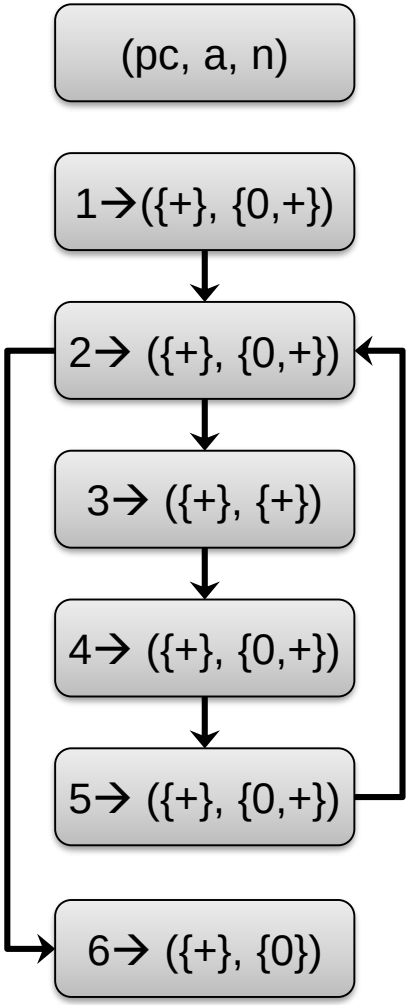



**Sample program** (factorial)

```
1:      a := 1 ;
2:      if n = 0 goto done ;
3:      a := a * n ;
4:      n := n + -1 ;
5:      goto 2 ;
6:
```

But will the program terminate for all other values of n as input?

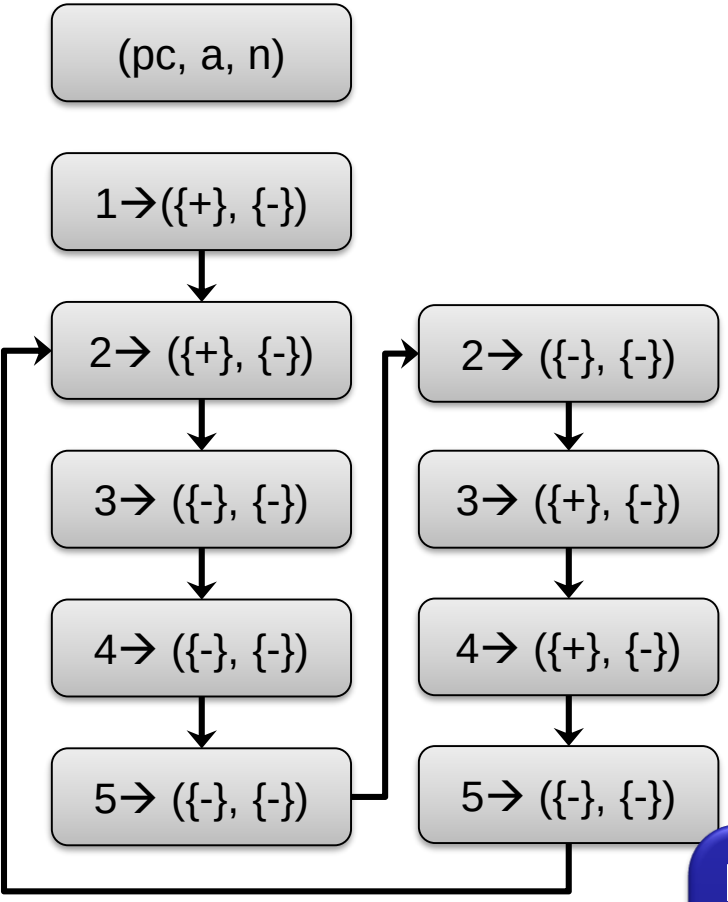
# Example: Abstract Evaluation I




**Sample program** (factorial)

```
1:      a := 1 ;
2:      if n = 0 goto done ;
3:      a := a * n ;
4:      n := n + -1 ;
5:      goto 2 ;
6:
```

# Example: Abstract Evaluation II




**Sample program** (factorial)

```
1:      a := 1 ;
2:      if n = 0 goto done ;
3:      a := a * n ;
4:      n := n + -1 ;
5:      goto 2 ;
6:
```

Many other abstractions used in practice:

- Intervals, pointers,
- Taint analysis (security)

Precise treatment by Galois connections

# Static Analysis: Pros vs Cons

**Pros**

- May achieve higher coverage then testing
- May prove absence of defects
- May find subtle programming flaws

**Cons**

- Properties are limited to functional correctness (no SA for performance)
- False alarms / Missed errors
- May be time consuming to run (or non-terminating)

# Examples: Infer

- **Static Analysis tool acquired by Facebook**
  - Main focus: mobile development
  - End users: Facebook, Instagram, Spotify, ...
- **Supported languages**
  - Android, Java:
    - null pointer exceptions, resource leaks, annotation reachability, missing lock guards, concurrency race conditions
  - C/C++/Objective-C:
    - null pointer dereferences, memory leaks, coding conventions and unavailable API

# Example: Infer

Infer

Docs Support Blog Twitter Facebook GitHub

```
infer@facebook:~/infer/examples$ vim Infer.java
infer@facebook:~/infer/examples$ infer -- javac Infer.java
Starting analysis (Infer version v0.5.0)
Computing dependencies... 100%
Analyzing 1 cluster. 100%
Analyzed 2 procedures in 1 file
No issues found
infer@facebook:~/infer/examples$ vim Infer.java
infer@facebook:~/infer/examples$ infer -- javac Infer.java
Starting analysis (Infer version v0.5.0)
Computing dependencies... 100%
Analyzing 1 cluster. 100%
Analyzed 3 procedures in 1 file

Found 1 issue
Infer.java:12: error: NULL_DEREFERENCE
  object s last assigned on line 11 could be null and is dereferenced at line 12
  10.       int mayCauseNPE() {
  11.           String s = mayReturnNull(0);
  12. >         return s.length();
  13.       }
  14.
infer@facebook:~/infer/examples$
```

Infer Java Tutorial

Root
- Hello.java
- Pointers.java
- Resources.java

Hello.java | Pointers.java

```
1  /**
2   * The Infer "Hello World" Java examp
3   *
4   * Click the "Analyze" button to run
5   * Learn more about Infer at http://f
6   *
7   */
8
9  import java.io.File;
10 import java.io.FileInputStream;
11 import java.io.FileOutputStream;
12 import java.io.IOException;
13 import java.io.OutputStream;
14
```

This will display the output.

Src: http://fbinfer.com/

# Example: PolySpace

- **Static Analysis tool acquired by MathWorks**
  - Code prover:
    - Formal methods based static analysis to verify program execution at language level
    - Checks normal + abnormal usage conditions
    - Can prove absence of defects: overflow, division by zero, index out of bounds
    - Colors the code (next slide)
  - Bug finder
    - Perform static analysis on source code
      - Concurrency, runtime errors, vulnerabilities
      - Coding guidelines, code-level metrics

# PolySpace example

**Green: reliable**
safe pointer access

**Red: faulty**
out of bounds error

**Gray: dead**
unreachable code

**Orange: unproven**
may be unsafe for some conditions

**Purple: violation**
MISRA-C/C++ or JSF++ code rules

***Range data***
*tool tip*

```
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
      *p = 0;
      p++;
    }

  if (get_bus_status() > 0) {
    if (get_oil_pressure() > 0) {
      *p = 5;
    } else {
      i++;
    }
  }

  i = get_bus_status();

  if (i >= 0) {
    *(p - i) = 10;
  }
}
```

variable 'I' (int32): [0 .. 99]
assignment of 'I' (int32): [1 .. 100]

https://www.mathworks.com/products/polyspace-code-prover/features.html