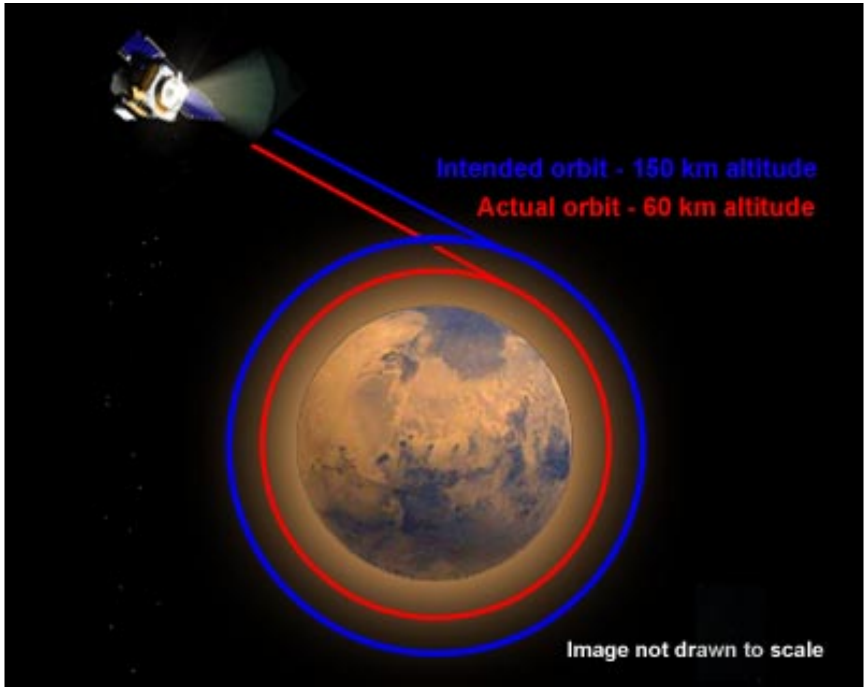
Software Test and Analysis

Luciano Baresi
Politecnico di Milano

Leonardo Mariani (University of Milano Bicocca)

Motivations

- Software systems permeate (almost) every aspect of our life
  - Software is buggy
  - In 2002 the costs related to software errors are estimated in 60 Billion USD



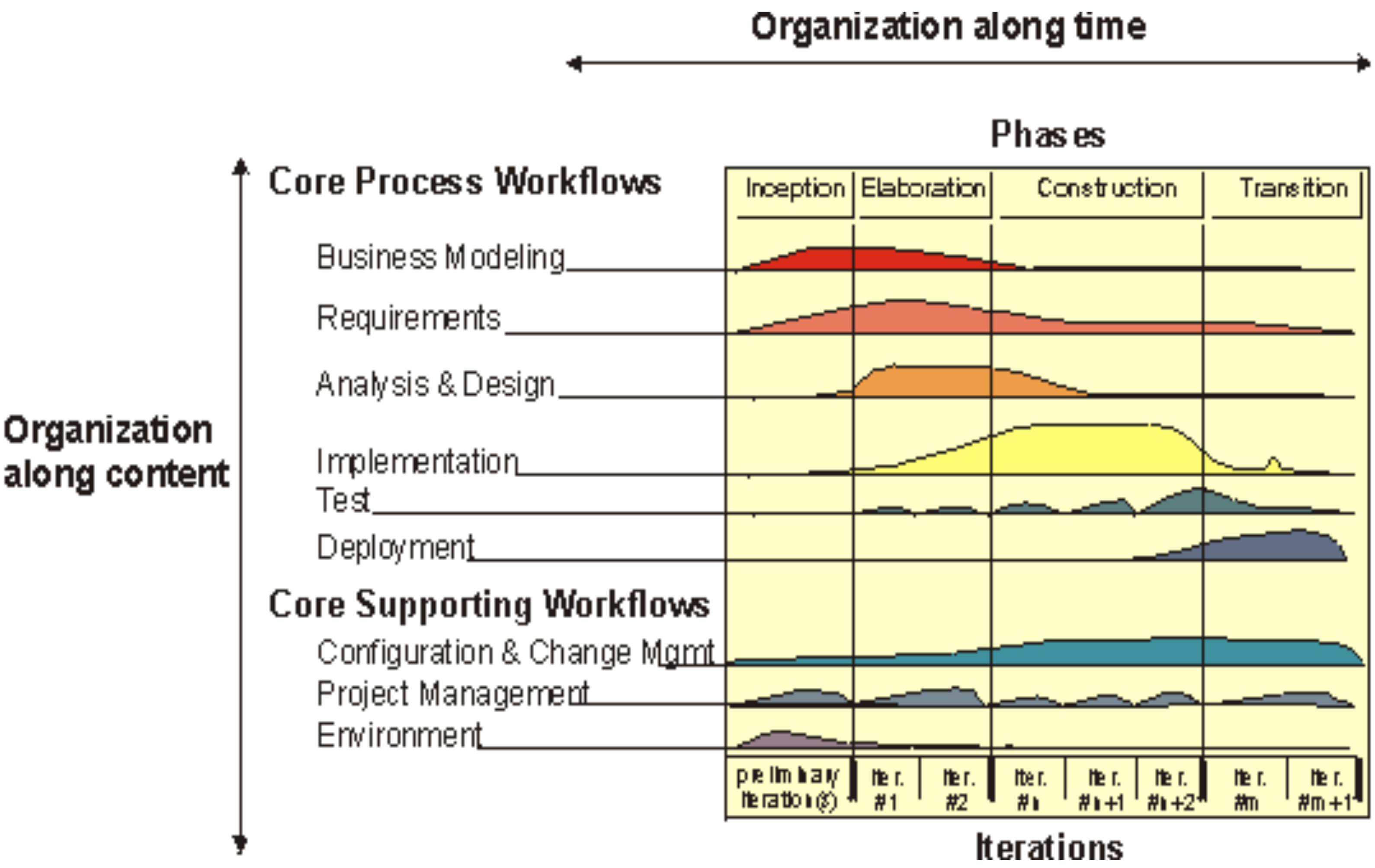
1999: NASA Mars Climate Orbiter ($125 million)

2011: iPhone Alarm

Dijkstra (1972)

- Program testing can be used to show the presence of bugs, but never to show their absence

RUP: Rational Unified Process



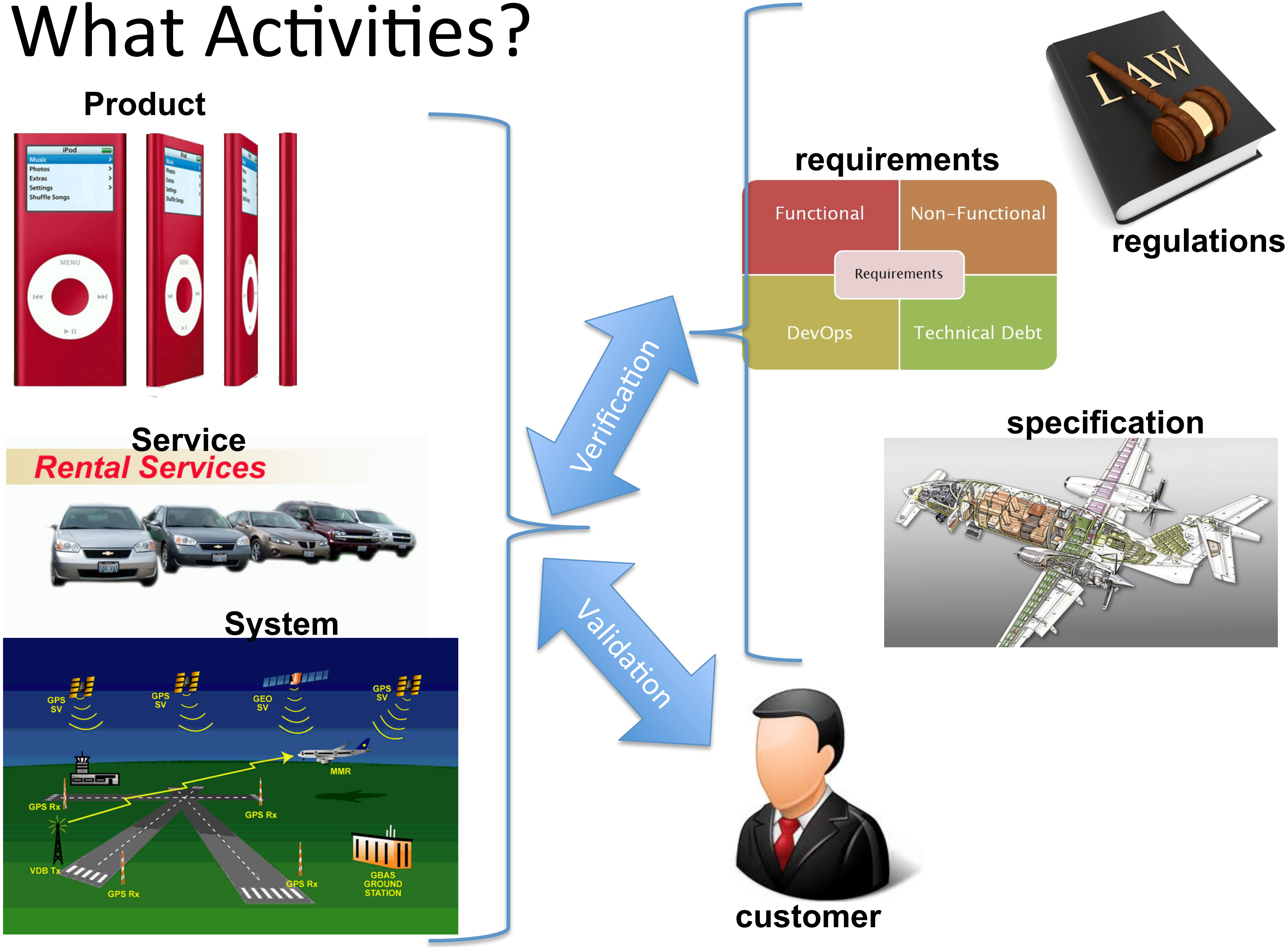
Quality

- Process Qualities
- Product Qualities
  - Internal qualities (maintainability, …)
  - External qualities
    - Performance
    - Usability
    - Correctness
    - Portability
    - …

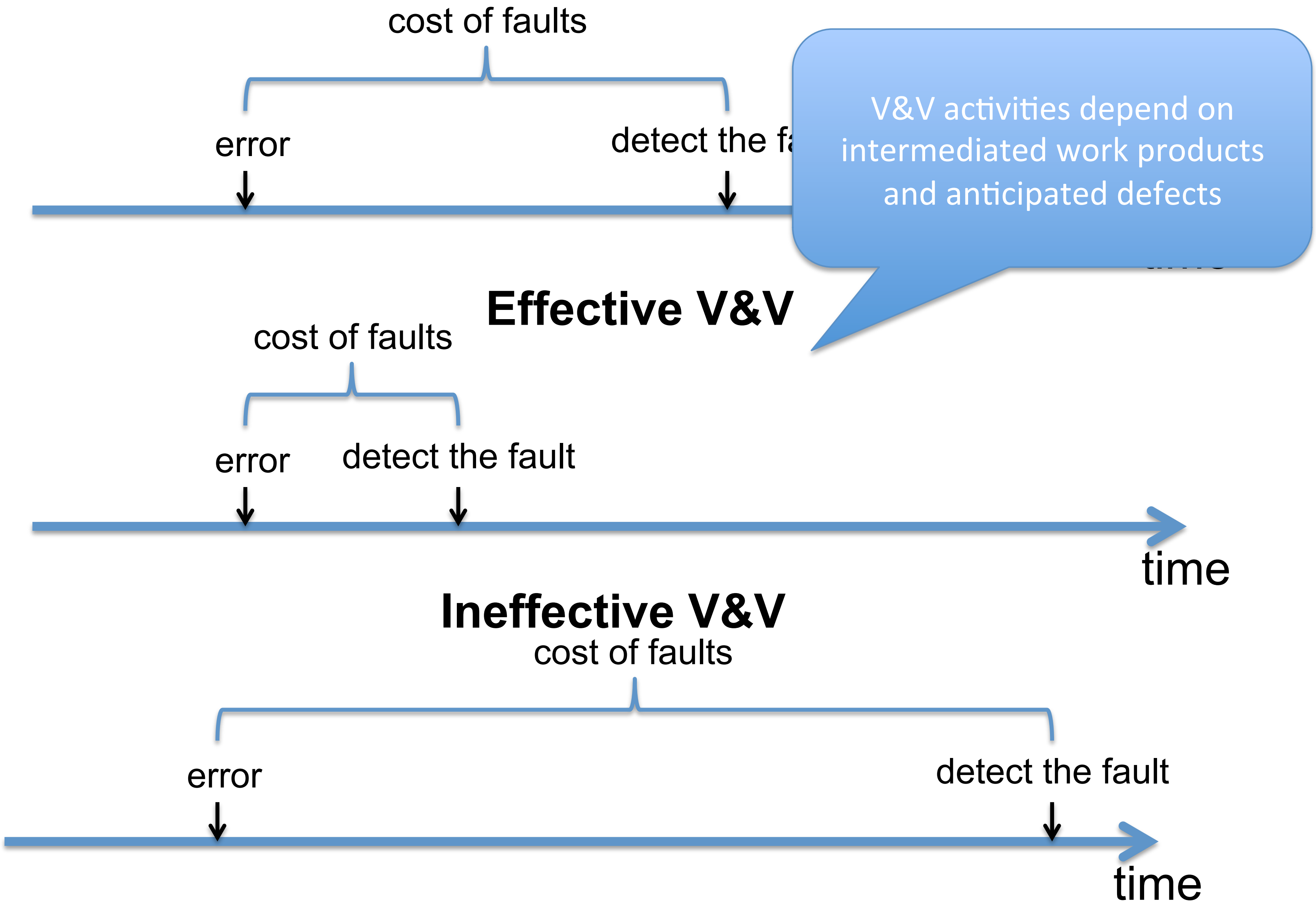
Quality Process

- activities + responsibilities
  - focused primarily on ensuring adequate quality
  - concerned with project schedule
  - integral part of the development process

What Activities?



Key Principle of Quality Planning



Verification Steps for Intermediate Artifacts

- Internal consistency checks
  - compliance with structuring rules that define "well-formed" artifacts of that type
  - prevent and/or ease detection of common errors
  - e.g., compliance to MISRA rules
- External consistency checks
  - consistency with related artifacts
  - often conformance to a specification
  - e.g., conformance to Example SRS v2.0

Strategies vs Plans

	Strategy	*Plan*
Scope	Organization	Project
Structure and content based on	Organization structure, experience and policy over several projects	Sta[illegible]rd[illegible]ucture pres[illegible] in strate[illegible]
Evolves	Slowly, with organization and policy changes	Quic[illegible]apting to pro[illegible]ds

Test & Analysis Strategy

Test and Analysis Strategy

- Lessons of past experience
  - an organizational asset built and refined over time
- Body of explicit knowledge
  - more valuable than islands of individual competence
  - amenable to improvement
  - reduces vulnerability to organizational change (e.g., loss of key individuals)
- Essential for
  - avoiding recurring errors
  - maintaining consistency of the process
  - increasing development efficiency

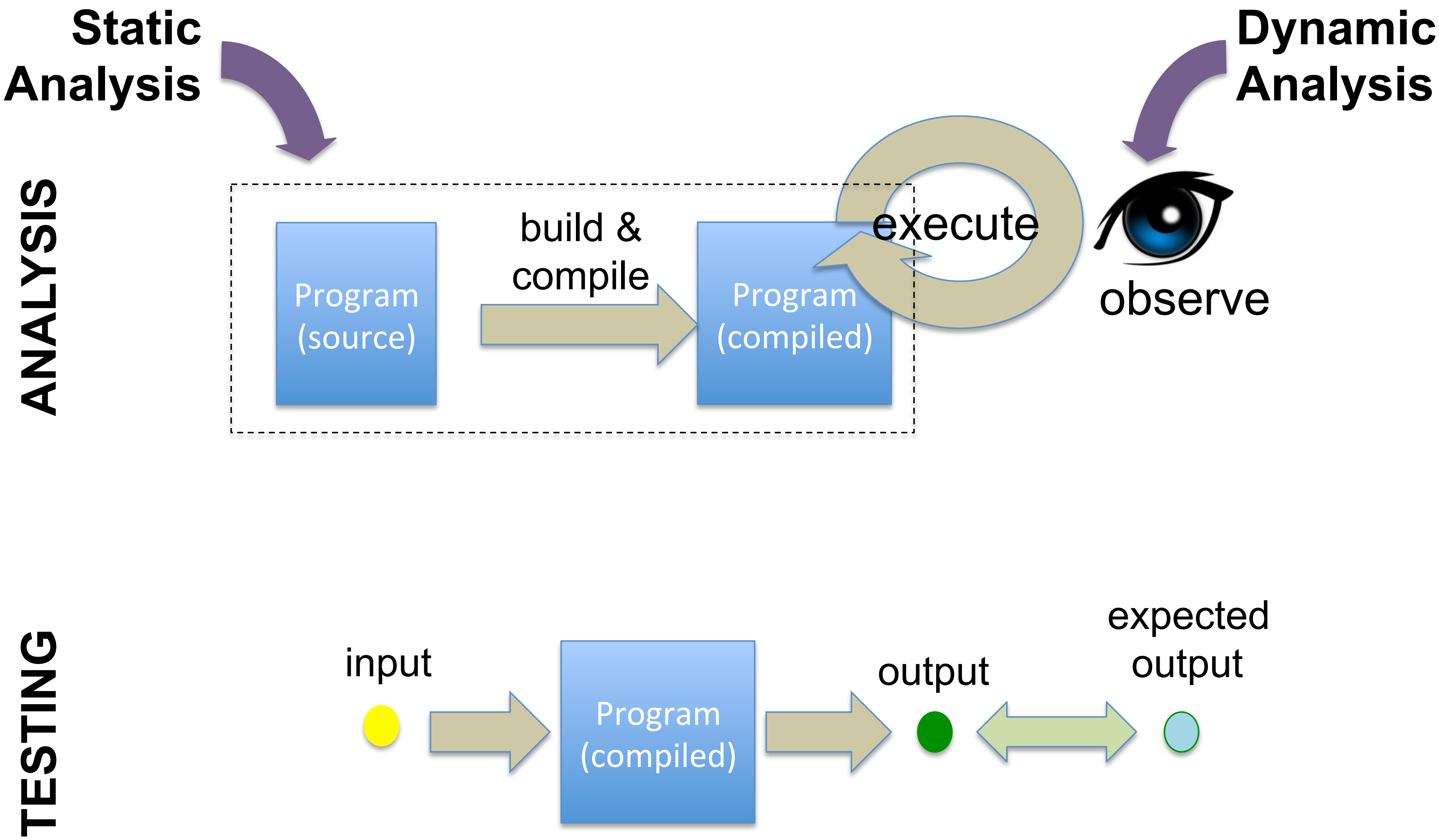
Elements of a Strategy

- Common quality requirements that apply to all or most products
  - unambiguous definition and measures
- Set of documents normally produced during the quality process
  - contents and relationships
- Activities prescribed by the overall process
  - standard tools and practices
- Guidelines for project staffing and assignment of roles and responsibilities

What You Will See Today

Some practices and tools that you might want to consider as part of your testing and analysis strategy

Testing and Analysis



- Why Static Analysis?
  - corner cases hard to execute
    - `if ((currentHour>23) && (isLeapYear))`
`{…do something terribly wrong…}`
  - prevention
    - check if variables are always initialized before use
- Why Dynamic Analysis?
  - Easy to execute but hard to fail bugs
    - Memory leak: allocate memory without freeing it
- Why Testing?
  - Main approach to check correctness
  - Most intuitive way to compare the behavior of a program wrt an expectation

Our Plan

- Program Analysis
  - Static Analysis
    - cppCheck, metriculator
  - Dynamic Analysis
    - Valgrind
- Testing
  - Unit testing
    - Boost unit tests
  - Mocking
    - G(oogle)Mock
  - Coverage
    - gcov
  - *Functional*

Why Program Analysis?

- Exhaustively check properties that are difficult to test
  - Faults that cause failures
    - rarely
    - under conditions difficult to control

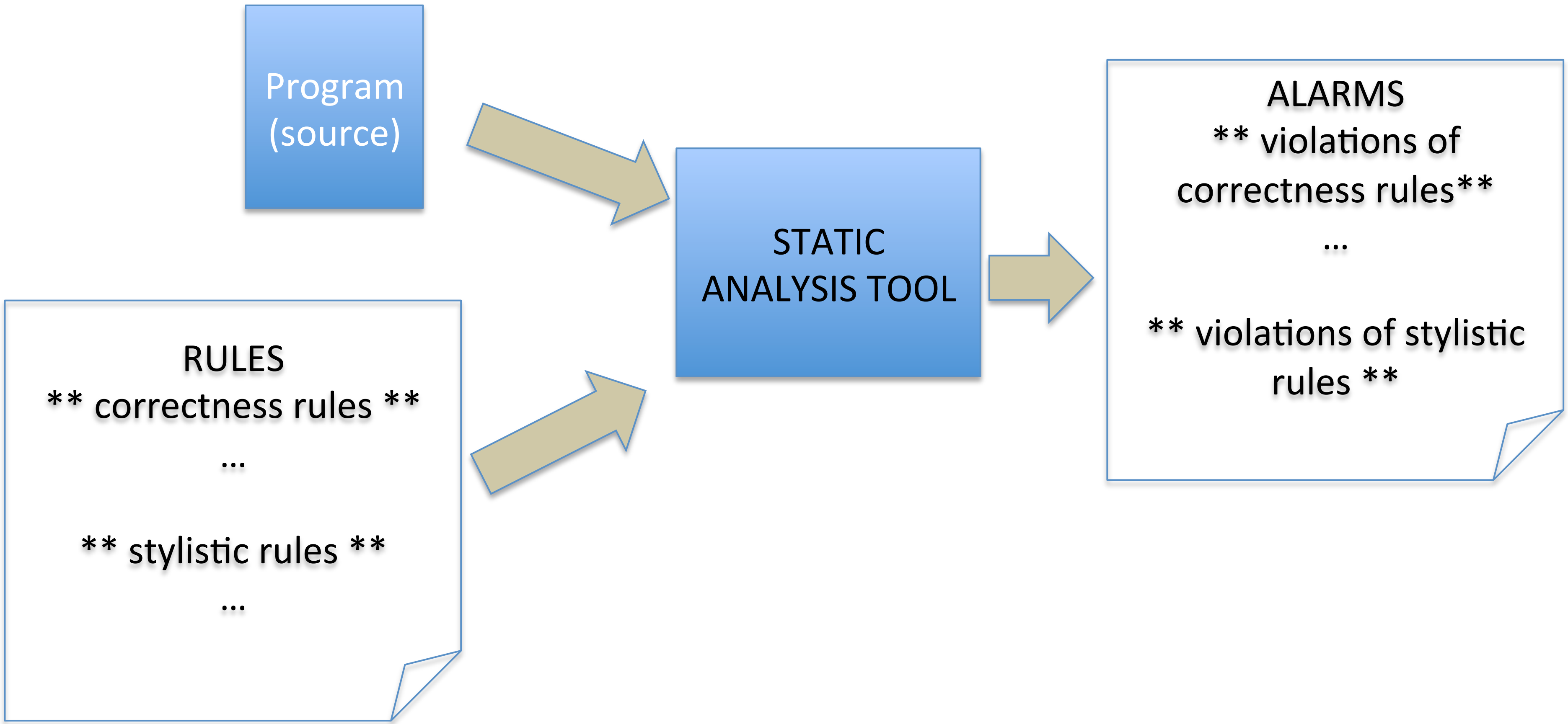
Why Automated Analysis?

- Manual program inspection effective in finding faults difficult to detect with testing
- But humans are not good at
  - repetitive and tedious tasks
  - maintaining large amounts of detail
- Automated analysis replace human inspection for some classes of faults

Static vs dynamic analysis

- Static analysis
  - examine program source code
    - examine the complete execution space
    - but may lead to false alarms
- Dynamic analysis
  - examine program execution traces
    - no infeasible path problem
    - but cannot examine the execution space exhaustively

Rule-Based Static Analysis
(of source code)



In some domains the code must comply to a standard set of rules
e.g., MISRA in the automotive domain

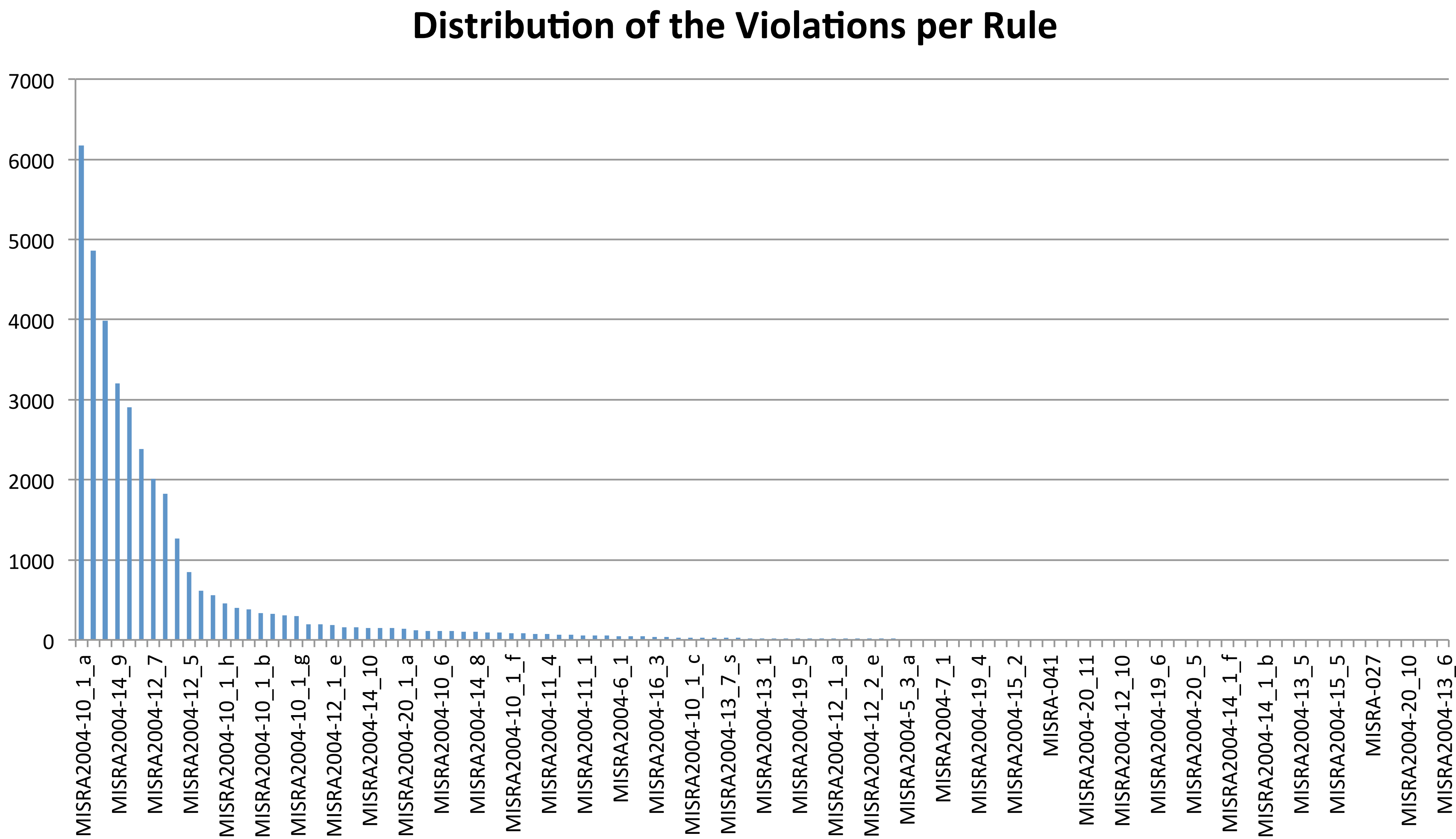
Example

- cppCheck
  - open source static analysis tool for C/C++
- Poco C++ Library
  - Library for building C++ network-applications

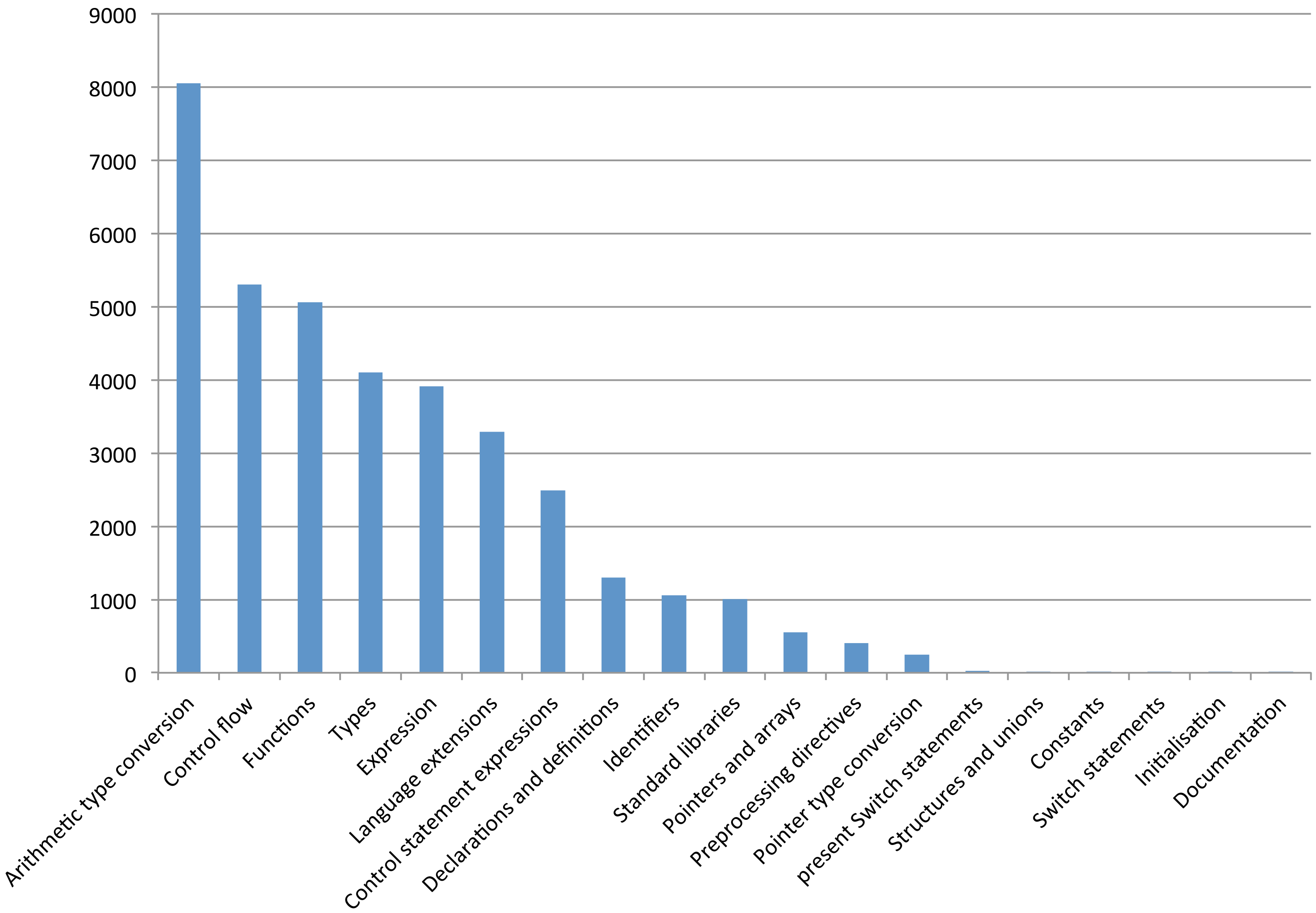
An Experience from a Real Case: Checking MISRA Rules

214 rules dedicated to development of better and more reliable automotive software

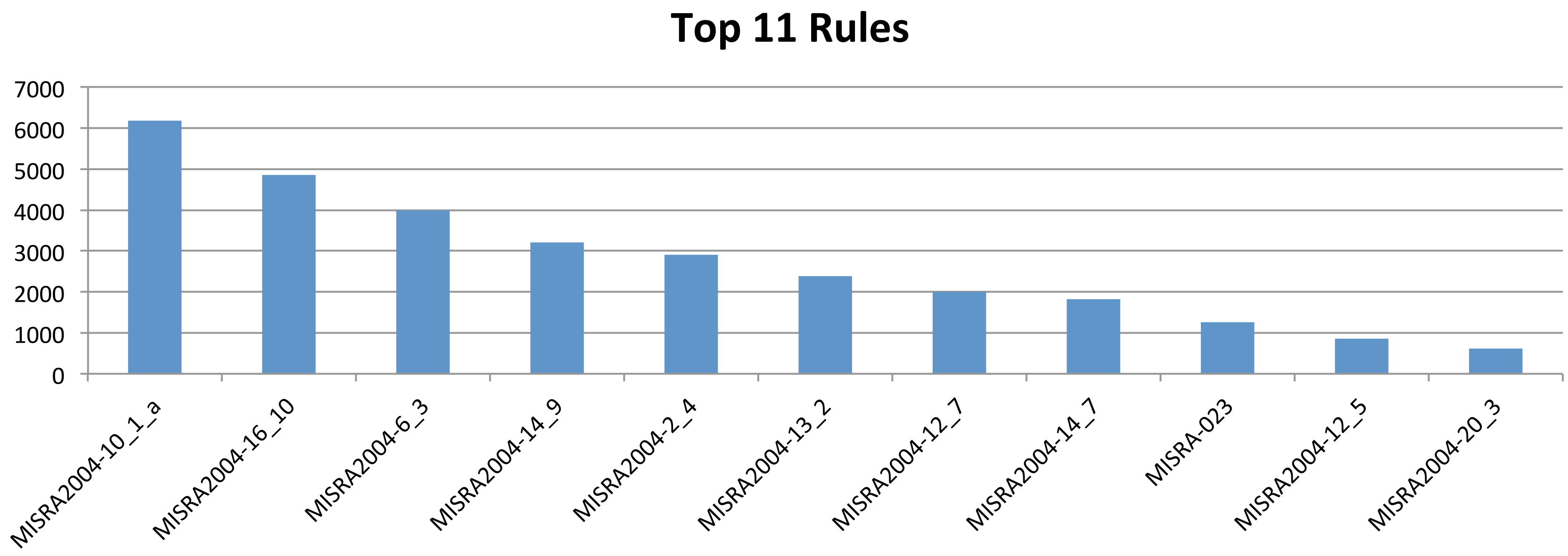
36.850 rule violations

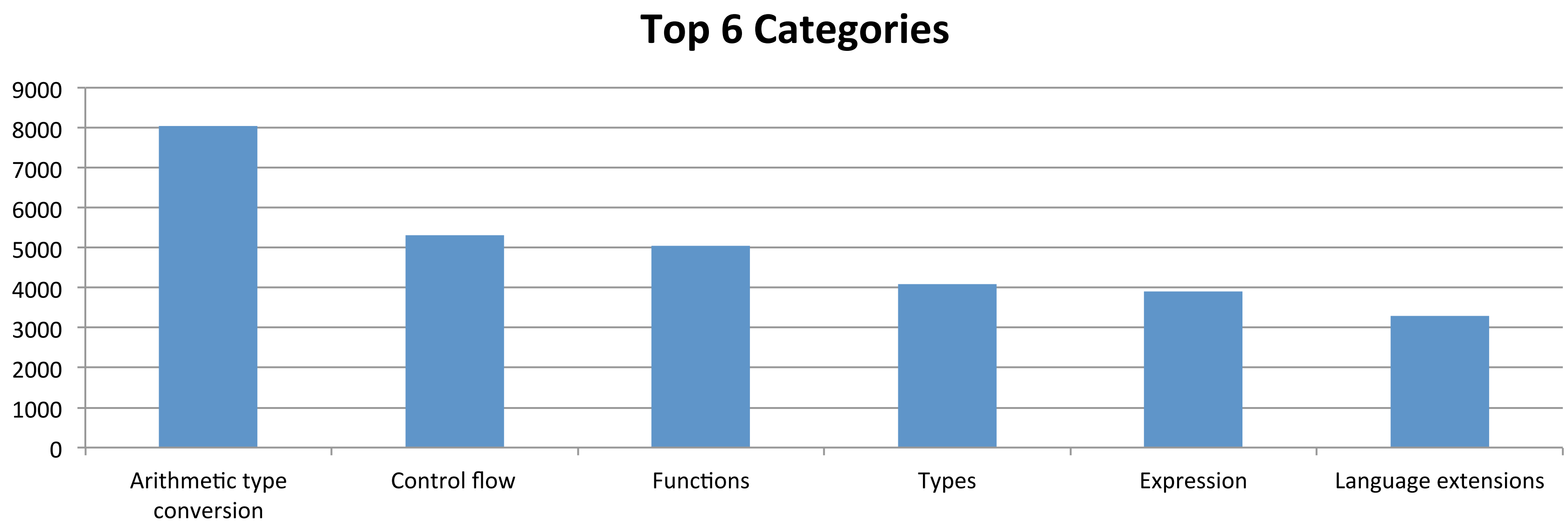


Distribution by category



Pareto Analysis





Top 11 Rules

MISRA2004-10_1_a	Arithmetic type conversion
Avoid implicit conversions between signed and unsigned integer types	
MISRA2004-16_10	Functions
If a function returns error information, then that error information shall be tested	
MISRA2004-6_3	Types
typedefs that indicate size and signedness should be used in place of the basic types	
MISRA2004-14_9	Control Flow
if' and 'else' should be followed by a compound statement	
MISRA2004-2_4	Language Extensions
Sections of code should not be commented out	
MISRA2004-13_2	Control Statement Extensions
Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	

Top 11 Rules

MISRA2004-12_7	Expressions
Bitwise operators shall not be applied to operands whose underlying type is signed	
MISRA2004-14_7	Control Flow
A function shall have a single point of exit at the end of the function	
MISRA2004-23	Declarations and definitions
Make declarations at file scope static where possible	
MISRA2004-12_5	Expressions
The operands of a logical && or || shall be primary-expressions	
MISRA2004-20_3	Standard Libraries
The validity of values passed to library functions shall be checked	

Complexity Metrics (static analysis)

- Code Complexity = how hard is to maintain, test, debug, … the software
- Thus do no write complex code!

How to Measure Complexity?

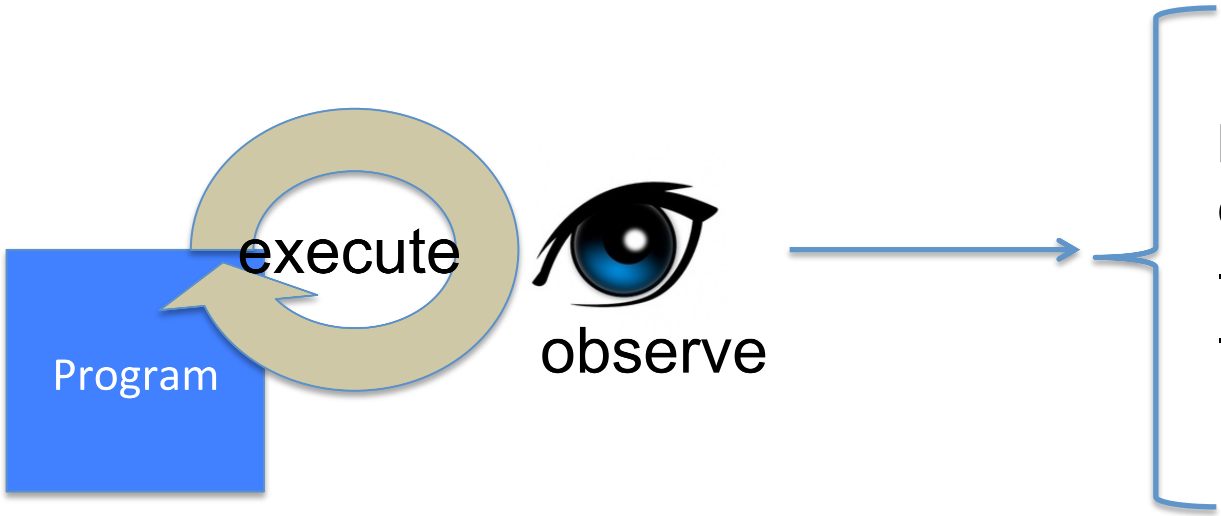
Code Complexity

- No single measure
  - Cyclomatic complexity = complexity of decisions in a function
    - CC < 10 from McCabe
  - LOCs = number of lines of code in a function
    - Loc < 200 from the literature
  - MaxDepth = the nesting level of code blocks in a function
    - MD < 5 from the literature

Example

- Metriculator
  - Free open source metrics calculator
  - LSLOC – logical source lines of code
  - McCabe – cyclomatic complexity
  - NbParams – number of parameters in a function/method
  - EfferentCoupling – number of types references from a class
  - NbMembers – number of attributes in a class (recursive)

Dynamic Analysis



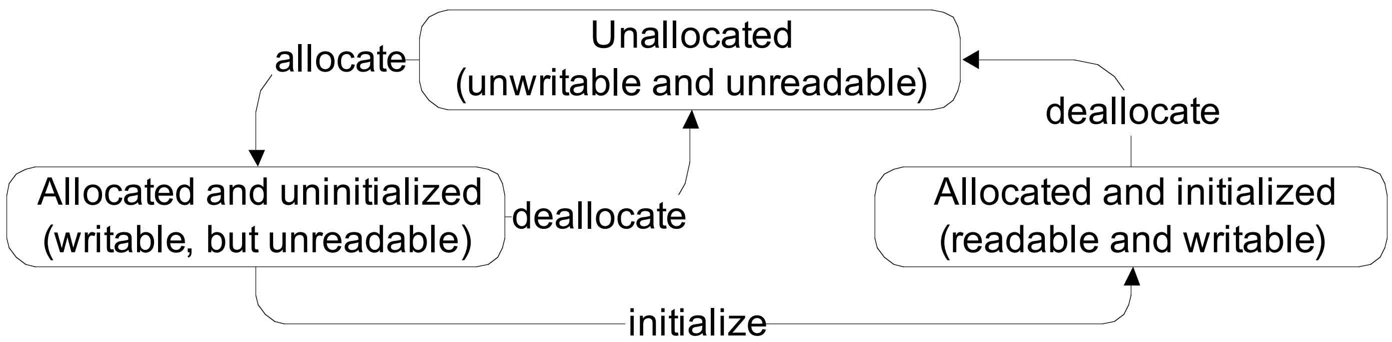
Easy to execute but hard to fail cases:

- Memory Leak
- Data races

…

(Dynamic) Memory Analysis

- Instrument program to trace memory access
  - record the state of each memory location
  - detect accesses incompatible with the current state
    - attempts to access unallocated memory
    - read from uninitialized memory locations
  - array bounds violations:
    - add memory locations with state unallocated before and after each array
    - attempts to access these locations are detected immediately



Data Race

- Serious problem in highly concurrent software

Dynamic Lockset Analysis

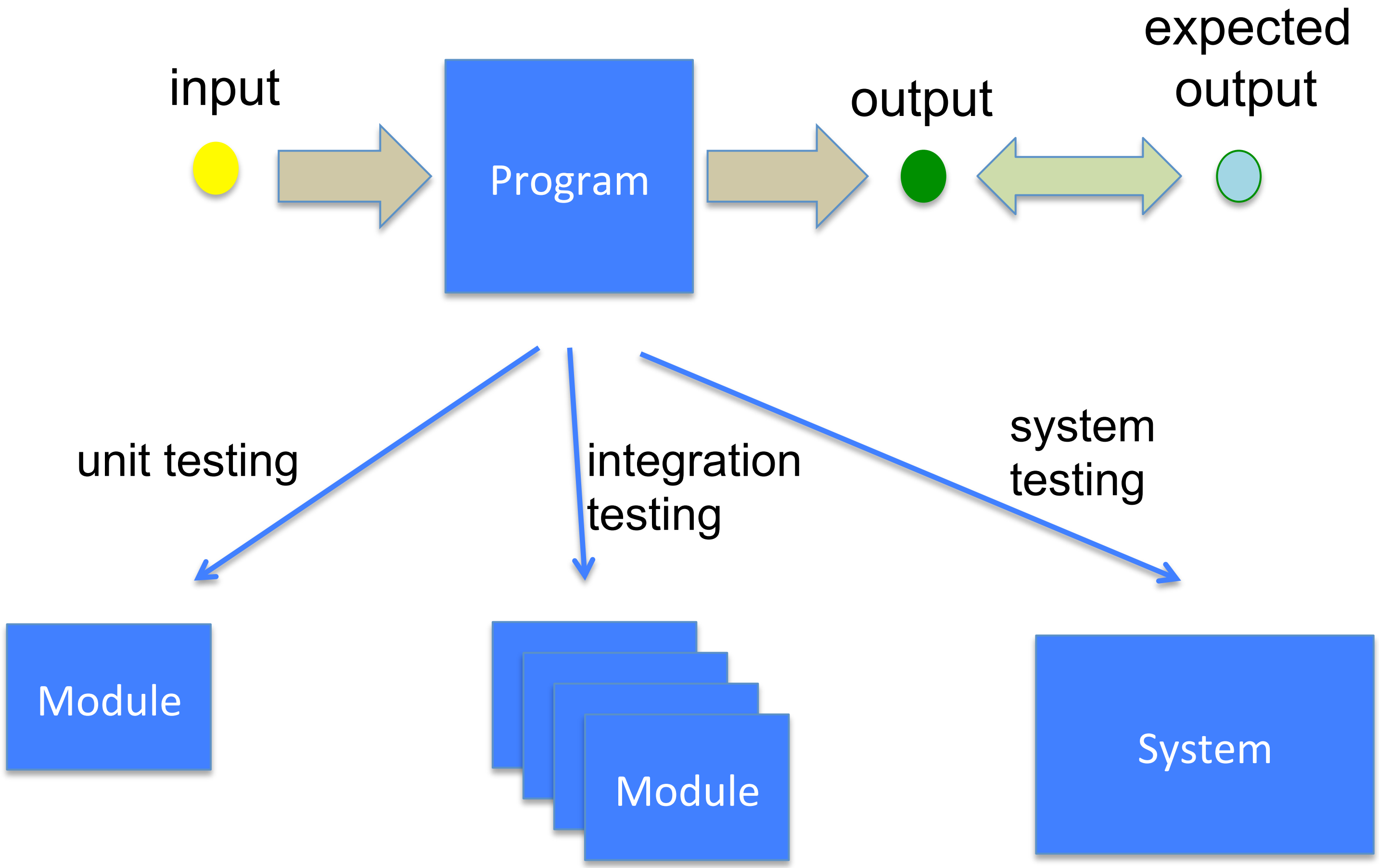
- Lockset discipline: set of rules to prevent data races
  - Every variable shared between threads must be protected by a mutual exclusion lock
- Dynamic lockset analysis detects violation of the locking discipline
  - Identify set of mutual exclusion locks held by threads when accessing each shared variable
  - INIT: each shared variable is associated with all available locks
  - RUN: thread accesses a shared variable
    - intersect current set of candidate locks with locks held by the thread
  - END: set of locks after executing a test = set of locks always held by threads accessing that variable
    - empty set for v = no lock consistently protects v

Simple lockset analysis: example

Thread	Program trace	Locks held	Lockset(x)	
		{}	{lck1, lck2}	INIT:all locks for x
thread A	lock(lck1)			
		{lck1}		lck1 held
	x=x+1			
			{lck1}	Intersect with locks held
	unlock(lck1}			
		{}		
tread B	lock{lck2}			
		{lck2}		lck2 held
	x=x+1			
			{}	Empty intersection potential race

Testing

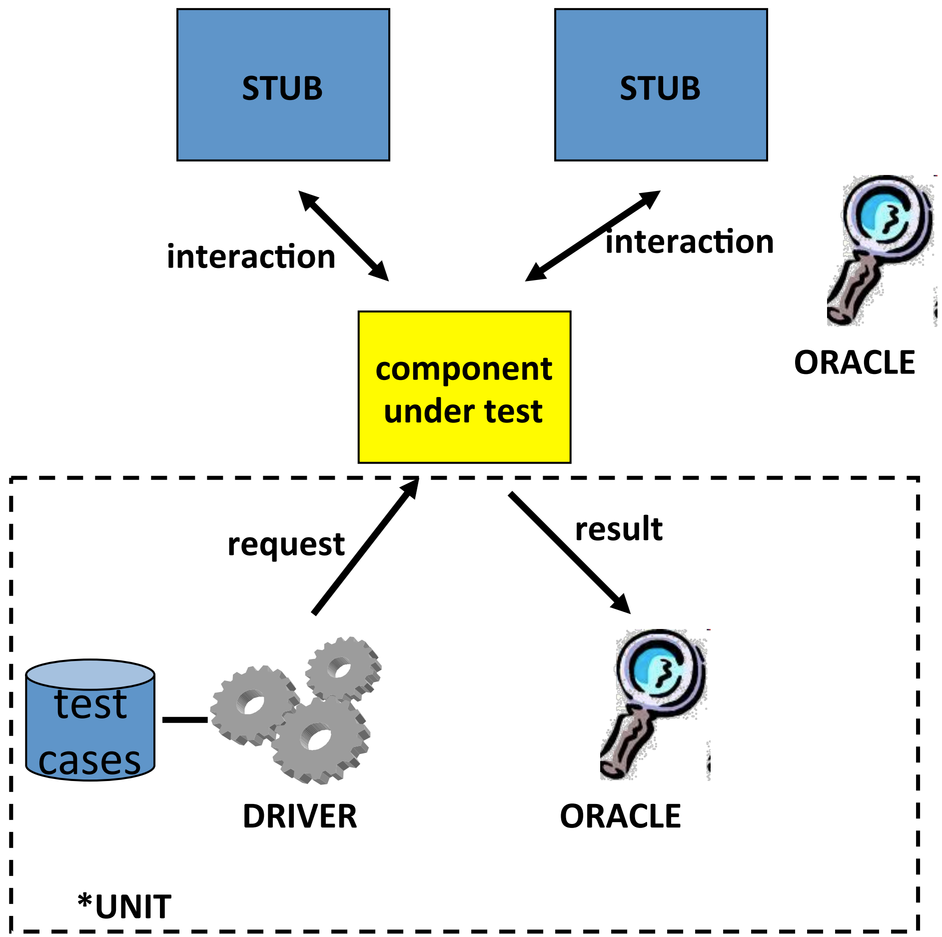
Testing Levels



Test Case Implementation

- To automate testing we need
  - driver
  - stubs
  - oracles

- *Unit (e.g., Gunit, Boot unit testing, QTUnit): framework that supports development of
  - drivers and
  - Oracles



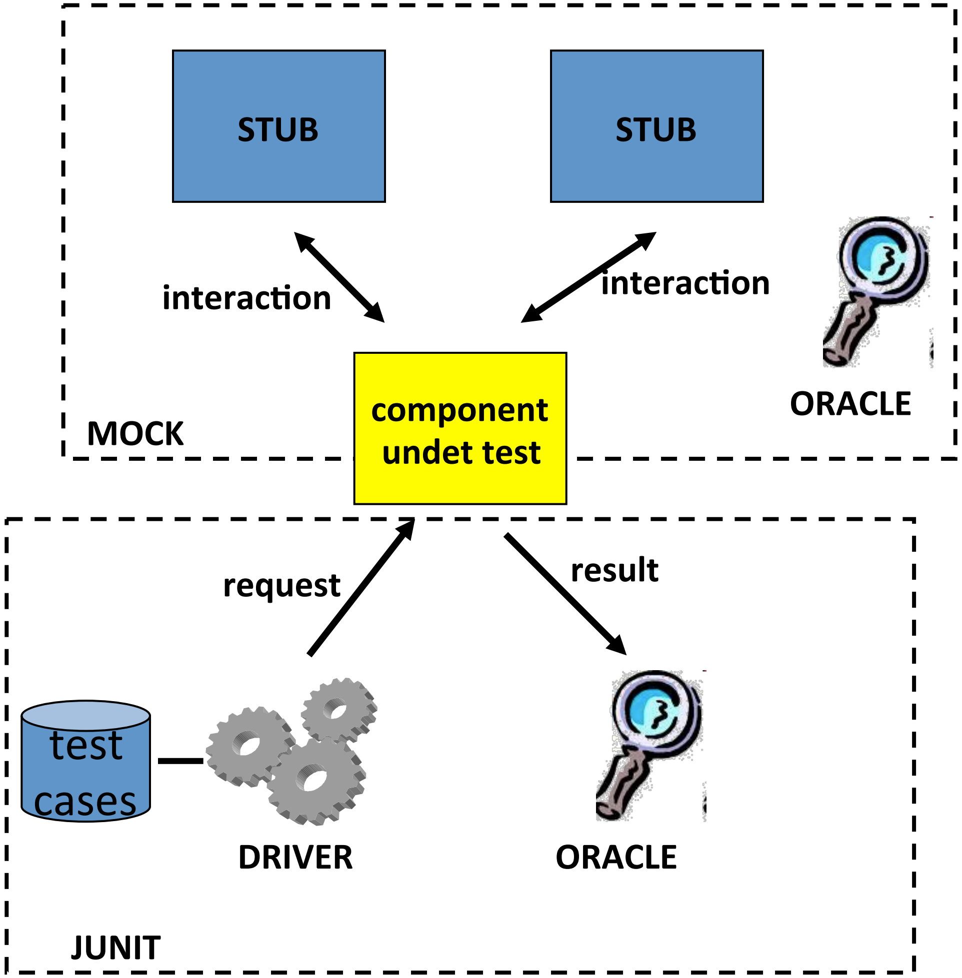
A Sample BOOST Test Case

```
int add( int i, int j ) { return i + j; }

BOOST_AUTO_TEST_CASE( my_test )
{
 // seven ways to detect and report the same error:
BOOST_CHECK( add( 2,2 ) == 4 );                          // #1 continues on error
BOOST_REQUIRE( add( 2,2 ) == 4 );                        // #2 throws on error
if( add( 2,2 ) != 4 )
       BOOST_ERROR( "Ouch..." );                         // #3 continues on error
if( add( 2,2 ) != 4 )
       BOOST_FAIL( "Ouch..." );                          // #4 throws on error
if( add( 2,2 ) != 4 ) throw "Ouch...";                   // #5 throws on error
BOOST_CHECK_MESSAGE( add( 2,2 ) == 4,                    // #6 continues on error
                  "add(..) result: " << add( 2,2 ) );
BOOST_CHECK_EQUAL( add( 2,2 ), 4 );                      // #7 continues on error
}
```

Stub

- *Unit does not support stubs
  - testers must manually develop them
  - create stubs that provide different results to different test cases may be complex and time-consuming
  - faulty stubs reduce productivity and quality of your testing
- *Unit allows to specify conditions on values returned from the object under test, but does not allow to specify the expected interactions
  - e.g., we want to verify that a ShoppingCart removes 2 items from a warehouse when a cart with 2 items is purchased (note that you do not have the warehouse)



Regression

- Yesterday it worked, today it doesn’t
  - I was fixing X, and accidentally broke Y
  - That bug was fixed, but now it’s back
- Tests must be re-run after any change
  - Adding new features
  - Changing, adapting software to new conditions
  - Fixing other bugs
- Regression testing can be a major cost of software maintenance
  - Sometimes much more than making the change

Basic Problems of Regression Test

- Maintaining test suite
  - If I change feature X, how many test cases must be revised because they use feature X?
  - Which test cases should be removed or replaced? Which test cases should be added?
- Cost of re-testing
  - Often proportional to product size, not change size
  - Big problem if testing requires manual effort
    - Possible problem even for automated testing, when the test suite and test execution time grows beyond a few hours

The Oracle Problem

- It is not always possible to predict the result of a test
- E.g., what is the expected result of an
    - HPC system that simulates and plan delivery of millions of items for FedEx?
    - HPC system that processes billion of transactions for NASDAQ stock exchange?
    - HPC Graphic technology used at Dreamworks?
    - HPC fluid dynamics simulations carried on at Whirpool?

Weak Oracles

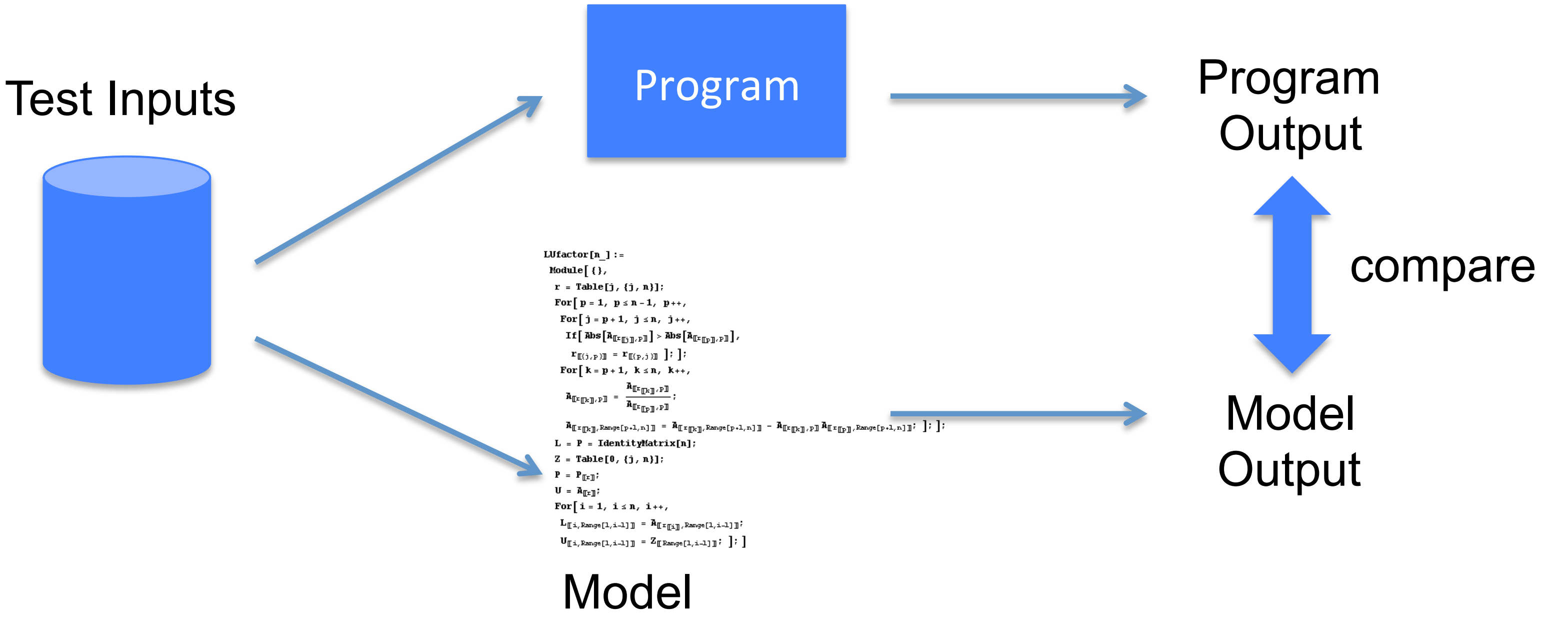
- You do not know the precise result of a simulation but you may know the properties that must hold for the simulation
  - Every item must be part of a travel plan
  - The total money in the stock does not change as a consequence of stock exchanges
  - Items hit by a light cannot be darker than the original item
  - The results obtained assuming fluid incompressibility must not be … than the results obtained with the simulation

Metamorphic Testing

- You do not know the precise result of a simulation but you may know properties that relate the result of a simulation with the result of another simulation
  - If all the items have been scheduled for shipping in simulation X, all the items must be also scheduled for shipping in all the simulations consistent with X that have to ship a smaller number of items
  - Given the brightness of an item in simulation X, the same item cannot be darker in any simulation consistent with X that uses a stronger light

Executable Models

- You have an executable model of your implementation that can be used as an oracle
  - E.g., MatLab or Mathematica model



Did I Write Enough Test Cases?

Why structural testing?

"What is *missing* in our test suite?"

Judging test suite thoroughness based on the *structure* of the program itself

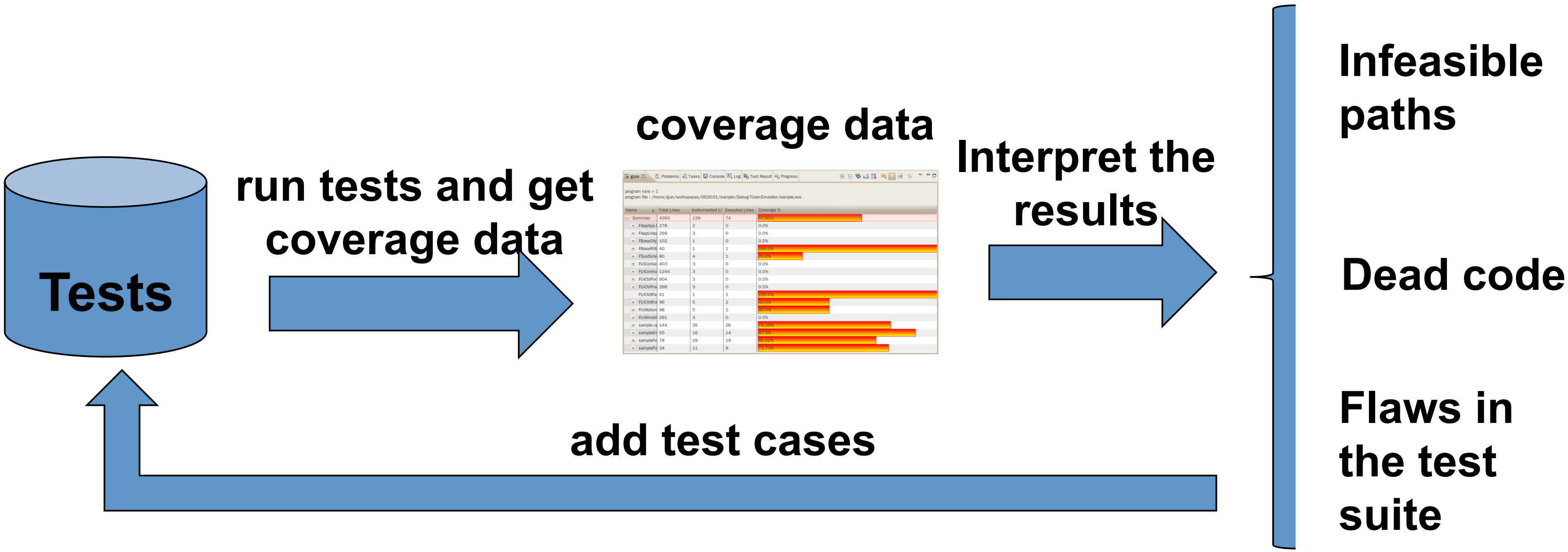
- If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
- But what's a "part"?
  - Typically, a control flow element or combination: e.g., Statements, Branches

No guarantees

Executing all control flow elements does not guarantee finding all faults

- The state may not be corrupted when the statement is executed with some data values
  - E.g., a/b generates a failure only if b == 0
- Corrupt state may not propagate through execution to eventually lead to failure
  - E.g., trainSpeed = 3 X 10^8 m/s generates a problem only if the speed of the train is used in a computation
- What is the value of structural coverage?
  - Increases confidence in thoroughness of testing by removing obvious inadequacies

Structural testing in practice



- Attractive because automated
  - coverage measurements are convenient progress indicators
  - sometimes used as a criterion of completion

Statement testing

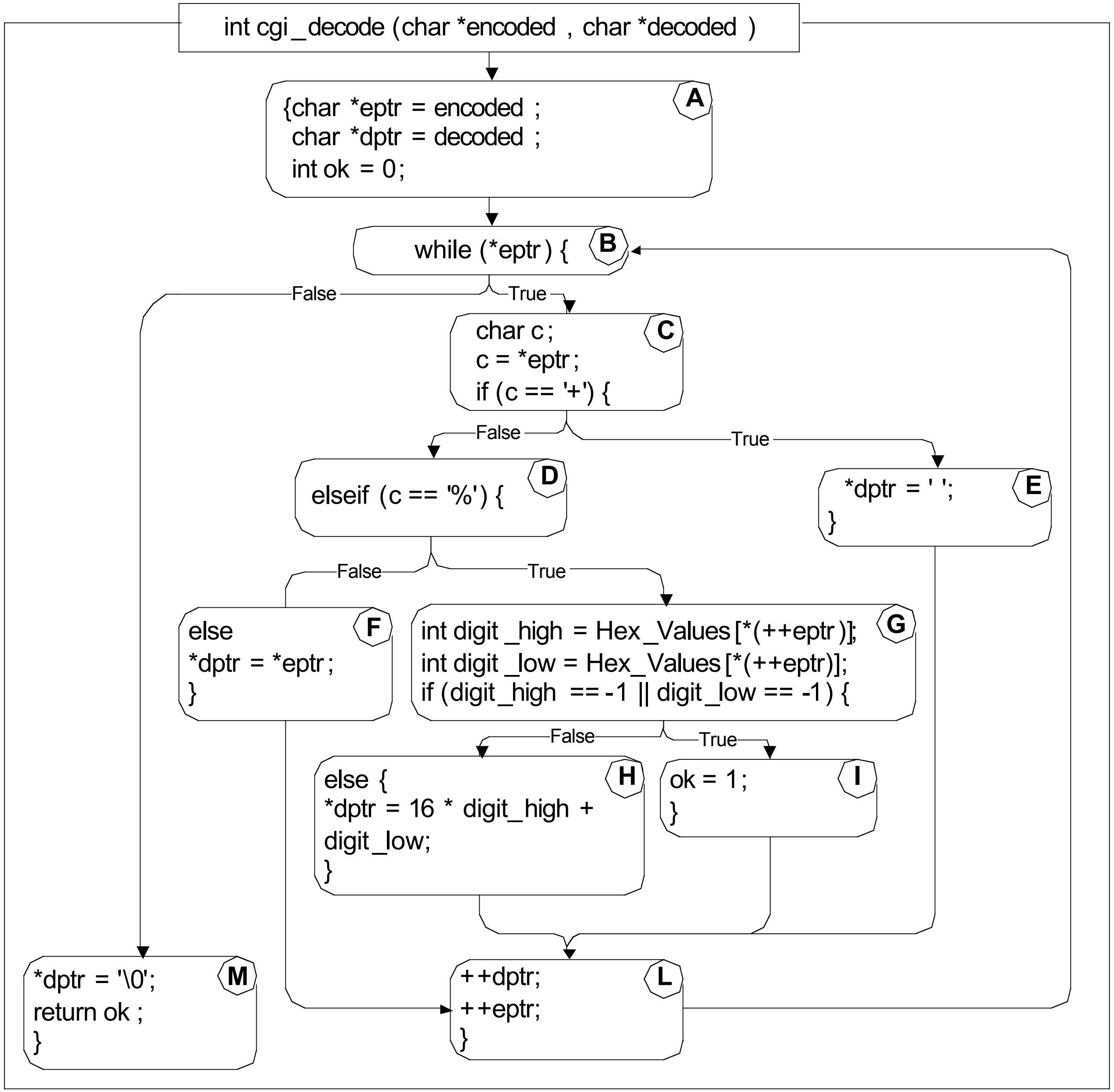
- Adequacy criterion: each statement must be executed at least once
- Coverage:
  - # executed statements/ # statements
- Rationale: a fault in a statement can only be revealed by executing the faulty statement

Example

T_0 =
{“”, “test”,
“test+case%1Dadequacy”}
17/18 = 94% Stmt Cov.

T_1 =
{“adequate+test
%0Dexecution%7U”}
18/18 = 100% Stmt Cov.

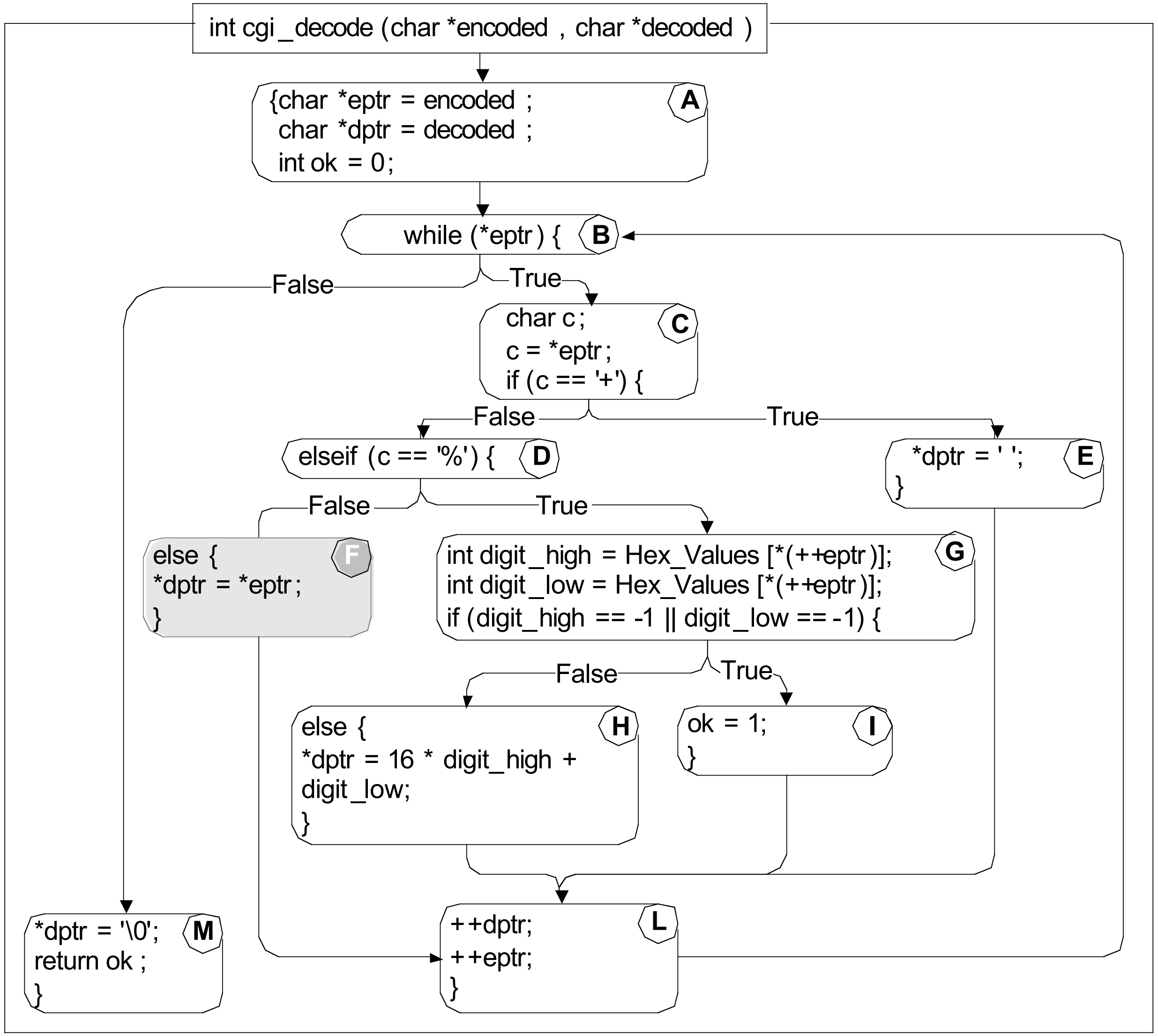
T_2 =
{“%3D”, “%A”, “a+b”,
“test”}
18/18 = 100% Stmt Cov.



"All statements" can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F were missing
  - Statement adequacy would not require *false* branch from D to L

T_3 =
{"", "+%0D+%4J"}
100% Stmt Cov.
No *false* branch from D



Branch testing

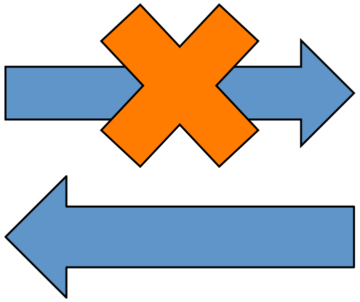
- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage:
  - # executed branches/# branches

T_3 = {“”, “+%0D+%4J”}
100% Stmt Cov. 88% Branch Cov. (7/8 branches)

T_2 = {“%3D”, “%A”, “a+b”, “test”}
100% Stmt Cov. 100% Branch Cov. (8/8 branches)

Statements vs branches

Covering all statements



Covering all branches

Did I Write the Right Test Cases?

Functional testing

- Functional testing: Deriving test cases from program specifications
    - Functional refers to the source of information used in test case design, not to what is tested
- Also known as:
    - specification-based testing (from specifications)
    - black-box testing (no view of the code)
- Functional specification = description of intended program behavior
    - either formal or informal

Systematic vs Random Testing

- Random (uniform):
  - Pick possible inputs uniformly
- Systematic (non-uniform):
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are likely to fail often or not at all
- Functional testing is systematic testing

Why Not Random?

- Non-uniform distribution of faults
- *Example:*

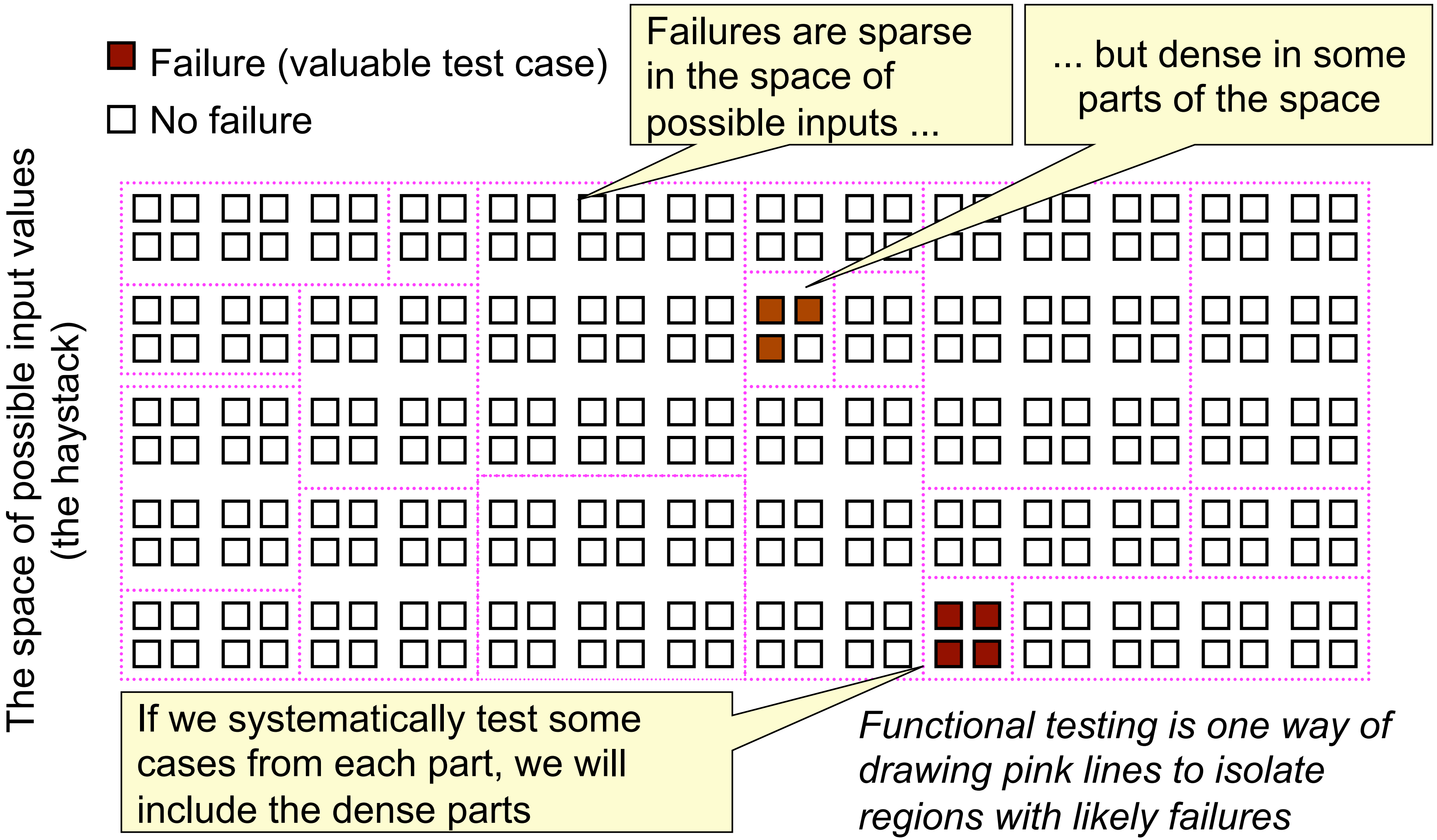
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Assume that fault is an incomplete implementation logic: Program does not properly handle the case in which

$b^2 - 4ac = 0$ and $a=0$

Failing values are *sparse* in the input space — needles in a very big haystack. Random sampling is unlikely to choose a=0.0 and b=0.0

Systematic Partition Testing

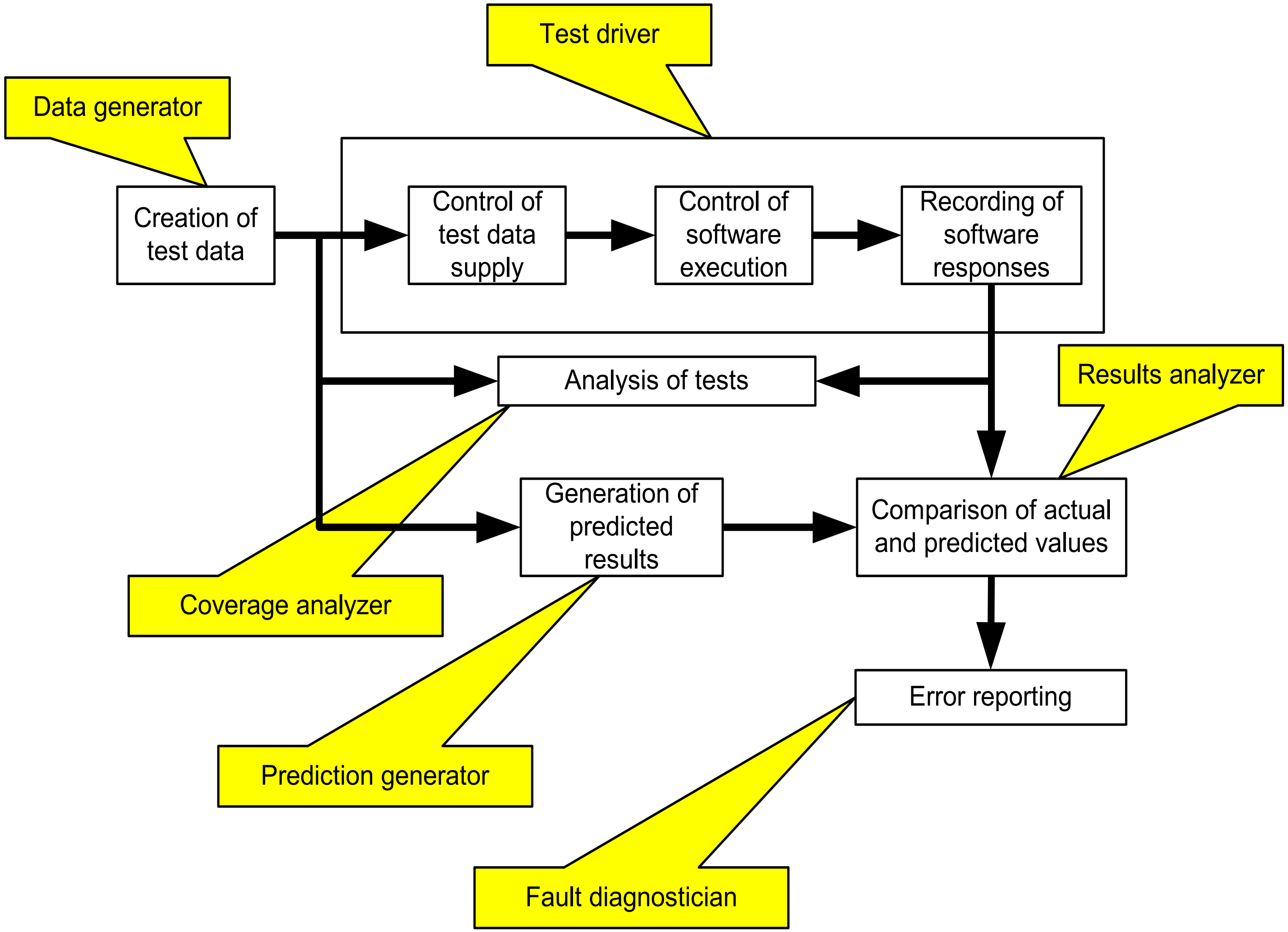


Functional testing is one way of drawing pink lines to isolate regions with likely failures

Steps: From specification to test cases

- 1. Decompose the specification
    - If the specification is large, break it into *independently testable features* to be considered in testing
- 2. Select representatives
    - Representative values of each input, or
    - Representative behaviors of a *model*
- 3. Form test specifications
    - Typically: combinations of input values, or model behaviors
- 4. Produce and execute actual tests

Test environment



Some conclusions

Grazie !!!

luciano.baresi@polimi.it