Verification of the source code

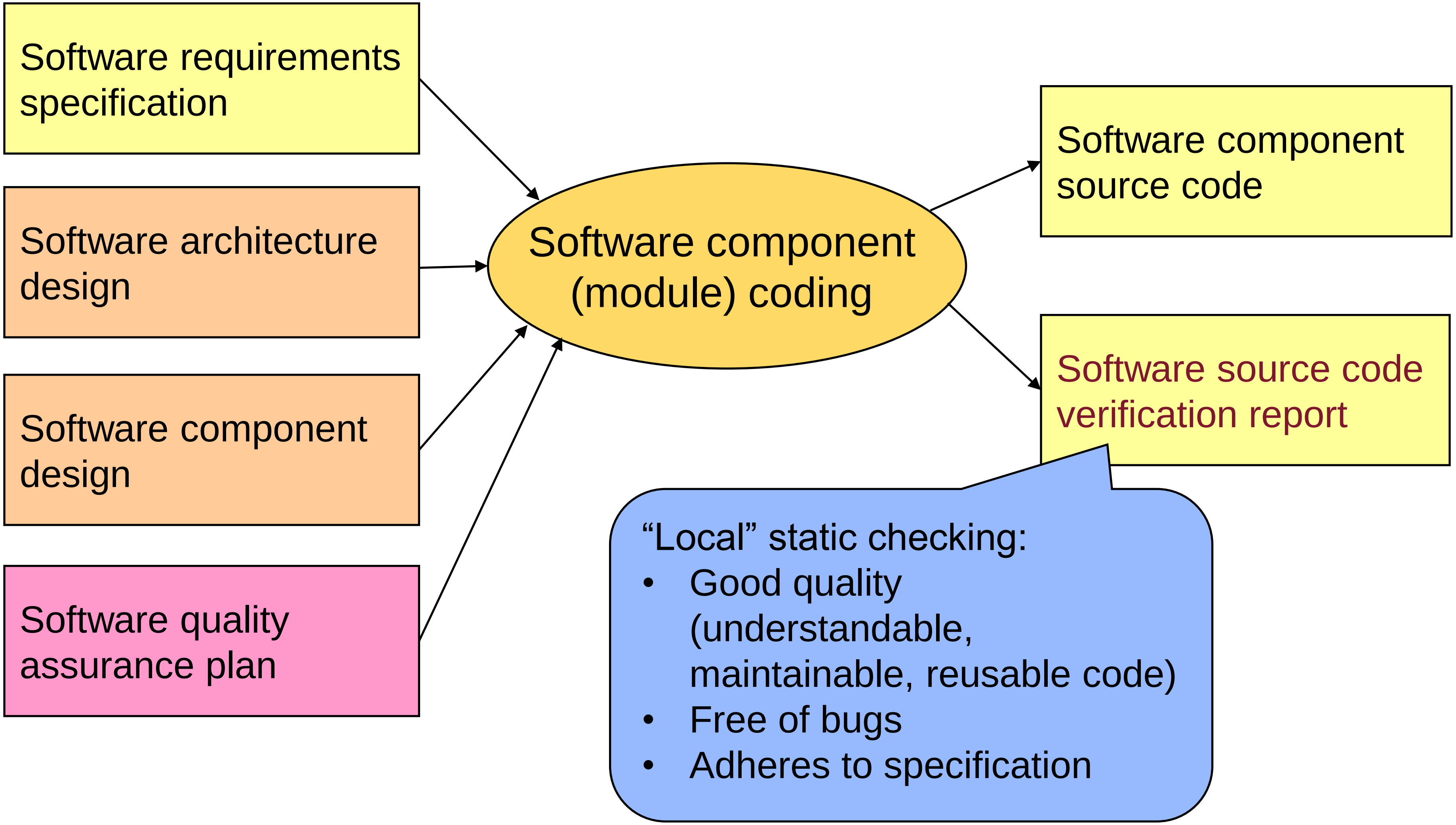
Istvan Majzik
majzik@mit.bme.hu

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Typical development steps and V&V tasks

Development step	V&V tasks
Requirement analysis	
System specification	• Checking completeness, consistency, feasibility, verifiability • Assuring traceability
Architecture design	• Trade-off analysis, interface analysis, fault effects analysis • Model based quantitative evaluation
Module design	• Formal verification by (temporal logic based) model checking • Equivalence checking
Module implementation	• Source code analysis • …
System integration	
System delivery	
Operation, maintenance	

Inputs and outputs of the phase



Overview: What is checked?

- Checking coding guidelines
  - Domain / platform / company specific rules
  - Well-known coding standards (guidelines)
- Checking software metrics
  - Estimation of quality aspects (e.g., maintainability)
  - Relation of metrics and fault-proneness
- Checking fault patterns by static analysis
  - Extensible tools
- Checking runtime failures by code interpretation
  - Static verification of dynamic properties

Checking coding guidelines

Coding guidelines – introduction

- Set of rules giving recommendations on
  - Style: formatting, naming, structure
  - Programming practices: constructs, architecture
- Main categories
  - Industry/domain specific
    - Automotive, nuclear, …
  - Platform specific
    - C, C++, C#, Java, …
  - Organization specific
    - Google, CERN, …

Coding guidelines in critical systems (standards)

- Programming style
  - Code formatting, comments, source code metrics
- Restricted constructs
  - Recursion, pointers, automatic type conversion, unconditional branch, …
  - OO constructs: Polymorphism, multiple inheritance, embedding objects, runtime construction and destruction of objects
- Programming languages (e.g., in EN50128):
  - Analyzable, strongly typed, structured or OO language
  - SIL1-SIL4 HR: Ada, Modula-2, Pascal
  - SIL1-SIL4 NR: BASIC, SIL3-SIL4 NR: unconstrained C/C++
  - SIL3-SIL4 R: C and C++ with coding rules (language subset)
- Tools (compilers, libraries):
  - Certified, validated or proven-in-use

Example: Part of SoHaR guidelines (nuclear industry)

Group	Number	Guideline
	1	Reliability
	1.1	Predictability of Memory Utilization
Specific	1.1.1	Minimizing Dynamic Memory Allocation
Outside	1.1.2	Minimizing Memory Paging and Swapping
	1.2	Predictability of Control Flow
Specific	1.2.1	Maximizing Structure
Specific	1.2.2	Minimizing Control Flow Complexity
Specific	1.2.3	Initialization of Variables before Use
Specific	1.2.4	Single Entry and Exit Points in Subprograms
Specific	1.2.5	Minimizing Interface Ambiguities
Specific	1.2.6	Use of Data Typing
General	1.2.7	Precision and Accuracy
Specific	1.2.8	Use of Parentheses rather than Default Order of Precedence
Specific	1.2.9	Separating Assignment from Evaluation
Outside	1.2.10	Proper Handling of Program Instrumentation
General	1.2.11	Control of Class Library Size
General	1.2.12	Minimizing Dynamic Binding
General	1.2.13	Control of Operator Overloading
	1.3	Predictability of Timing
Outside	1.3.1	Minimizing the Use of Tasking
Outside	1.3.2	Minimizing the Use of Interrupt Driven Processing

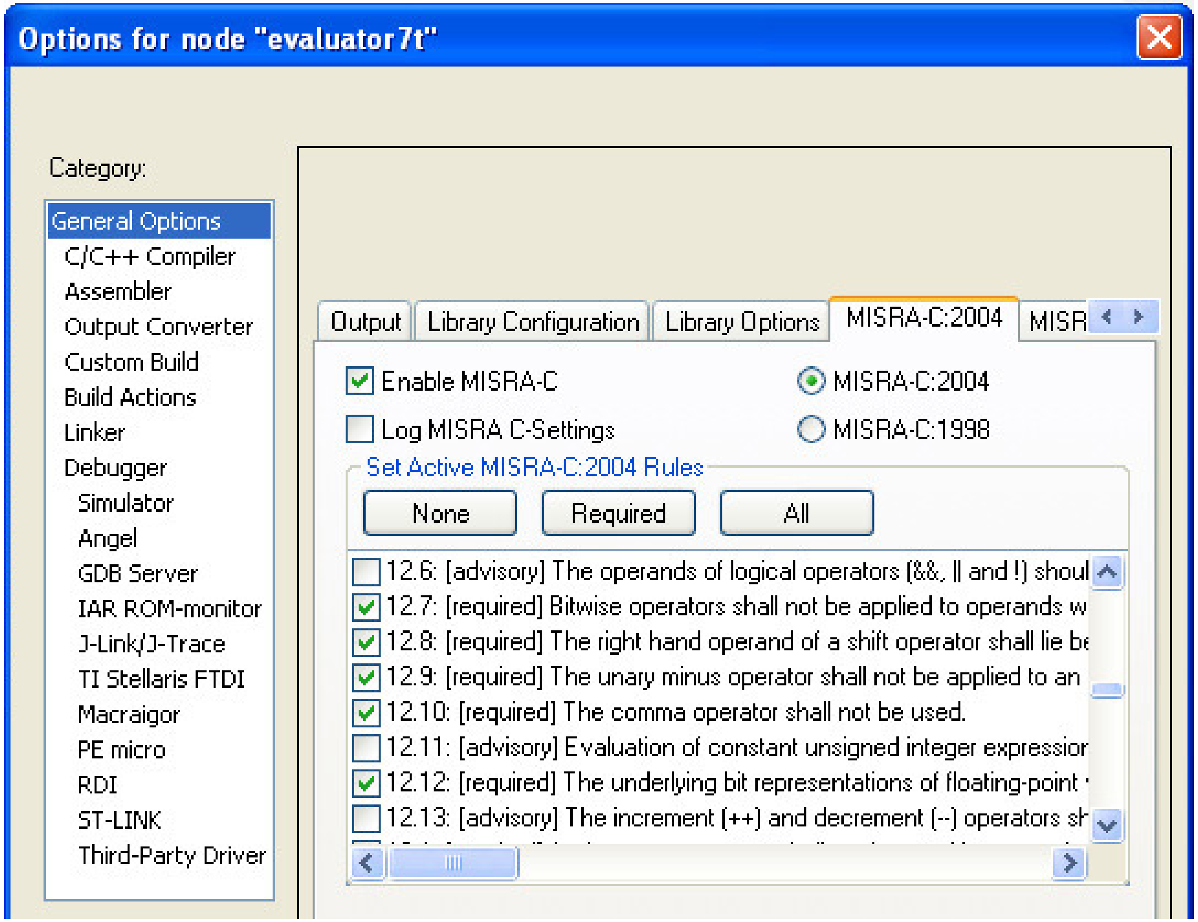
C and C++ coding guidelines (rule sets)

- MISRA C (Motor Industry Software Reliability Association)
  - MISRA C:2004: 142 rules (122 mandatory)

    Examples:
    - Rule 33 (Required): The right hand side of a "&&" or "||" operator shall not contain side effects.
    - Rule 49 (Advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
    - Rule 59 (R): The statement forming the body of an "if", "else if", "else", "while", "do ... while", or "for" statement shall always be enclosed in braces.
  - MISRA C:2012: 143 rules + 16 directives
    - Rules: For static checking of the source code
    - Directives: Related to process, design documents
- MISRA C++ (2008): 228 rules
- US DoD, JSF C++: 221 rules (including code metrics)
  - „Joint Strike Fighter Air Vehicle C++ Coding Standard"

Example: Checking MISRA compliance

- Tools for checking MISRA compliance
  - LDRA, IAR Embedded Workbench, QA-C, SonarQube, Coverity, …



Example: Compiler-dependent implementation

- Results of integer division depending on compiler implementation:
  - (-5/3) may be -1 and the remainder is -2, or
  - (-5/3) may be -2 and the remainder is +1
- Out-of-range results when adding or multiplying integers:

```
uint16_t u16a = 40000;          /* unsigned short / unsigned int ? */
uint16_t u16b = 30000;          /* unsigned short / unsigned int ? */
uint32_t u32x;                  /* unsigned int / unsigned long  ? */

u32x = u16a + u16b;             /* u32x = 70000 or 4464 ?          */
```

  - If the addition is implemented using unsigned short (16 bits) corresponding to the types of the operands then overflow may occur
  - If the addition is implemented using unsigned int (32 bits) corresponding to the type of the result then there is no overflow
- These compiler-dependent implementations have to be validated (tested)

Checking software metrics

Software source code metrics

- Goals
  - Measurable characteristics of the source code
  - Linked with the quality of the source code
- Quality aspects for source code (MISRA)
  - Complexity
  - Maintainability
  - Modularity
  - Reliability
  - Structuredness
  - Testability
  - Understandability
  - Maturity
- Besides quality, costs can also be estimated on the basis of metrics
  - Cost of development, testing, maintenance

Example: MISRA metrics

Software Attributes	Type of Technique	Area of Application	Technique or Metric
Structuredness	Method	Component Source Code	Interval Reduction
	Metrics	Component Source Code	Cyclomatic Number Essential Cyclomatic Complexity Number of Entry Points Number of Exit Points Number of Structuring Levels Number of Unconditional Jumps Number of Execution Paths
	etrics	mponent ource Code	Cyclomatic Number Number of Distinct Operands Number of Unconditional Jumps Number of Execution Paths Number of Decision Statements IB Coverage DDP Coverage LCSAJ Coverage PPP Coverage
		System Source Code	Number of Calling Paths Number of Components IB Coverage DDP Coverage LCSAJ Coverage PPP Coverage

Cyclomatic Number: „Number of basic paths through the component which can generate every possible path of a component."

Essential Cyclomatic Complexity: „Computed by reducing the control flow graph by systematically (from the inner parts) replacing structured code blocks with a single node"

Example: Limits for MISRA metrics

Average number of operators and operands in statements

CSC = Cyclomatic Number * (Fan-In * Fan-Out)^2

Structured control flow graph (ESC=1)

Average number of components at call levels in the function call tree

Software Metric	Area of Application	High Level Languages		Low Level Languages	
		Min	Max	Min	Max
Average Statement Size	Component	2	8	N/A	N/A
Comment Frequency	Component	0.5	1	1	1
Component Length	Component	3	250	3	250
Component Stress Complexity	Component	1	10000	1	10000
Cyclomatic Number	Component	1	15	1	15
DDP Coverage	Both	80%	100%	80%	100%
Essential Cyclomatic Complexity	Component	1	1	1	1
Hierarchical Complexity	System	1	5	1	5
IB Coverage	Both	100%	100%	100%	100%
LCSAJ Coverage	Both	60%	100%	60%	100%
Number of Calling Levels	System	1	8	1	8
Number of Calling Paths	System	1	250	1	250
Number of Components	System	1	150	1	150
Number of Decision Statements	Component	0	8	0	8
Number of Distinct Operands	Component	1	50	1	50
Number of Distinct Operators	Component	1	35	1	35

Categories of OO metrics

- Size: Counting source code elements
  - Number of code lines, attributes, methods (private/public/protected)
- Complexity: Cyclomatic numbers
  - CK: Max. number of independent paths in the control flow graph
  - Sum of cyclomatic complexities of methods
- Coupling: How many elements of other classes are used
  - Number of (directly) called methods
  - Number of classes with called method or used attribute
- Inheritance: Based on the inheritance graph
  - Number of levels below / above a given class, directly / all
  - Number of inherited methods
- Cohesion: Links among the methods and attributes of a class
  - Number of methods sharing (using together) an attribute
  - Number of methods calling each other

Correlation of OO metrics and fault-proneness (1)

- Goal: Prediction of the fault-proneness of classes
  - To support focusing the testing activities on risky classes
- Experiments: Measuring correlation of metrics and number of bugs detected in a class during testing
  - Open source projects were examined (Mozilla, 4500 classes)
  - Bugs recorded in bug databases were analyzed (Bugzilla, 230 000 bugs)

Inefficient metrics for fault-proneness prediction:

- Inheritance category
  - NOA: Number of Ancestors
  - NOC: Number of Children
- Cohesion category
  - LCOM: Lack of Cohesion in Methods: Number of method pairs that do not share attribute minus the number of methods that share

Correlation of OO metrics and fault-proneness (2)

Efficient metrics for fault-proneness prediction:

- Coupling category:
  - CBO (Coupling Between Objects): Number of classes coupled with the examined class (calling their methods, using attributes, or inherit)
  - NOI (Number of Outgoing Invocations): Number of directly called methods
  - RFC (Response Set of a Class): Number of methods of the class + directly called other methods
  - NFMA (Number of Foreign Methods Accessed): Number of foreign methods (not owned and not inherited) that are directly called
- Size category:
  - NML (Number of Methods Local): Number of local methods of a class
  - LLOC (Logical Lines of Code): Number of lines that are not empty and not comment only

Checking fault patterns by static analysis

Pattern based tools

Types of static analysis tools

- Early tools: syntactic „well-formedness” checking
  - Examples: Lint (for C, from 1979, Bell Labs), JLint (for Java)
- Static analysis tools looking for fault patterns
  - Built-in fault patterns (bad practice) + extensible by new patterns
  - Checking is not safe (false errors may occur)
  - Examples: FindBugs, PMD (Java), Gendarme (.Net CIL), …
- Static analysis tools using abstract code interpretation
  - Computing the ranges of variables in program statements
  - Detecting arithmetic overflow, underflow, out-of-bound indexing etc.
  - Examples: CodeSurfer, CodeSonar (C/C++, template based), Prevent (MS Win API, supporting PThreads), Klocworks

Example: Fault categories and patterns in FindBugs

- Bad practice
  - Random object created and used only once
- Correctness
  - Bitwise add of signed byte value
- Malicious code vulnerability
  - May expose internal static state by storing a mutable object into a static field
- Multithreaded correctness
  - Synchronization on Boolean could lead to deadlock
- Performance
  - Method invokes toString() method on a String
- Security
  - Hardcoded constant database password
- Dodgy
  - Useless assignment in return statement

Example: Bug found by static checking (1)

```
public class Main {
   public static void chk(boolean s1, boolean s2){
    if(s1 = s2) {System.out.println("foo");}
    else {System.out.println("bar");}}
   public static void main(String[] args) {
    boolean b1 = false;
    boolean b2 = true;
    Main.chk(b1, b2);}}
```

'=' instead of '=='

JLint:

Verification completed: 0 reported messages.

FindBugs:

The parameter s1 to Main.chk(boolean, boolean) is dead upon entry but overwritten

Dead store to s1 in Main.chk(boolean, boolean)

PMD:

No problems found

Example: Bug found by static checking (2)

```
public static void main(String[] args) {
   String b = "bob";
   b.replace('b', 'p');
   if(b.equals("pop")){
    System.out.println("Equals");
   }
}
```

The function String.replace() (called as a member function of an instance) does not alter the concrete instance, but returns the modified string as its return value

JLint:

java\lang\String.java:1: equals() was overridden but not hashCode().

Verification completed: 1 reported messages.

FindBugs:

Main.main(String[]) ignores return value of String.replace(char, char)

PMD:

An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itself

Example: Extension of PMD rules

```
class Example {
   void bar() {
       while (baz)
              buz.doSomething();
   }
}
```

We would like to detect when there aren't curly braces around the body statement of a "while" loop

```
public class WhileLoopsMustUseBracesRule extends AbstractRule {
  public Object visit(ASTWhileStatement node, Object data) {
    SimpleNode firstStmt = (SimpleNode)node.jjtGetChild(1);
    if (!hasBlockAsFirstChild(firstStmt)) {
       addViolation(data, node);
    }
    return super.visit(node,data);
  }
  private boolean hasBlockAsFirstChild(SimpleNode node) {
    return (node.jjtGetNumChildren() != 0 && (node.jjtGetChild(0)
   instanceof ASTBlock));
  }
}
```

The checker rule (in Java)

- Abstract Syntax Tree (AST) based representation of the source code
- Rule to be checked at a given place of the AST

How to use static analysis tools

- Integrate to build process
  - Perform check before/after each commit, generate reports
  - Use from the start of a project: Too many problems found at a later phase would discourage developers
- Configure the tools
  - Filter based on severity or category of rules
  - Add custom rules
- Review the results
  - False positive: No errors found does not mean correct software
  - False negative: An error found may not cause a real failure
  - Ignore rule / one occurrence, with explanation

Checking runtime failures by code interpretation

Dynamic properties to be checked

- Goal: Detection of runtime failures without executing the software
- Failures to be detected include
  - Null pointer
  - Array index out-of-bound
  - Uninitialized data
  - Access conflict on shared variable
  - Arithmetic error: division by zero, overflow, underflow
  - Dangerous type conversion
  - Dead code (unreachable)
- Performed by control flow and data flow analysis
  - Calculate interval (range) for each variable
  - Propagate intervals based on control flow

Example: Detecting a runtime error by static analysis

```
20: int ar[10];
21: int i,j;
22: for (i=0; i<10; i++)
23: {
24:   for (j=0; j<10; j++)
25:   {
26:     ar[i-j] = i+j;
27:   }
28: }
```

Error: Out-of-bound array access in line 26

Example: The Infer tool

- Static analysis tool by Facebook
  - Focus on mobile code development
  - Users: Facebook, Instagram, Oculus, Spotify, WhatsApp, …
- Android and Java
  - Null pointers, resource leaks
- iOS and Objective-C
  - Null pointers, memory leaks, resource leaks

```
Found 3 issues

./Root/Hello.java:27: error: NULL_DEREFERENCE
  object a last assigned on line 25 could be null and is dereferenced at line 27
  25.         Pointers.A a = Pointers.mayReturnNull(rng.nextInt());
  26.         // FIXME: should check for null before calling method()
  27. >       a.method();
  28.     }
```

Example: QA-C, QA-C++ tools

Security Issues:

- Buffer under- and overflow
- Arithmetic overflow and wraparound
- Format string mis-use

Crash-Inducing Defects:

- Null pointer operations, invalid pointer values, operations on unrelated pointers
- Divide-by-zero
- Uncaught exceptions, throw-catch specification mismatches, improper exception use

Flawed Logic Issues:

- Invariant (always true/false) logic and unreachable code
- Unset variables
- Redundant expressions, initializations and assignments
- Infinite loops
- Return value mismatches

Memory Issues:

- Memory allocation mismatches
- Memory leaks

API Mis-use:

- Standard library pre- and post-condition verification

„A combination of SMT solver and in-house language and parsing expertise result in exceptionally accurate dataflow and semantic modeling of C and C++ code – a foundation for a set of unique analysis checks.”

How does code interpretation work?

Source code to be examined:

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

k = k/(i-j);
```

Risk: Division by 0.
Is it possible?
What is the input resulting in division by 0?

Phase 1: Collecting information about the values of variables

- $X0=\{(0,0,k) \mid k \in [-2^{31}, 2^{31}-1]\}$
- $X1=\{(2,j,k) \mid (i,j,k) \in X0\}$
- $X2=\{(i,k+5,k) \mid (i,j,k) \in X1\}$
- $X3= X2 \cup X6$
- $X4=\{(i+1,j,k) \mid (i,j,k) \in X3,\ i<10\}$
- $X5=\{(i,j+3,k) \mid (i,j,k) \in X4\}$
- $X6= X5$
- $X7=\{(i,j,k) \mid (i,j,k) \in X3,\ i=10\}$
- $X8=\{(i,j,k/(i-j)) \mid (i,j,k) \in X7,\ i-j \neq 0\}$

What are the potential values of (i,j,k)

Based on the previous step

This statement can be reached from two places

Inside of the loop

Exit from the loop

Phase 2: Computation of the ranges (1)

- $X0=\{(0,0,k) \mid k\in[-2^{31},2^{31}-1]\}$

  $X0=\{(0,0,k) \mid k\in[-2^{31},2^{31}-1]\}$

- $X1=\{(2,j,k) \mid (i,j,k)\in X0\}$

  $X1=\{(2,0,k) \mid k\in[-2^{31}, 2^{31}-1]\}$

- $X2=\{(i,k+5,k) \mid (i,j,k)\in X1\}$

  $X2=\{(2,k+5,k) \mid k\in[-2^{31}, 2^{31}-1]\}$

- $X3= X2 \cup X6$

  $X3=\{(i,j,k) \mid k\in[-2^{31}, 2^{31}-1], i\in[2,10], j=k+3i-1\}$

- $X4=\{(i+1,j,k) \mid (i,j,k)\in X3, i<10\}$

  $X4=\{(i,j,k) \mid k\in[-2^{31}, 2^{31}-1], i\in[3,10), j=k+3i-4\}$

Ranges calculated using the information collected in the previous phase

Resolving references by propagating information from X0

Assignment before the loop, and condition to be in the loop

Loop invariant: $j=k+5+3(i-2)$

i increased; j was not assigned its new value thus 3 subtracted

Phase 2: Computation of the ranges (2)

- $X5=\{(i,j+3,k) \mid (i,j,k) \in X4\}$

  $X5=\{(i,j,k) \mid k \in [-2^{31}, 2^{31}-1], i \in [3,10), j=k+3i-1\}$

- $X6= X5$

  $X6=X5$

- $X7=\{(i,j,k) \mid (i,j,k) \in X3, i=10\}$

  $X7=\{(10,j,k) \mid k \in [-2^{31}, 2^{31}-1], j=k+29\}$

  j=k+5+3(i-2),
  and here i=10

- $X8=\{(i,j,k/(i-j)) \mid (i,j,k) \in X7\}$

  $X8=\{(10,j,k/(i-j)) \mid k \in [-2^{31}, 2^{31}-1], j=k+29\}$

Error, if i-j=0, in this case since i=j=10, k=j-29=-19

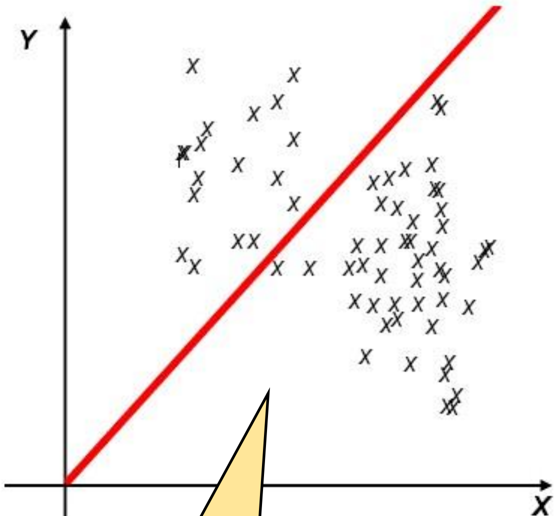
$X8_{error}=\{(10,10,-19)\}$

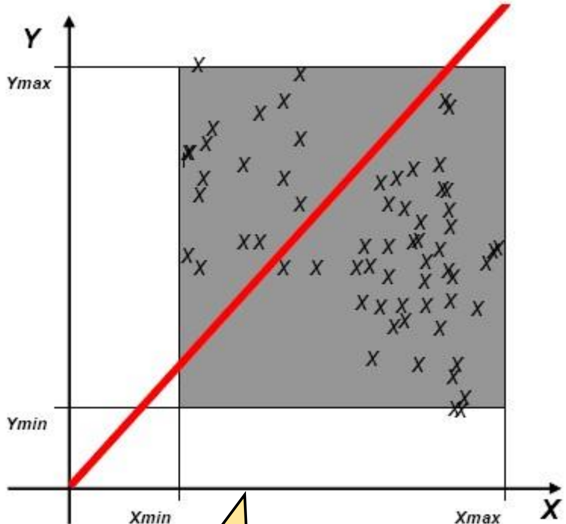
Analyzing dynamic properties

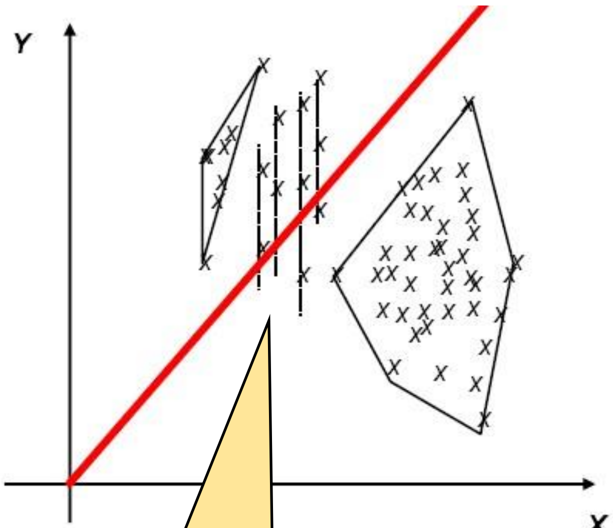
- Based on analyzing control flow and data flow
  - Operations with intervals and constraints
  - Loops: determine loop invariants
- Calculating loop invariants
  - Hard problem (undecidable in general)
  - Approximations are required
- Abstraction: over-approximating the intervals
  - All errors are detected
  - False negatives (errors) are possible
    - Can be treated as a hint for further analysis or testing

Illustration of abstraction

- Problem: Division by (x-y); is x==y possible?







Possible values of x and y (without abstraction)

Rough abstraction by intervals: many false positives

Better abstraction (regions): 4 cases shall be checked

Example: Color-coded output of the PolySpace tool

```c
static void Square_Root_conv (double alpha, float *beta_pt, float *gamma)
{
  *beta_pt = (float)((1.5 + cos(alpha))/5.0);

  if(*beta_pt < 0.3)
  *gamma = 0.75;
}

static void Square_Root (void)
{
  double alpha = random_float();
  float beta;
  float gamma;

  Square_Root_conv (alpha, &beta, &gamma);

  if(random_int() > 0){
     gamma = (float)sqrt(beta - 0.75);
  }
  else{
     gamma = (float)sqrt(gamma - beta);
     if(beta > 1)
            alpha = 0;
  }
}
```

The Colors of PolySpace

Each function and operation is verified for **all** possible values, and then colored according to its reliability.

Green **Proven** safe under **all** operating conditions. Focus your efforts elsewhere.

Red **Proven** definite error each time the operation is executed.

Orange **Unproven.**

Grey **Proven** unreachable code. May point to a functional issue.

Tools supporting code interpretation

- Abstract interpretation of code:
  - PolySpace C/Ada
    - Ariane 5 (70k lines of code), Flight Management System (500k lines of code)
  - Astrée
    - Airbus flight control software
  - C Global Surveyor
    - NASA Mars PathFinder, Deep Space One
- Annotation based tools (design by contract):
  Loop invariants, pre- and post-conditions are given manually
  - ESC/Java (based on JML):
    Annotation based synthesis of monitor components, test oracle
    - E.g., jmlc+jmlrac, jmlunit
  - Microsoft PreFix, PreFast, Boogie (Spec#, BoogiePL):
    Verification conditions (theorems to be proved) are generated and given to an external theorem prover

Summary: Techniques for source code analysis

- Manual review on the basis of checklists
  - Coding guidelines (e.g., naming conventions)
  - Typical mistakes (error guessing)
  - Analysis of the structure
    - Control flow checking: complexity, clear structure
    - Data flow analysis: looking for limits and boundary values
- Static analysis tools
  - Checking coding standards (built-in rules)
  - Checking the limits of source code metrics
  - Looking for fault patterns: Syntactic and possibly semantic faults
- Dynamic analysis tools
  - Checking potential runtime faults by code interpretation
  - Calculate and propagate the interval for each variable
  - Performance problems may also be detected