E0:227, Program Analysis and Verification

3:1, January - April 2009

E-Classroom, CSA, M-W 11:30am-1pm

http://www.csa.iisc.ernet.in/~raghavan/pav09/index.html

K. V. Raghavan and Deepak D'Souza

Software development is hard

Average software-development project [Barry Boehm, ICSE '06 keynote] incurs:

- 90% cost overrun
- 121% time overrun
- delivers only 61% of initially promised functionality

Software development lifecycle

For each release of the software:

- Requirements
- Analysis and Design
- Coding
- Testing
- Production/deployment; feedback from users

Software development lifecycle

For each release of the software:

- Requirements
- Analysis and Design
- Coding
- Testing
- Production/deployment; feedback from users

Testing, finding and fixing bugs (i.e., Quality Assurance) consumes 50% of total cost and time of software development.

Software development lifecycle

For each release of the software:

Requirements

Analysis and Design

Coding

Testing

Production/deployment; feedback from users

Testing, finding and fixing bugs (i.e., Quality Assurance) consumes 50% of total cost and time of software development.

The problem gets worse after multiple releases, because:

- People lose knowledge of the code
- Code becomes bigger, more complex, and poorer structured

Why quality assurance takes so much effort

- Defects are common
- Are hard to find
  - often, get identified only after release
  - no good tools, and people don't use ones that are there
- When a program crashes, or gives wrong answer, hard to detect the root defect
- Defects in requirements or design should be found before coding starts, else code will manifest it
  - however, no widely used formal techniques for these
- Incorrect understanding of customers requirements

Kinds of software defects

- Crashes
  - Null pointers, uninitialized values
  - Array index out of bounds, buffer overrun
  - Memory leaks
  - Misuse of pointers and buffers (in languages like C)
  - Unreachable code
- Does not interact with other software in the same way as the previous version.
- Leaks information to unauthorized channels
- Performs poorly
- Logical errors (design-time errors)

What's wrong with this program?

```
int middle(int x,
           int y,
           int z) {
  int m = z;
  if (y < z)
    if (x < y)
      m = y;
    else if (x < z)
      m = x;
  else
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  return m;
}
```

What's wrong with this program?

```
int middle(int x,
           int y,
           int z) {
  int m = z;
  if (y < z)
    if (x < y)
      m = y;
    else if (x < z)
      m = x;
  else
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  return m;
}
```

⇒

```
int middle(int x,
           int y,
           int z) {
  int m = z;
  if (y < z)
    if (x < y)
      m = y;
    else if (x < z)
      m = x;
    else
      if (x > y)
        m = y;
      else if (x > z)
        m = x;
  return m;
}
```

Tool BLAST identifies the two lines before `return` as unreachable

A common approach to software validation: Testing

- A test suite (set of test cases) is created, and executed for each version.
- **Black box** testing: Test cases are created manually by user, or generated randomly.
- **White box** testing: Test cases are generated by an analysis of the program code to increase code coverage.
    - Typically needs tool support.
- What's good about testing? *All bugs found are real bugs.*
- What's bad about testing?

A common approach to software validation: Testing

- A test suite (set of test cases) is created, and executed for each version.
- **Black box** testing: Test cases are created manually by user, or generated randomly.
- **White box** testing: Test cases are generated by an analysis of the program code to increase code coverage.
    - Typically needs tool support.
- What's good about testing? *All bugs found are real bugs.*
- What's bad about testing?
    - 100% coverage of the program's behavior is impossible.
    - Therefore, cannot find all bugs or prove the absence of bugs.
- Very hard to test the portion inside the "if" statement!

```
input x
if (hash(x) == 10) {
   ...
}
```

Program verification

The algorithmic discovery of properties of a program by inspection of the source text.

– *Manna and Pneuli, "Algorithmic Verification"*

Program verification

The algorithmic discovery of properties of a program by inspection of the source text.

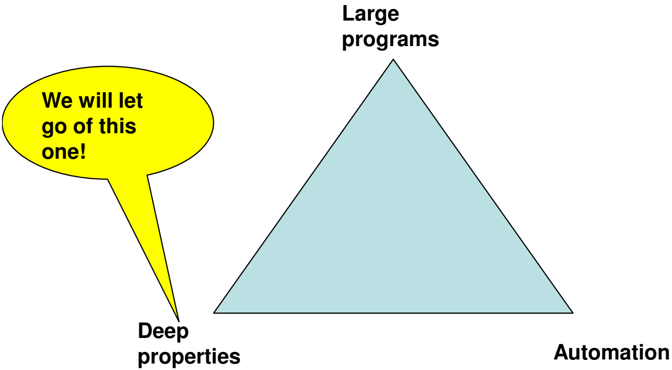
– *Manna and Pneuli, "Algorithmic Verification"*

Also known as: static analysis, static program analysis, formal methods, ...

Difficulty of program verification

- What will we prove?
    - "Deep" specifications of complex software are as complex as the software itself
    - Are difficult to prove
    - State of the art tools and automation are not good enough
- We will focus on "shallow" properties
    - That is, we will prove "partial correctness", or absence of certain classes of errors (e.g., null pointer dereferences)

Elusive triangle



Credit: Sriram Rajamani, Microsoft Research India

Example: Determining whether variables are odd (o) or even (e)

p = oddInput()	(p,*o*)	
q = evenInput()	(p,*o*)	(q,*e*)
if (p > q)	(p,*o*)	(q,*e*)
p = p*2 + q	(p,*e*)	(q,*e*)
write(p)	(p,*oe*)	(q,*e*)
if (p <= q)	(p,*o*)	(q,*e*)
p = p+1	(p,*e*)	(q,*e*)
write(p)	(p,*e*)	(q,*e*)
q = q+2	(p,*e*)	(q,*e*)

A verification approach: abstract interpretation

- A kind of program execution in which variables store *abstract* values from bounded domains, not concrete values
- Input values are also from the abstract domains
- Program statement semantics are modified to work on abstract variable values
- We execute the program on *all* (abstract) inputs and observe the program properties from these runs

Example: The abstraction

- Possible values of each variable: $\{o, e, oe\}$.
- Modified statement semantics:

$+$	o	e	oe
o	e	o	oe
e	o	e	oe
oe	oe	oe	oe

$*$	o	e	oe
o	o	e	oe
e	e	e	e
oe	oe	e	oe

Example: The abstract interpretation

Abstract interpretation

p = oddInput()	<(p,*o*)>
q = evenInput()	
if (p > q)	
p = p*2 + q	
write(p)	
if (p <= q)	
p = p+1	
write(p)	
q = q+2	

Example: The abstract interpretation

Abstract interpretation

p = oddInput()	<(p,*o*)>
q = evenInput()	<(p,*o*), (q,*e*)>
if (p > q)	
p = p*2 + q	
write(p)	
if (p <= q)	
p = p+1	
write(p)	
q = q+2	

Example: The abstract interpretation

Abstract interpretation

p = oddInput()	<(p,*o*)>
q = evenInput()	<(p,*o*), (q,*e*)>
if (p > q)	<(p,*o*), (q,*e*)>
p = p*2 + q	
write(p)	<(p,*o*), (q,*e*)>
if (p <= q)	
p = p+1	
write(p)	
q = q+2	

Example: The abstract interpretation

Abstract interpretation

p = oddInput()	<(p,*o*)>
q = evenInput()	<(p,*o*), (q,*e*)>
if (p > q)	<(p,*o*), (q,*e*)>
p = p*2 + q	<(p,*e*), (q,*e*)>
write(p)	<(p,*o*), (q,*e*)>
if (p <= q)	
p = p+1	
write(p)	
q = q+2	

Example: The abstract interpretation

	Abstract interpretation	
p = oddInput()	<(p,*o*)>	
q = evenInput()	<(p,*o*), (q,*e*)>	
if (p > q)	<(p,*o*), (q,*e*)>	
p = p*2 + q		<(p,*e*), (q,*e*)>
write(p)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
if (p <= q)		
p = p+1		
write(p)		
q = q+2		

Example: The abstract interpretation

Abstract interpretation

p = oddInput()	<(p,*o*)>	
q = evenInput()	<(p,*o*), (q,*e*)>	
if (p > q)	<(p,*o*), (q,*e*)>	
p = p*2 + q		<(p,*e*), (q,*e*)>
write(p)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
if (p <= q)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
p = p+1		
write(p)		
q = q+2		

Example: The abstract interpretation

Abstract interpretation

p = oddInput()	<(p,*o*)>	
q = evenInput()	<(p,*o*), (q,*e*)>	
if (p > q)	<(p,*o*), (q,*e*)>	
p = p*2 + q		<(p,*e*), (q,*e*)>
write(p)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
if (p <= q)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
p = p+1	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>
write(p)		
q = q+2		

Example: The abstract interpretation

Abstract interpretation		
p = oddInput()	<(p,*o*)>	
q = evenInput()	<(p,*o*), (q,*e*)>	
if (p > q)	<(p,*o*), (q,*e*)>	
p = p*2 + q		<(p,*e*), (q,*e*)>
write(p)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
if (p <= q)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
p = p+1	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>
write(p)	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>
q = q+2		

Example: The abstract interpretation

	Abstract interpretation	
p = oddInput()	<(p,*o*)>	
q = evenInput()	<(p,*o*), (q,*e*)>	
if (p > q)	<(p,*o*), (q,*e*)>	
p = p*2 + q		<(p,*e*), (q,*e*)>
write(p)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
if (p <= q)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>
p = p+1	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>
write(p)	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>
q = q+2	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>

Example: The abstract interpretation

Abstract interpretation			Ideal results	
p = oddInput()	<(p,*o*)>		(p,*o*)	
q = evenInput()	<(p,*o*), (q,*e*)>		(p,*o*)	(q,*e*)
if (p > q)	<(p,*o*), (q,*e*)>		(p,*o*)	(q,*e*)
p = p*2 + q		<(p,*e*), (q,*e*)>	(p,*e*)	(q,*e*)
write(p)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>	(p,*oe*)	(q,*e*)
if (p <= q)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>	(p,*o*)	(q,*e*)
p = p+1	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>	(p,*e*)	(q,*e*)
write(p)	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>	(p,*e*)	(q,*e*)
q = q+2	<(p,*e*), (q,*e*)>	<(p,*o*), (q,*e*)>	(p,*e*)	(q,*e*)

Example: The abstract interpretation

	Abstract interpretation		Ideal results	
p = oddInput()	<(p,*o*)>		(p,*o*)	
q = evenInput()	<(p,*o*), (q,*e*)>		(p,*o*)	(q,*e*)
if (p > q)	<(p,*o*), (q,*e*)>		(p,*o*)	(q,*e*)
p = p*2 + q		<(p,*e*), (q,*e*)>	(p,*e*)	(q,*e*)
write(p)	<(p,*o*), (q,*e*)>	<(p,*e*), (q,*e*)>	(p,*oe*)	(q,*e*)
if (p <= q)	<(p,*o*), (q,*e*)>	<(p,*e*)**X**, (q,*e*)>	(p,*o*)	(q,*e*)
p = p+1	<(p,*e*), (q,*e*)>	<(p,*o*)**X**, (q,*e*)>	(p,*e*)	(q,*e*)
write(p)	<(p,*e*), (q,*e*)>	<(p,*o*)**X**, (q,*e*)>	(p,*e*)	(q,*e*)
q = q+2	<(p,*e*), (q,*e*)>	<(p,*o*)**X**, (q,*e*)>	(p,*e*)	(q,*e*)

Another verification approach: Type systems

- Treat assignment statements as a *set* of mathematical equations, and program variables as mathematical variables.

  p = oddInput()
  q = evenInput()
  p = p*2 + q
  p = p+1
  q = q+2

- Let domain of variables be $\{o, e, oe\}$. Let operators "$*$" and "$+$" have the meanings as described in tables earlier.
- Solve the set of equations.

Another verification approach: Type systems

- Treat assignment statements as a *set* of mathematical equations, and program variables as mathematical variables.

  p = oddInput()
  q = evenInput()
  p = p*2 + q
  p = p+1
  q = q+2

- Let domain of variables be $\{o, e, oe\}$. Let operators "$*$" and "$+$" have the meanings as described in tables earlier.
- Solve the set of equations.
- Two solutions for the above equations: (1) $< p = oe, q = e >$, (2) $< p = oe, q = oe >$.
  - Solution (1) is more precise than solution (2).

Comparing abstract interpretation and type systems

- Reminder: The type solution is $< p = oe, q = e >$.
- Type systems approach is "flow insensitive": It gives each variable a single value valid at *all* program points, whereas abstract interpretation gives different values at different points.
- The single value is a over-approximation (union) of values at all program points. Therefore, type system approach is *less precise* than flow-sensitive abstract interpretation.
- However, type system approach is more efficient.

  Both approaches produce over-approximations of the ideal results. *This is true of verification approaches in general.* In contrast, testing produces an under-approximation of the ideal results.

- In other words, Flow-insensitive verification $\supseteq$ flow-sensitive verification $\supseteq$ ideal results $\supseteq$ testing.

Overview of PAV course

- Introduction (1 lecture) (*durations are tentative*)
- Specifying semantics of programming language formally. (1)
- Verification approaches
  - Dataflow analysis (7)
  - Abstract interpretation (3)
  - Type inference (8)
  - Assertional reasoning (2)
- Program slicing (6) (*Time permitting*)

Flavour of the course

- Semantics: associating a mathematical function with each kind of statement in the language.
- Dataflow analysis
  - Setting up a set of mathematical equations, and using a kind of graph traversal to solve these equations
  - Proving termination of the approach
- Abstract interpretation and type systems
  - Examples of abstract domains and abstract statement semantics
  - Proving that the results computed are an over-approximation of the ideal results
  - Proving termination of the approach
- Assertional reasoning: A first-order predicate logic for deriving facts about a program

Prerequisites

- Discrete structures such as sets, relations, partially ordered sets, functions
- (Undergraduate level) algorithms
- Mathematical logic (propositional, first-order)
- General mathematical maturity: comfort with notation, understanding and writing proofs
- Familiarity with imperative languages like C
- (Moderate) programming experience

What we will not cover

- Software engineering
  - How to collect requirements from customers and prioritize them
  - Planning and management of software development
  - Design, architecture, coding
- Programming languages
- Analysis of parallel/concurrent programs, distributed systems

What we will not cover

- Software engineering
  - How to collect requirements from customers and prioritize them
  - Planning and management of software development
  - Design, architecture, coding
- Programming languages
- Analysis of parallel/concurrent programs, distributed systems

Compilers course offered by Prof. Y. N. Srikant this semester will cover applications of program analysis to compiling, among other topics.

Assignments and exams (tentative)

- Assignments
  - 5-6 assignments
  - Most of them written, some involve coding
  - 50% weight
- Mid-sem exam (20%), End-sem exam (30%)

Misconduct policy

- Academic misconduct (e.g., copying) will not be tolerated
- Discussion in exams $\Rightarrow$ automatic fail grade for both students
- Assignments
    - Try to work individually.
    - If you choose to discuss with other students
        - You may discuss only with students registered in the class (or with the Deepak or Raghavan)
        - You must write your answer individually, in your own words. No copying, no looking at the other person's answer!
    - For *each* violation of above policy $\Rightarrow$ zero for the entire assignment *plus* one grade-point reduction in final grade (for the one who copied).
        - Grade-point reductions over multiple violations will accumulate.
    - **Grading:** Your marks will be based on your written answer *and* on a viva. (There will be a viva for each assignment.)

Late policy for assignments

- 10 "free" late days for use over all assignments.
- For each late day after free days have been exhausted $\Rightarrow$ 25% penalty on the assignment marks. (Weekends and weekdays treated the same.)
- No late days allowed on final assignment.