Static Analysis for Software Verification

Leon Moonen

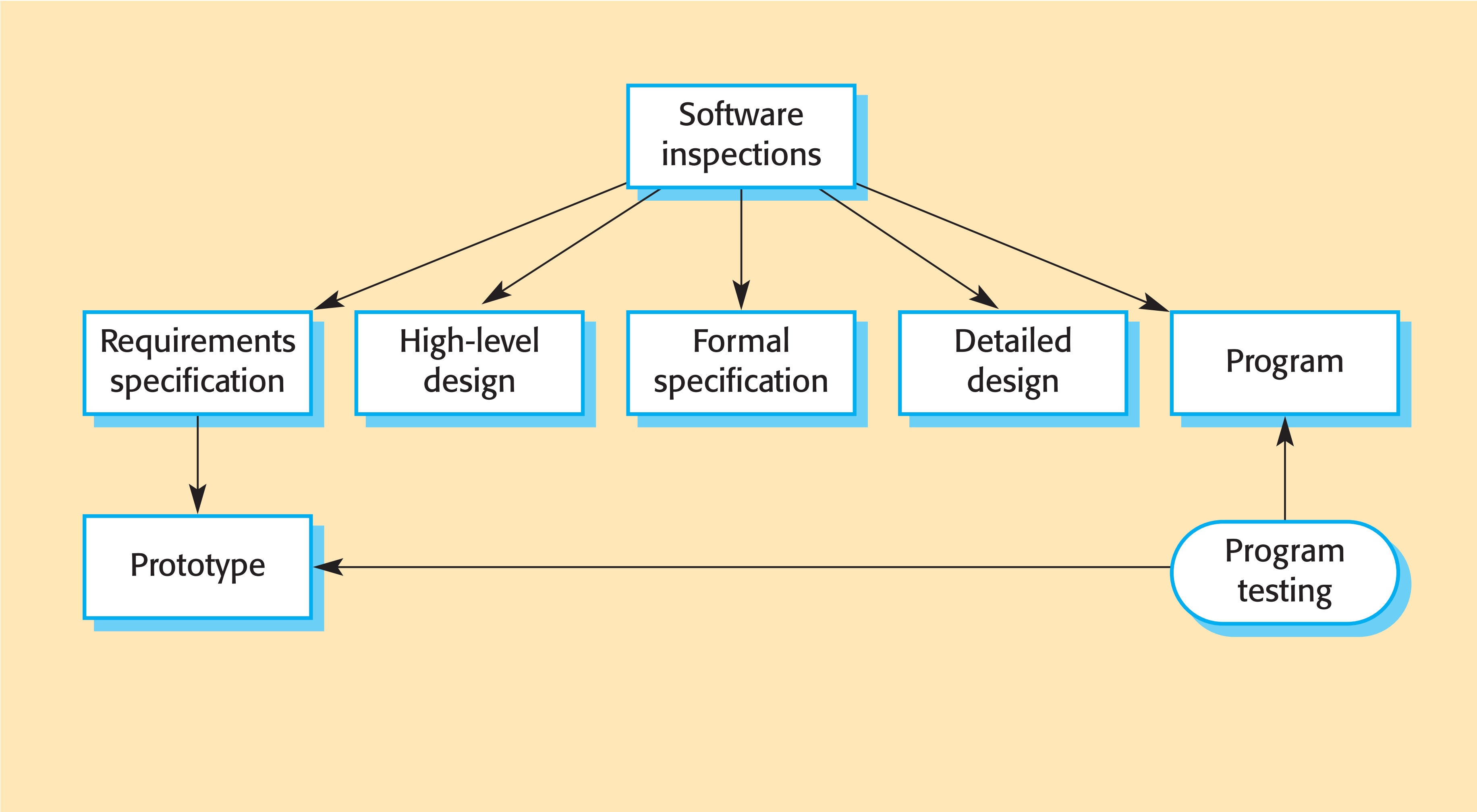
Today's topics

- Software inspection
  - it's relation to testing
  - benefits and drawbacks
- Static (program) analysis
  - potential benefits
  - limitations and their cause
  - examples
  - building blocks of static analysis tools
  - static analysis tools in practice
- Improving static analysis tools
  - prioritize based on static profiling
  - by learning from the past

Software Inspection

- examine representation of a software system with the aim of discovering anomalies and defects
  - *“check software artifacts for constructs that are known to be problematic from past experience”*
- systematic & detailed review technique
  - peer review (not author or his boss but inspection team)
- applicable to all kinds of software artifacts
  - requirement specification, design documents, source code, …
  - e.g. last week you heard about requirement inspections, this week we’ll focus on code inspections
- defined in the 70’s by Fagan (IBM)
  - several alternatives & extensions proposed that vary in rigorousness of the review and focus on particular goals
  - pair programming can be seen as a light & informal instance

Inspections in relation to testing



Inspection benefits

- inspections do not require execution of a system, so can be done before implementation is completely finished
  - many different defects may be discovered in a single inspection
  - in testing, one defect may mask another so repeated executions are required
- inspections reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise

Inspection benefits (2)

- inspections identify defects that are missed by testing
  - inspection does not replace testing (nor the other way around)
  - complementary techniques; find different types of faults
  - various studies that compare code inspection and testing approaches to QA
    - general conclusion: there is no clear 'best' approach, but techniques found different faults; recommended combining them

      [Hetzel, Myers, Basili & Selby, Kamsties & Lott, Wood et al.]
- inspections helps to educate new team members
  - learn by collectively going through a design, code, …
  - make people (more) aware of desired quality standards

Inspections known to be very effective

Benefits of Formal Inspections

- Formal inspection works well for programming:
  - For applications programming:
    - more effective than testing
    - most reviewed programs run correctly first time
      - compare: 10-50 attempts for test/debug approach
  - Data from large projects
    - error reduction by a factor of 5; (10 in some reported cases)
    - improvement in productivity: 14% to 25%
    - percentage of errors found by inspection: 58% to 82%
    - cost reduction of 50%-80% for V&V (even including cost of inspection)
  - Effects on staff competence:
    - increased morale, reduced turnover
    - better estimation and scheduling (more knowledge about defect profiles)
    - better management recognition of staff ability
- These benefits have been shown to apply to requirements inspections too

[simula . research laboratory] 16

remember the data shown last week

Inspection drawbacks

- (originally) manual process with strict guidelines
  - time-consuming (expensive)
  - tedious and error-prone (cannot be done full-time)
- inspections increase costs early in the software process
- precise standards or guidelines must be available and inspection team members must be familiar with them
- not incremental
  - repeated inspection costs same as first one

- as a consequence, (formal) inspections often end up not being performed well, or even abandoned

Is there a way out?

- *static analysis* aims at getting the traditional code inspection benefits by using automated checks
  - avoiding the drawbacks mentioned

- note that we are now focusing on *code inspections*

What is static analysis?

- analyzing what a program does without executing it
  - by careful examination of the program’s source code
- static analysis can diagnose:
  - violations of rules and conventions that
    - are needed for correct program execution (i.e. defect finding)
    - are desired for certain non-functional quality aspects such as maintainability and complexity (but not usability or performance)
  - coding standard compliance
  - best programming practices and unsafe programming
- so static analysis can be used to find a lot of the issues that traditional (manual) inspections aimed at
  - enables ”inspection for the masses”
  - additional inspections can still pay off (e.g. safety critical sw.)

A simple static analysis example

- most of you know a simple static analysis by heart…
  - is the outcome of 346288 * –8782332 positive or negative?
  - and what about –439232 * –2323347 ?

- the ‘rule of signs’

  + * + = +

  + * – = –

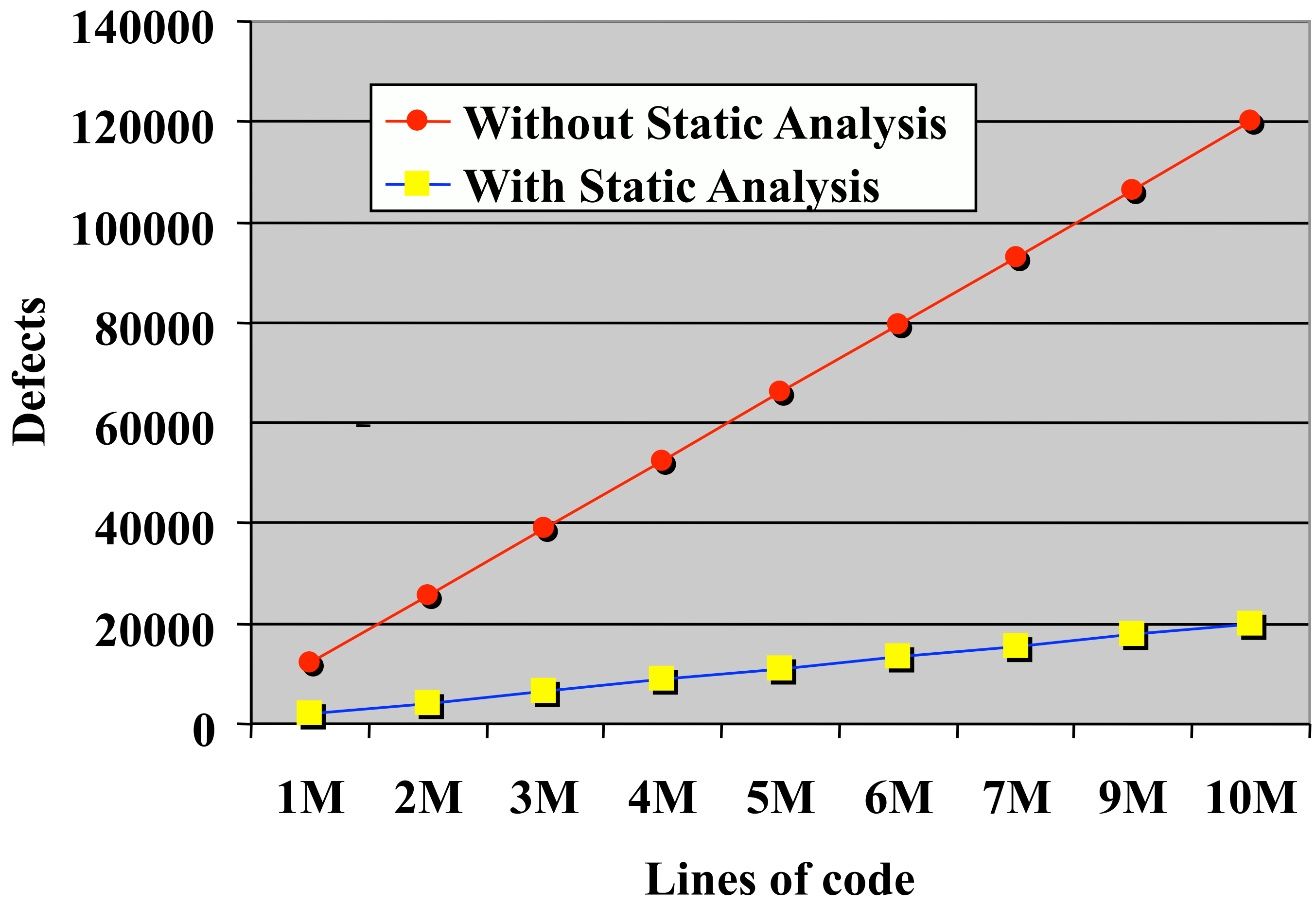
  – * – = +

  allows to statically analyze the sign of the outcome of a multiplication without actually doing the computation

Properties of static analysis

- advantage:
  - static analysis provides information that is valid for all possible executions of the program
- disadvantage:
  - the information provided is not always precise since it is usually based on an approximation
- compare: testing is a form of dynamic analysis
  - advantage:
    - detailed and precise info for a single run (test case)
  - disadvantage:
    - no guarantees about other runs (can be addressed by doing multiple runs, each exercising different paths)

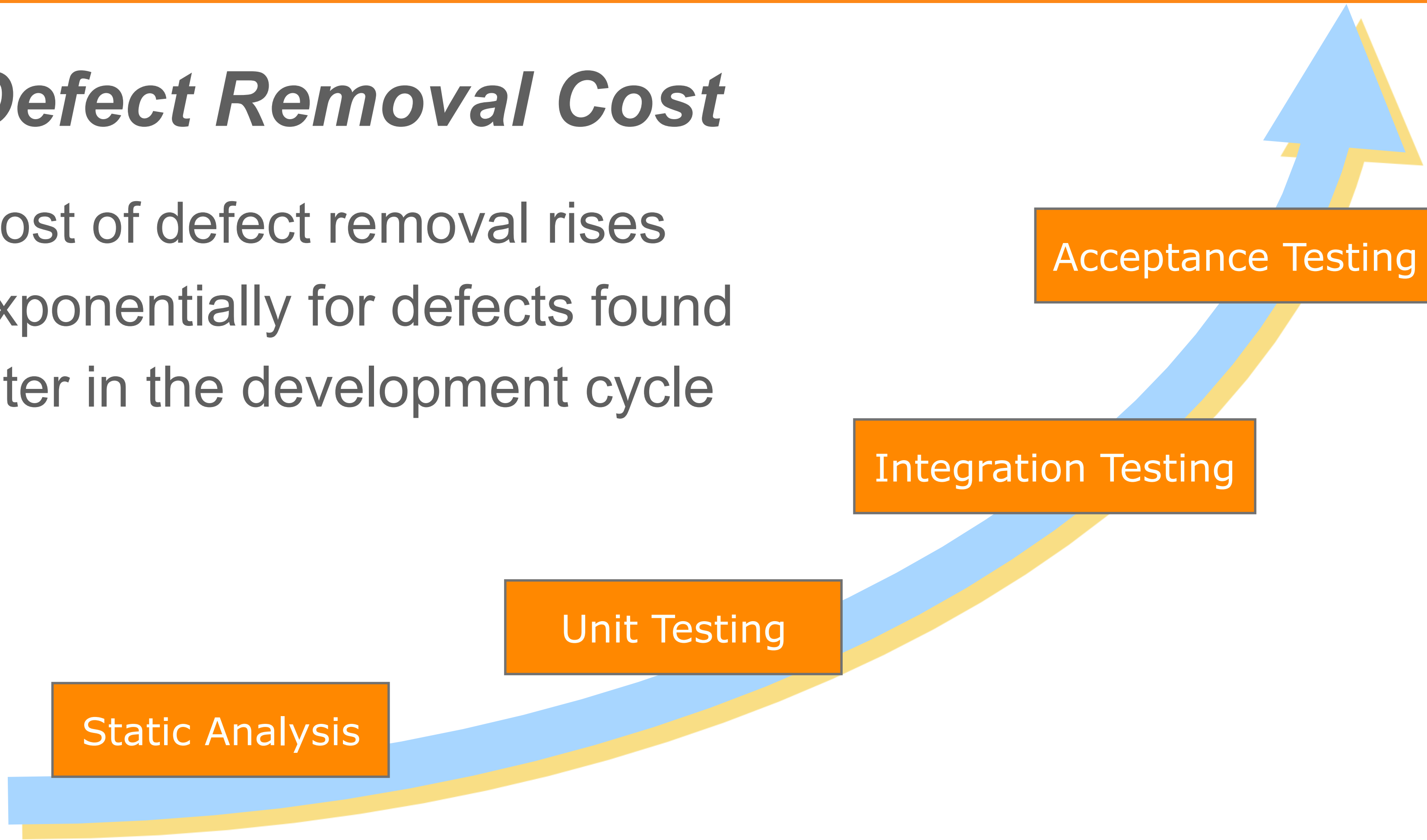
Potential benefits are high



Static Analysis can reduce defects by up to a factor of six
[Capers Jones, Software Productivity Group]

Defect Removal Cost

Cost of defect removal rises exponentially for defects found later in the development cycle



Rule of thumb: A defect that costs $1 to fix on the programmer's desktop costs $100 to fix once it is incorporated into a complete program and many thousands of dollars if it is identified only after the software has been deployed in the field

[Building a Better Bug Trap, The Economist, June 2003]

More findings on static analysis

- “We proved the tight relationship between static analysis and the reduction of support efforts on released software products.“

  Dr. Thomas Liedtke and Dr. Christian Ebert, Alcatel Germany, On the Benefits of Reinforcing Code Inspection Activities, EuroStar 1995

- “60% of the software faults that were found in released software products could have been detected by means of static analysis”

  Bloor Research Ltd., UK CAST Tools report of 1996

- “On average, 40% of the faults that could be found through static analysis will eventually become a defect in the field.”

  professor Dr. Les Hatton, University of Kent

For the software manager

- static analysis helps to
  - reduce risk of expensive after-deployment bugs
  - reduce time to market
  - reduce cost & time of code review and testing
    - automate (part of) review, no or more limited manual inspections
    - removing obvious bugs improves focus & speed of testing
  - improve code quality (adhere to coding standards)
  - achieve higher coverage (more code is checked)
    - related to, but not same as testing coverage, since focus differs

For the software developer

- static analysis helps to
  - find / prevent bugs earlier (before they are hard to fix)
    - tools can be used as part of development cycle, like a compiler
    - more direct and obvious feedback
  - find / prevent “hard to test” bugs
    - e.g., good at detecting potential memory leaks & buffer overflows
    - make developers more efficient
    - spend less time debugging

Example from LINT static analysis tool

```
> cat lint_ex.c
#include <stdio.h>
printarray (Anarray)
 int Anarray;
{   printf("%d",Anarray);  }

main () {
 int Anarray[5]; int i; char c;
 printarray (Anarray, i, c);
 printarray (Anarray) ;
}

> cc lint_ex.c
> lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Other examples of static analysis tools

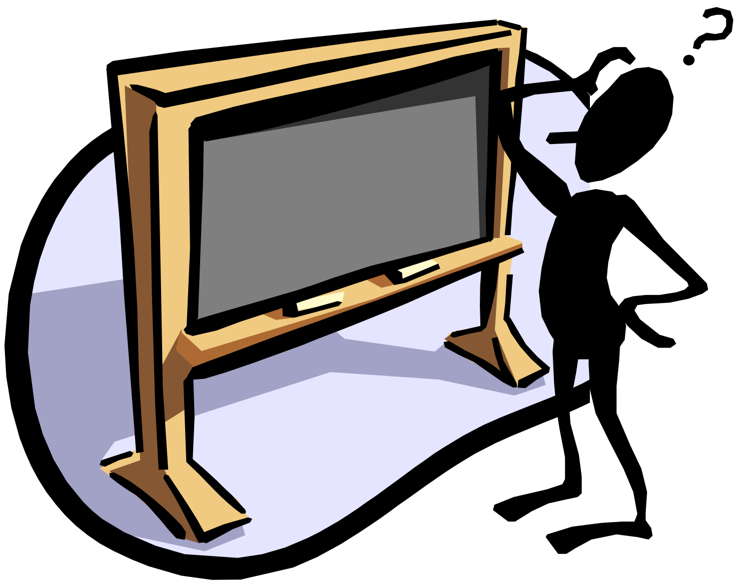
- several open source tools for Java, e.g.
  - Findbugs http://findbugs.sourceforge.net/
  - QJPro http://qjpro.sourceforge.net/
  - PMD http://pmd.sourceforge.net/
- most of these integrate with an IDE like Eclipse
- also other languages like FxCop for .NET, Perl::Critic
- many commercial tools
  - often aimed at ‘industrial’ languages such as C, C++, Ada
  - typically come with rulesets to check conformance with various coding standards (e.g. the MISRA C standard)
  - some vendors include Coverity, Grammatech, Klocwork, QA-C
  - licenses typically based on # lines of code to analyze

How does it work?

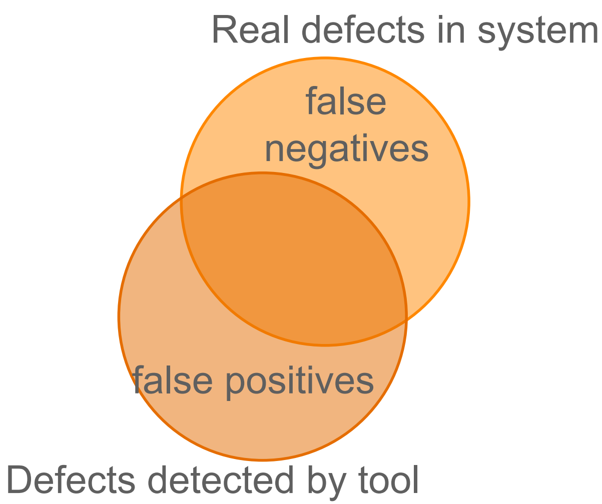
- static analysis tools parse the source code and build one or more abstractions (models) of the system
  - typically typed (colored) and directed graphs
  - staged process, looks a lot like compilation
- abstractions are used to check if certain properties hold
  - check violations based on a given set of well-defined rules
  - by performing graph traversals (sometimes transformations)
- if the tool finds a violation in the abstraction, the violation is expected also to be in the source code
  - provided that the abstraction is correct for this particular check

Types of imprecision

- suppose we have an static analysis tool that checks for violation of certain coding rules
- what could go wrong?



- defects can be missed
  - false negatives
- tool can report defects where there are none
  - false positives



Limitations: Dealing with undecidabilty

- halting problem makes it undecidable to prove non-trivial properties of a program (in general)
  - static analysis tools will need to work around this
- only analyze loop-free programs
  - works in some situations, not desirable in general
- analyze simple static properties
  - e.g. metrics collected to assess complexity (size, McCabe)
- use an interactive solution: human-in-the-loop
- make conservative estimates
  - e.g. every statement in a loop is executed at least once, in a conditional statement both branches are executed, …
  - the aim is that every potential defect is caught
    - but some "false alarms" may be triggered

Limitations (2)

- static analysis computes an approximation of what would happen during execution
  - such an approximation is bound to have imperfections
    - some false positives (noise)
    - some false negatives (didn’t find what it should have)

- don’t believe that a program is good because the analysis tool says so
  - it’s simply saying that it couldn’t find anything!

  - from an internal Microsoft presentation on their in-house static analysis tools for driver verification (SLAM/SDV & PRE*fast*)
    - *“The biggest risk with using static tools is overconfidence in the results. The second biggest risk is not using them at all.”*

Building blocks of static analysis tools

- syntactic analysis
  - parse the code, build syntax tree
  - can be used to check some coding standards
  - e.g. missing ‘default’ or ‘break’ in a switch statement
  - first step of many other analyses…
- control flow analysis
  - can be used to detect poorly structured code, dead code and some cases of non-termination
  - e.g. find multiple exits from a loop
  - concepts already explained in the lecture on whitebox testing
- data flow analysis
- program slicing

Data Flow Analysis

- can be used to identify data flow that does not conform to sound programming practices, e.g. variables are not read before they are written; inactive code.
  - pure symbolic analysis, i.e. no specific values are used
  - based on relationships between variables and expressions
- annotate the control flow graph with data definitions (D), uses (U) and kills (K) (point where an earlier definition is invalidated)
- traverse the graph
  - DD paths suggest redundancy (why define twice?)
  - DK paths point at potential bugs (why kill a value before use?)

Program Slicing

- program slicing is a technique to zoom in on a particular subset of variables within a given program
- the part of the program that is relevant to the chosen subset of variables is called the program slice
  - i.e. a “subprogram” with all non-relevant statements removed
- various applications:
  - program testing & static analysis:
    - reduce program to limit overhead (e.g. warnings) from parts that you’re currently not interested in
    - can help to address scalability issues
  - program comprehension & debugging:
    - reduce program to make it easier to understand and analyze

Types of program slicing

- backward slicing:
  - for a given statement S, a backward slice through a program contains all statements that effect whether control reaches S and all statements that effect the value of variables in S
- forward slicing:
  - for a given statement S, a forward slice through a program contains all statements that are affected by S

- note that these slices can be calculated either statically or dynamically:
  - a static program slice is calculated symbolically, i.e. takes no account of concrete data values
  - a dynamic program slice is calculated based upon particular data values

Program Slicing Example

Program:

```
read(X);
read(Y);
Q := 0;
R := X;
while R >= Y do
   begin
   R := R - Y;
   Q := Q + 1
end;
print(Q);
print(R);
```

A program slice for R:

```
read(X);
read(Y);
R := X;
while R >= Y do
   begin
   R := R - Y;
end;
print(R);
```

Q: what statements would be in a slice for R ?

Static analysis tools in practice

- detailed analyses require considerable power
  - however, computer power still doubles every 18 months
  - and tools improve rapidly
    - software security analysis is a big driver
- not everyone happy with amount of data generated ;-)
  - in practice there can be many false positives (>50%)
  - lot's of complaints over trivial issues make introduction hard
  - prioritization of reported warnings is needed
- not always easy to add new rules to be checked
  - bug patterns can be obscure and depend on control & dataflow patterns (not just simple syntactic matching)

Making better static analysis tools

- we have conducted research on techniques that may help to improve on the results of existing tools
  - i.e. don’t compete and build a new static analysis tool but build a pre- or postprocessor that can be used with all tools

- two approaches investigated:
  1. aimed at improving the prioritization of inspection results
  2. aimed at reducing noise in the list of warnings

[based on joint work with Cathal Boogerd at Delft Univ. of Technology]

Prioritizing Inspection Results

- After getting a huge list of violations from a static analysis tool, the question arises: *“Where to start?”*
- our approach: determine *execution likelihood* of the violation locations and use it to prioritize the list
  - given a program P and location v, the execution likelihood Ev is the probability that v is executed in an arbitrary run of P
  - likelihood can be approximated by static profiling: an analysis of the program’s control-flow structure
    - simplistic: statements inside each of the branches of an in-then-else are half as likely to be executed than the statement itself
    - actually: compiler literature has heuristics for these probabilities depending on the types and comparison in the condition
      - e.g. integer comparison ‘less than zero’ likely to fail

Prioritizing Inspection Results (2)

- the above approach can be considered rather crude
  - developed “deeper” static profiling algorithms
  - use detailed data flow analysis (value range propagation) to better estimate which branch is taken
  - these did not systematically outperform the one above

- prioritization based on execution likelihood leads developer to violations that have high change of being triggered during an actual run
  - aimed at fixing issues with most impact first
  - but these might be found anyway, whereas faults deep down in the system are, in a way, hidden further from inspection…

Filter by learning from the past

- do violations of certain rules really indicate faults?
  - can we use this to prioritize or filter based on relevance?
- analyzed history of three (related) software projects
  - using a newly introduced coding standard checker
- compute *true positive rates* for rules
  - a ‘true positive’ is a violation in release *n* that correctly predicted a line to be faulty, i.e., part of a bug fix in a later release >*n*
- use these values to identify ‘significant rule sets’
  - those rules where the violation predictions outperform a random guess

Filter by learning from the past

- some conclusions:
  - partially consistent behavior of rules for the three cases
    - so it's possible to select a rule set within a product family
  - historical true positive rates help select effective subset of rules
    - selected sets cover 64%-86% of issues while reducing the number of violations by 63%-95%
- results from the past can give guidance for the future

Static analysis and formal methods

- formal methods can be used when a mathematical specification of the system can be created
- the analysis creates a formal argument that a program conforms to its mathematical specification (or refutes this by showing a counter example)
- advantages:
  - very precise; the ultimate static verification technique
  - creating the specification may already uncover errors
- disadvantages:
  - the pure version does not scale well to real-size programs
    - can be addressed by special measures (abstractions) that however reduce precision…
- you will hear more about *model checking* later

Summary

- software inspections are very effective way of QA
  - used in addition to testing, focus on different types of faults
- manual code inspections are expensive and error-prone
  - as a result not very popular outside specific domains (safety)
- these drawbacks can be partly removed by using automatic static analysis tools
  - we have seen how static analysis tools are made
  - and discussed why they have (inherent) limitations
  - presented some research aimed at better dealing with these

“The biggest risk with using static tools is overconfidence in the results. The second biggest risk is not using them at all.”

References

- *Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182–211*
- *Gilb, T. and Graham, D. (1993). Software Inspection. Addison-Wesley*
- *Young, M. and Pezze, M. (2007), Software Testing and Analysis*
- *Boehm, B. W. (1981). Software Engineering Economics. Prentice Hall*
- *Nagappan et al. (2004) “Preliminary Results on Using Static Analysis Tools for Software Inspection” (ISSRE)*
- *Boogerd, C. and Moonen, L. (2008). Assessing the value of coding standards: An empirical study (ICSM)*
- *Boogerd, C. and Moonen, L. (2009b). Evaluating the relation between coding standard violations and faults within and across versions (MSR)*