Verifying source code

Marussy Kristóf, Levente Bajczi, Ákos Hajdu,
Zoltán Micskei, István Majzik

MŰEGYETEM 1782

BME Faculty of Electrical Engineering and Informatics

ftsrg Critical Systems Research Group

Main topics of the lecture

Program Verification Overview

- Verification Goals
- Analysis Properties
- Analysis Methods

Model Checking with SMT

- SMT Problem
- Technique
- Advantages
- Disadvantages

Practical Software Verification

- Challenges
- Tools

Example

```
k = ioread32();
i = 2;
j = k + 5;
while (i < 10) {
    i = i + 1;
    j = j + 3;
}
k = k / (i - j);
```

Division by zero?
For what *k*?

```
Verification FAILED.
  Divison by zero in example.c at line 14 column 11.
Error trace:
------------
#0 in function main():
  call ioread32() returned -19   at 6:13
  k := -19        at 6:13
  i := 2
  j := -14        at 8:15

  ...

  i := 10         at 10:15
  j := 10         at 11:15
  j := 10
  i := 10
```

Typical verification goals

- Improper resource management
  - Resource leaks: memory, files, sockets
- Illegal operations
  - Division by zero, overflow, index out-of-bounds, null pointer, **assertions**
- Dead code and data
  - Code and data not reached or used
- Incomplete code
  - Uninitialized variables, unspecified return values, missing cases in switch
- Other
  - Non-termination, uncaught exceptions, race conditions

+ security

dl.acm.org/citation.cfm?id=1390956

Analysis Properties

		Analysis	
		Correct	Faulty
Source code	Correct	Proven correct	False alarm
	Faulty	Missed bug	Found bug

Analysis properties

- Checking most of the run time properties is computationally complex or undecidable (see *Theory of Algorithms* course)
- Approximations: sacrifice precision to save analysis time
  - Flow sensitive < Path sensitive < Interprocedural
- Precision
  - False positive (false alarm): report an error that does not cause a real problem
  - False negative (missed bug): an actual problem does not get reported

		Analysis	
		Correct	Faulty
Source code	Correct	Proven correct	False alarm
	Faulty	Missed bug	Found bug

Flow sensitive

- Execution order of program statements is considered

- Example
  - Flow insensitive: x and y might point to same location
  - Flow sensitive: x and y might point to same location after line 6

```
1: int* x = malloc(...)
2: int* y = malloc(...)
3: *x = ...
4: *y = ...
5: f(*x, *y)
6: x = g()
7: y = g()
```

Path sensitive

- Distinguish between different execution paths
- Consider only feasible paths

- Example
  - Path insensitive: division by 0 possible at line 5
  - Path sensitive: division by 0 is not possible

```
1: f(int a, int b) {
2:     if (a == b) {
3:         return 0
4:     } else {
5:         return 1 / (a - b)
6:     }
7: }
```

Interprocedural

- Analyze method body at each call site
  - Otherwise it is called *intra*-procedural

- Example
  - Intraprocedural: create might return null pointer
  - Interprocedural: create never returns null, x can be dereferenced

```
int* create(int i) {
    int* p = malloc(...)
    if (p == NULL) exit(1)
    *p = i
    return p
}

void main() {
    int* x = create(5)
    print(*x)
}
```

Analysis Methods

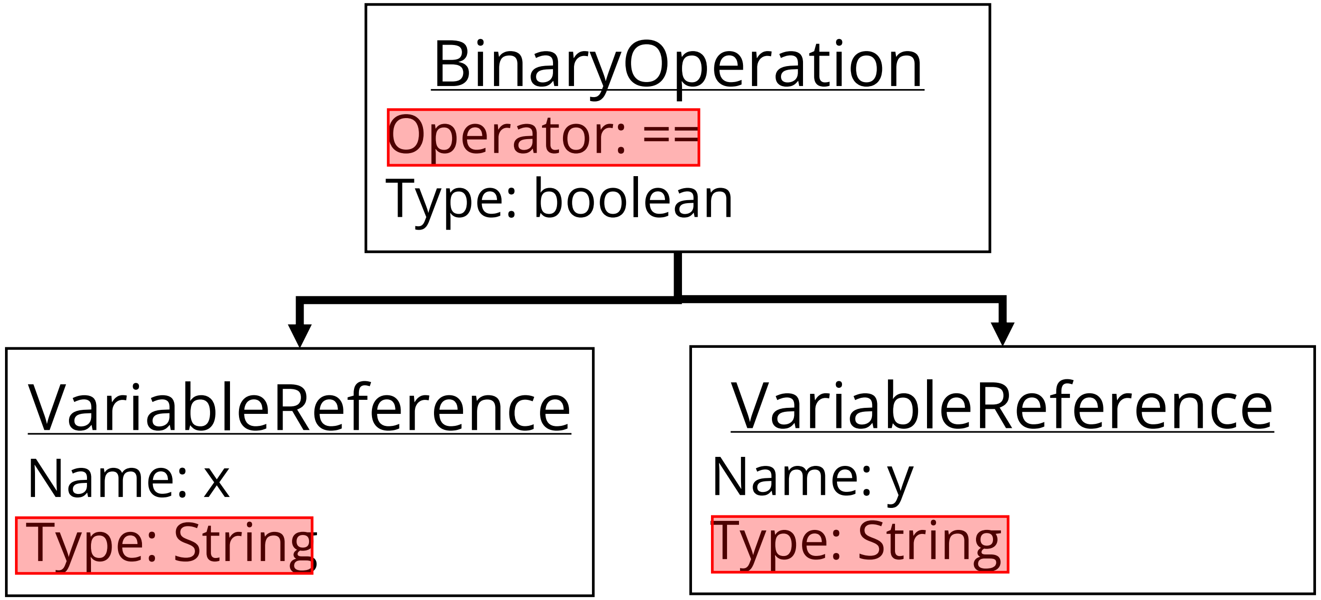
- Pattern-based
- Model checking
- Theorem proving

Pattern-based

- Syntactic pattern matching
  - Usually based on the AST (Abstract Syntax Tree)

```
f(String x, String y) {
    if (x == y) {
        …
    }
}
```

Use equals instead of == (Java)



Pattern-based (properties)

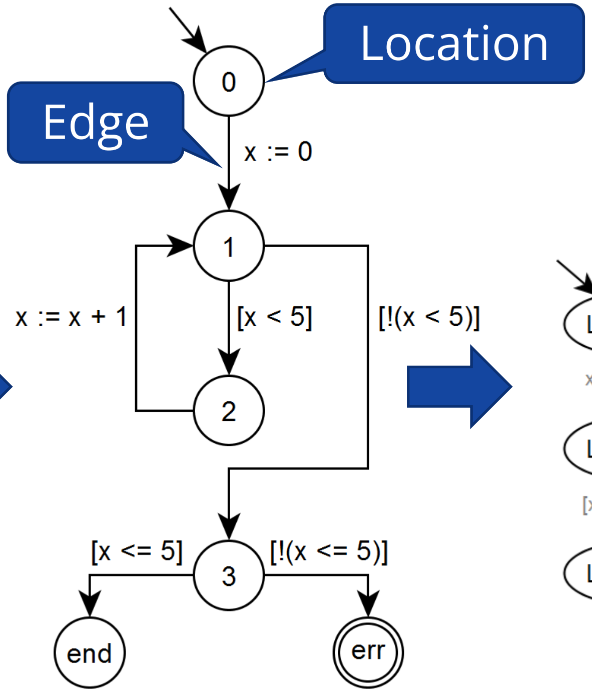
- Typically path and context insensitive
- Can analyze very large projects
- Fully automated
- Simple properties (patterns)
- Produce many false positives and false negatives

Model checking

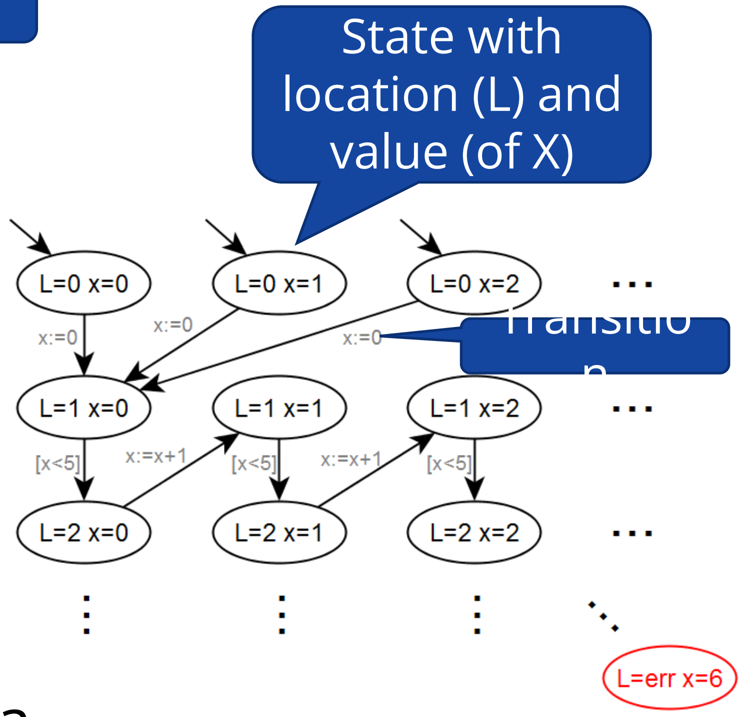
- Describe program and property in a formal language
- Enumerate all states and transitions

```
   x : int
0: x = 0
1: while (x < 5) {
2:     x = x + 1
   }
3: assert (x <= 5)
```

Program



Control Flow Automata (formal representation)



States and transitions

Model checking (properties)

- Flow and path sensitive
- Can be context sensitive and interprocedural
- Usually automated, but might require user annotations
- Fully formal: no false alarms, no missed bugs (w.r.t. property)
- Limited scalability
  - Symbolic techniques, bounded model checking, abstraction

Theorem proving

- Describe the program and property with logical formulas
- Apply inference rules to prove that the program satisfies the property

```
int abs(int x) {
    if (x < 0) {
        r = -x
    } else {
        r = x
    }
    return r
}
```

r >= 0 ?

$$(x < 0) \land (r = -x) \lor (x \geq 0) \land (r = x) \Rightarrow r \geq 0$$

Theorem proving (properties)

- Fully formal: no false alarms, no missed bugs (w.r.t. property)
- Manual user assistance is usually required
- Formal methods expertise required

Analysis methods: Summary

- General experience, but exceptions are possible
- Combined methods are possible

Method	Flow sensitive	Path sensitive	Context sensitive	Interprocedural	Arbitrary property	Automated	No false alarms	No missed bugs	Large codebase
Pattern matching	X	X	X	X	X	✓	X	X	✓
Model checking	✓	✓	(✓)	(✓)	✓	(✓)	✓	✓	(X)
Theorem proving	✓	✓	(✓)	(✓)	✓	(X)	✓	✓	X

Satisfiability Modulo Theories

Z3

CVC5

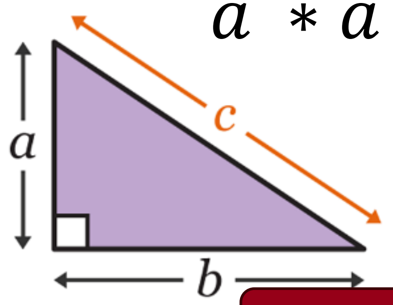
MathSAT 5

An SMT Solver for Formal Verification & More

SMT – Satisfiability Modulo Theories

- **Satisfiability**
  - *Given a logical formula, how to assign **values** to **symbols** to make it true?*

$$a * a + b * b = c * c$$



SAT

a	b	c
0	0	0
3	4	5
…	…	…

$$a < b \text{ and } b > a$$

UNSAT

- **Modulo Theories**
  - *What is the **semantics** of **operations**?*
  - *Overflow (UB)? Wraparound? Infinite integers?*

$$a = 127 + 1$$

Model Checking with SMT

```
k = ioread32();
i = 2;
j = k + 5;
i = i + 1;
j = j + 3;
k = k/(i - j);
```

input

Precise, can create test case

Complex, specialized SW needed

```
k⁰ = ioread32();
i⁰ = 2;
j⁰ = k⁰ + 5;
i¹ = i⁰ + 1;
j¹ = j⁰ + 3;
assert(i¹–j¹!=0);
k¹ = k⁰/(i¹ – j¹);
```

Static-Single-Assignment Form

```
Input: free variable
i0 = 2
j0 = k0 + 5
i1 = i0 + 1
j1 = j0 + 3
i1-j1 == 0 (negated!)
k1 = k0 / (i1 – j1)
```

SMT

SAT:	unsafe
UNSAT:	safe

Practical Software Verification

Challenges

- Language standards: rarely formal!
  - **Undefined behavior**: *anything* can happen – we cannot verify!
    - *Example in C:* data races, signed overflow (before C23)
      If anything can happen, why not assume the program is unsafe?
  - **Unspecified behavior**: *several things* can happen – who decides?
    - *Example in C:* when you call `f(g(), h())`, which of *`g()`* and *`h()`* is executed first?
      CLANG and GCC disagree! Which to choose?
- Floating points, arrays, structs, linked lists, trees, ...
  - Hard to transform into SMT
  - Floating points: differing standards (e.g.: C floats and Java floats differ!)

Pattern-based

- FindBugs (SpotBugs)
  - spotbugs.github.io
  - Java
  - 400+ bug patterns in different categories

- Error Prone (by Google)
  - errorprone.info
  - Java
  - 300+ bug patterns in different categories

Smart pattern-based (using additional techniques)

- Sonar{lint/cloud/qube}
  - sonarsource.com
  - 20+ languages, including Java, C#, C, C++
  - Code quality management platform
- PMD
  - pmd.github.io
  - Java, JavaScript, PLSQL
- Coverity
  - synopsys.com
  - Many languages, including Java, C, C++, C#, Python, JS

Model checker

- CPAchecker
  - Configurable software-verification platform
  - Input: C program + specification
    - Assertion, error label, deadlock, null dereference, ...
  - Highly configurable
    - Different kinds of abstractions
    - Various algorithms and strategies
- Theta
  - Generic, configurable and modular verification framework
  - Gazer: LLVM-based frontend for C programs
  - Abstraction-based algorithms for state reachability

github.com/FTSRG/theta
cpachecker.sosy-lab.org

Model checker

- SLAM
  - Part of Static Driver Verifier Research Platform (SDVRP)
  - Structure
    - Driver C code: analyzed components
    - Platform model: describe environment
    - Analysis: adherence to API usage rules
  - Applies abstraction and symbolic model checking

microsoft.com/en-us/research/project/slam

Theorem prover

- PVS
  - Specification language
  - Pre-defined theories
  - Integrated type checker, theorem prover and symbolic model checker
  - Utilities, e.g., code generator, random tester

SRI International

```
sum: THEORY
  BEGIN

   n: VAR nat
   f, g: VAR [nat -> nat]

   sum(f, n): RECURSIVE nat =
  (IF n = 0
      THEN 0
      ELSE f(n-1) + sum(f, n - 1)
   ENDIF)
  MEASURE n
```

```
sum_plus: LEMMA
   sum((lambda n: f(n) + g(n)), n)
  = sum(f, n) + sum(g, n)

 square(n): nat = n*n

 sum_of_squares: LEMMA
   6 * sum(square, n+1) = n * (n + 1) * (2*n + 1)

 cube(n): nat = n*n*n

 sum_of_cubes: LEMMA
   4 * sum(cube, n+1) = n*n*(n+1)*(n+1)

END sum
```

pvs.csl.sri.com

Using static verification in practice

- J. Carmack. In-Depth: Static Code Analysis
  - “**it is irresponsible to not use it**”
  - “there was an epic multi-programmer, multi-day bug hunt that wound up being traced to something that `/analyze` had flagged, but I hadn't fixed yet.”

- A few Billion Lines of code Later using static Analysis to find Bugs in the Real World
  - Turning a prototype into commercial tool
  - “False positives do matter. In our experience, more than 30% easily cause problems. People ignore the tool. True bugs get lost in the false.”

coverity®

Using verification in practice

- [Lessons from Building Static Analysis Tools at Google]
  - "Static analysis authors should focus on developer feedback"
  - "Crowdsourcing analysis development"
- [Facebook: Moving Fast with Software Verification]
  - Developers should see analysis results as part of their normal workflow, no context switch should be required
- [Amazon: Code Level Model-Checking in the Software Development Workflow]
  - Experience report on applying model checking at AWS
  - "AWS developers are increasingly writing their own formal specifications"