Advanced School on PARALLEL COMPUTING

Software Test and Analysis

Leonardo Mariani

University of Milano Bicocca

mariani@disco.unimib.it

Quality

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

Quality Process

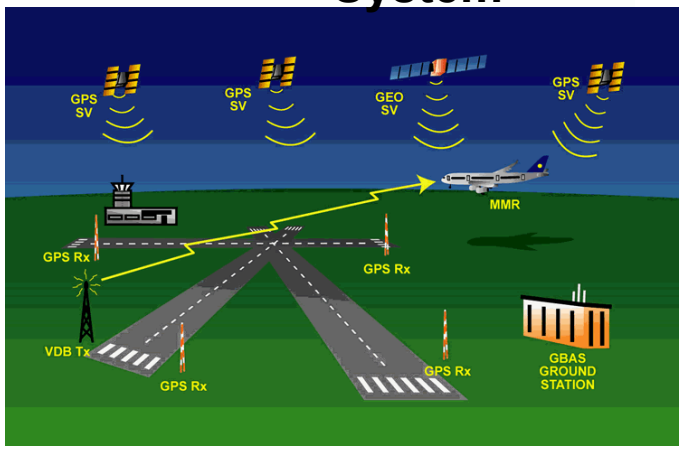
- **activities** + **responsibilities**
  - focused primarily on ensuring adequate **quality**
  - concerned with project **schedule**
  - integral part of the **development process**

What Activities?

Product

Service

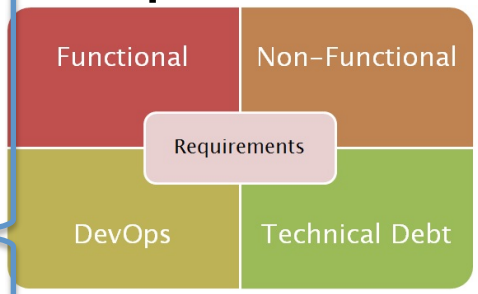
System

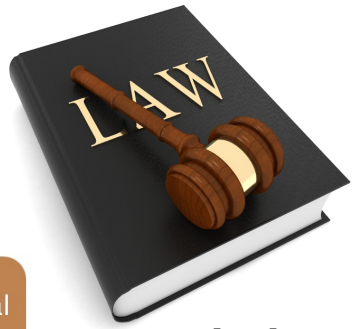


Verification

Validation

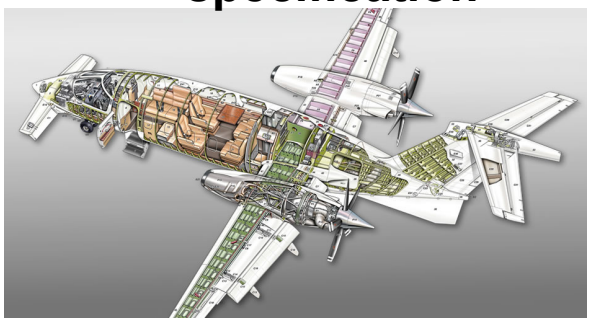
requirements





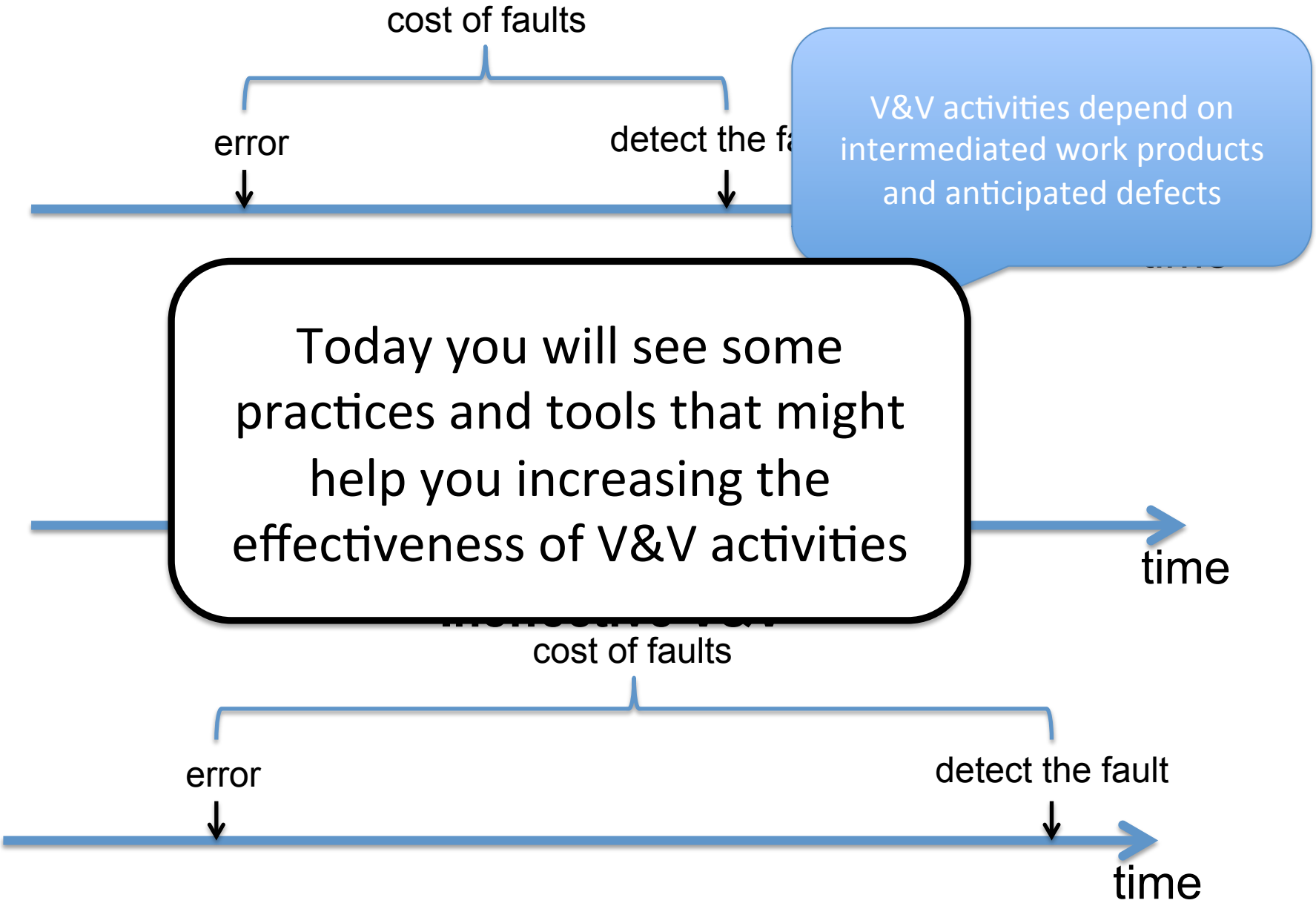
regulations

specification

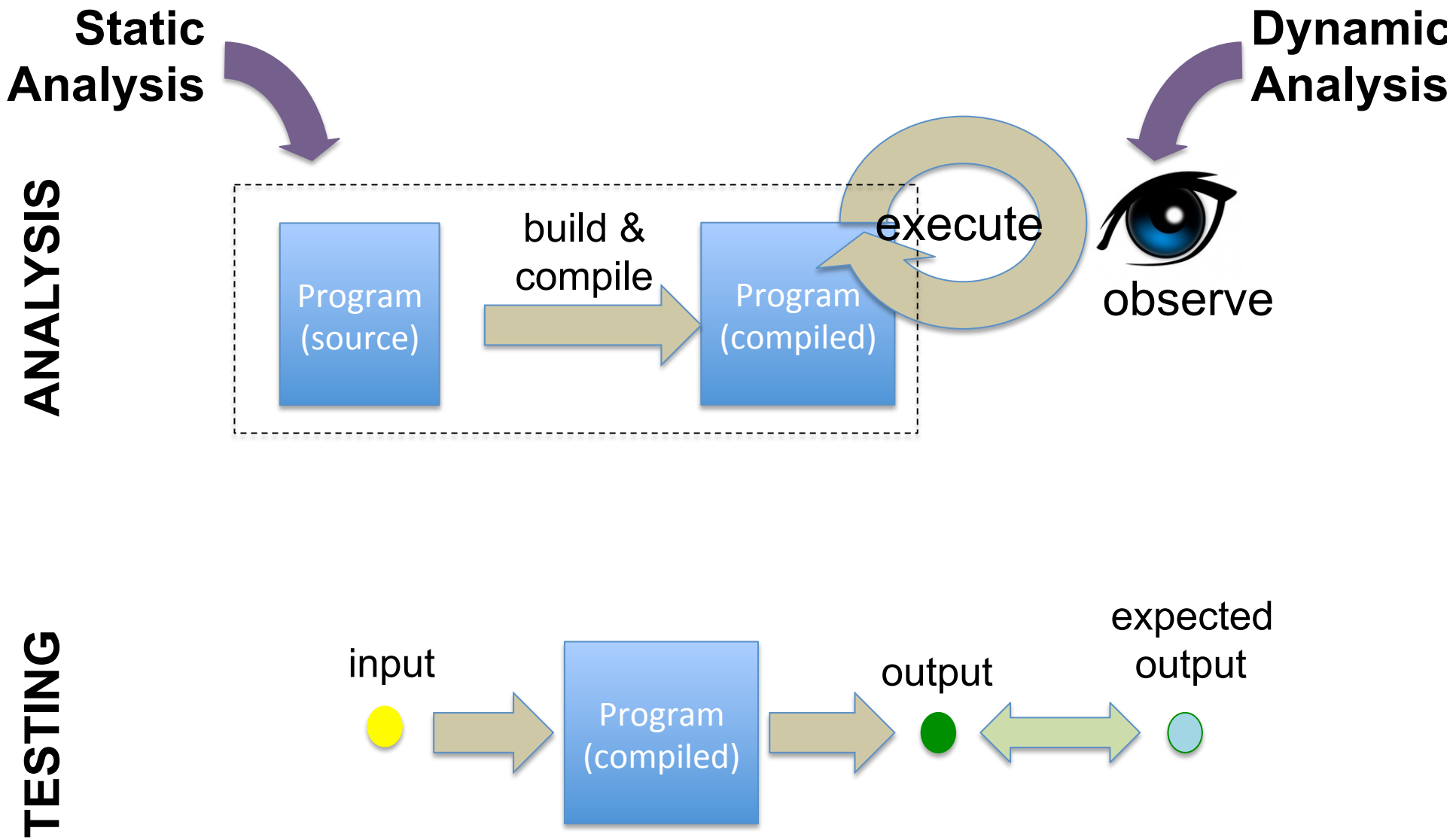


customer

Key Principle of Quality Planning



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
  - Dynamic Analysis
    - Valgrind
- Testing
  - Unit testing
    - Boost unit tests
  - Mocking
    - G(oogle)Mock
  - Coverage
    - gcov

Why Program Analysis?

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

Why Automated Analysis?

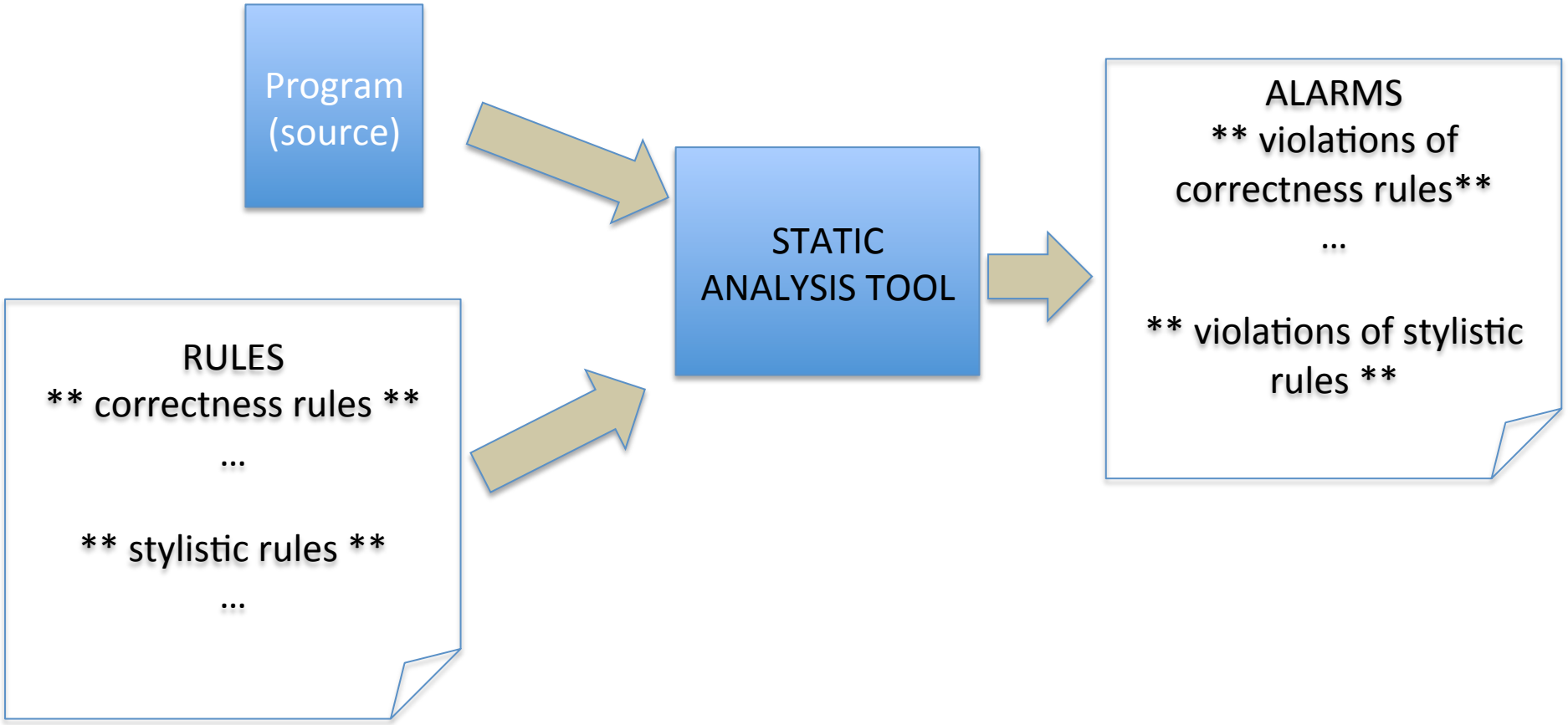
- Manual program inspection effective in finding faults difficult to detect with testing
- But humans are not good at
  - repetitive and tedious tasks
  - maintaining large amounts of detail
- Automated analysis replace human inspection for some classes of faults

Static vs dynamic analysis

- Static analysis
  - examine program source code
    - examine the complete execution space
    - but may lead to **false alarms**
- Dynamic analysis
  - examine
    - **no inf**
    - but **ca**

```
PowerManager::PowerManager(IMsgSender* msgSender)
  : msgSender_(msgSender) { }

void PowerManager::SignalShutdown()
{
    msgSender_->sendMsg("shutdown()");
}
```

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

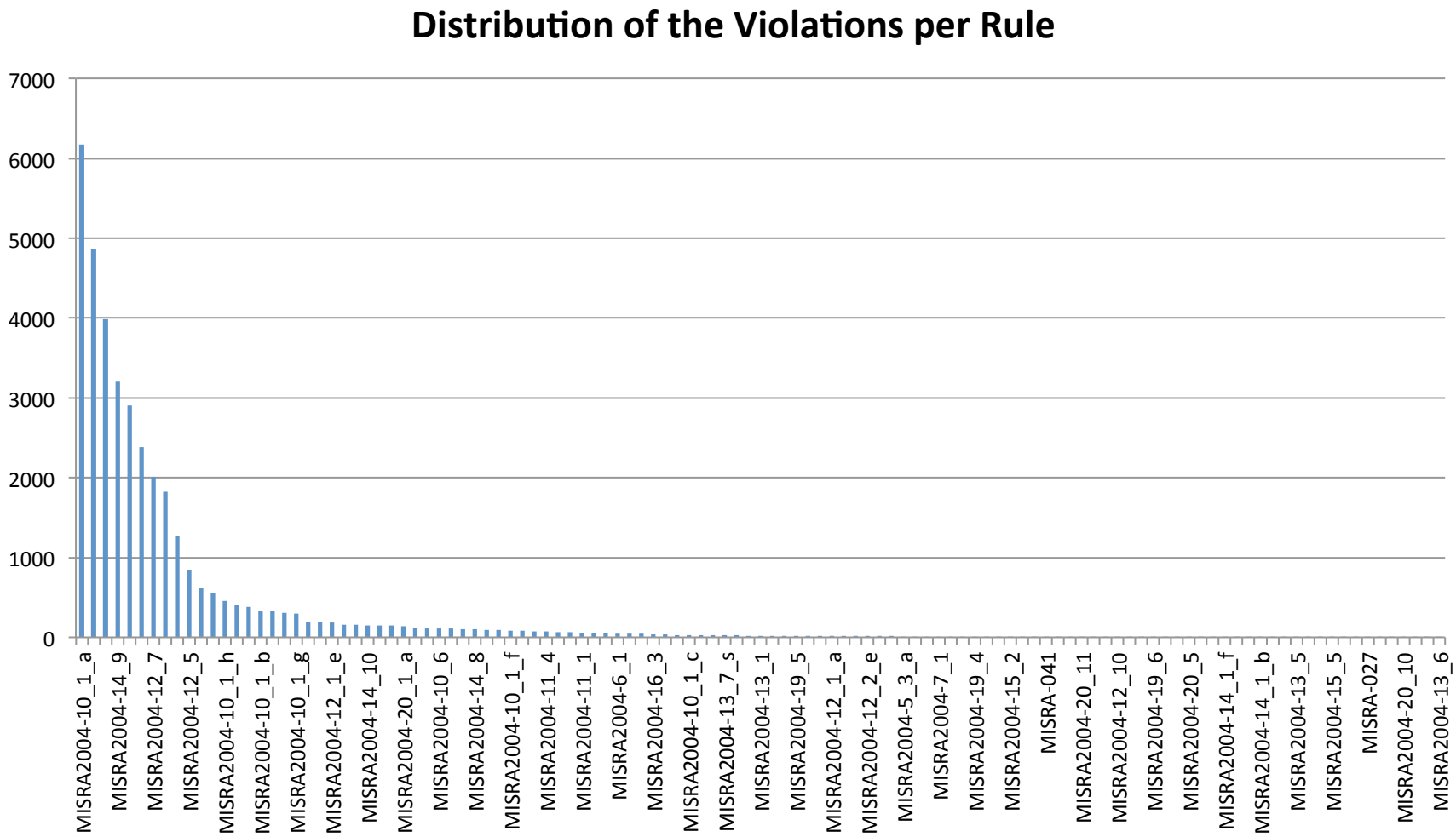
Example

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

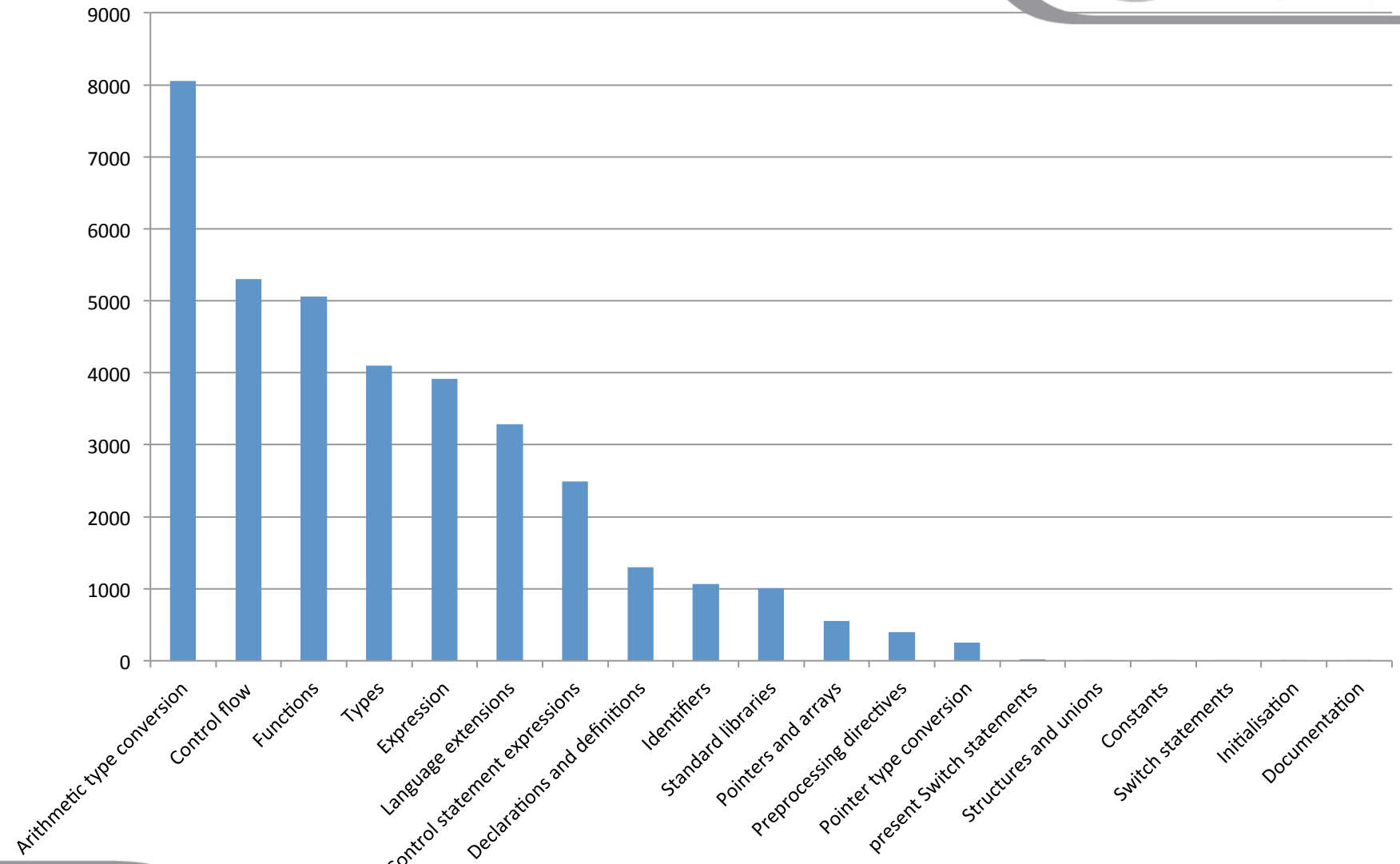
An Experience from a Real Case: Checking MISRA Rules

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

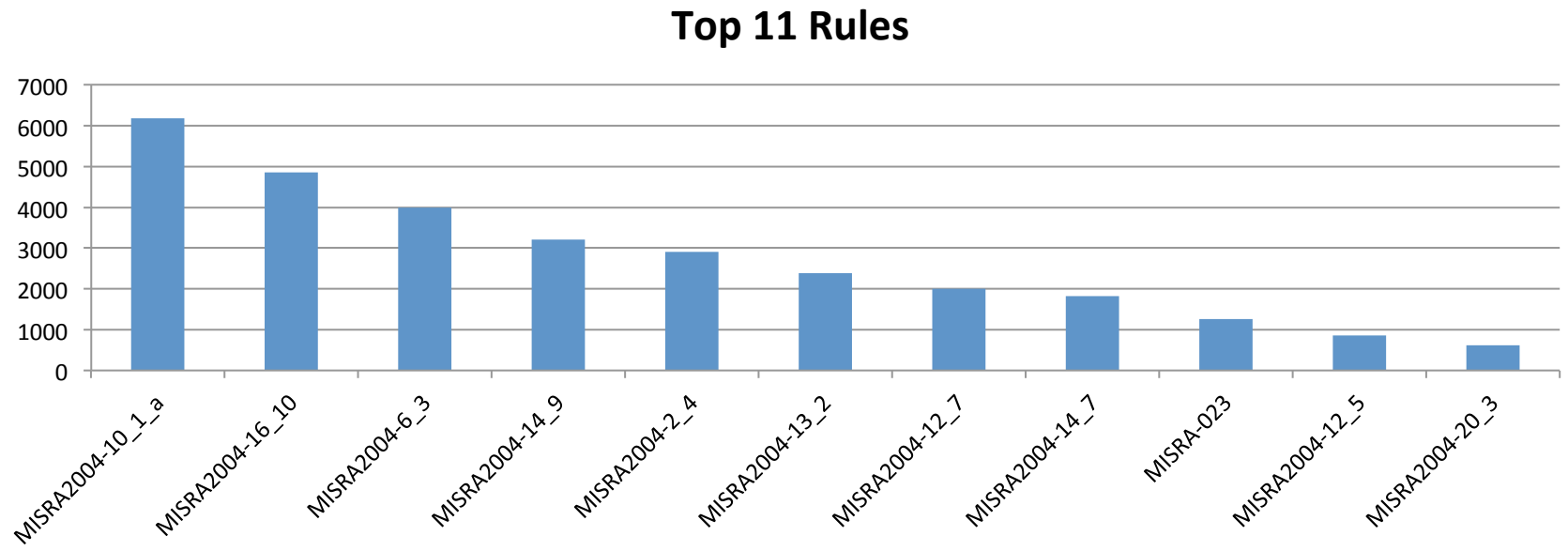
- 36.850 rule violations

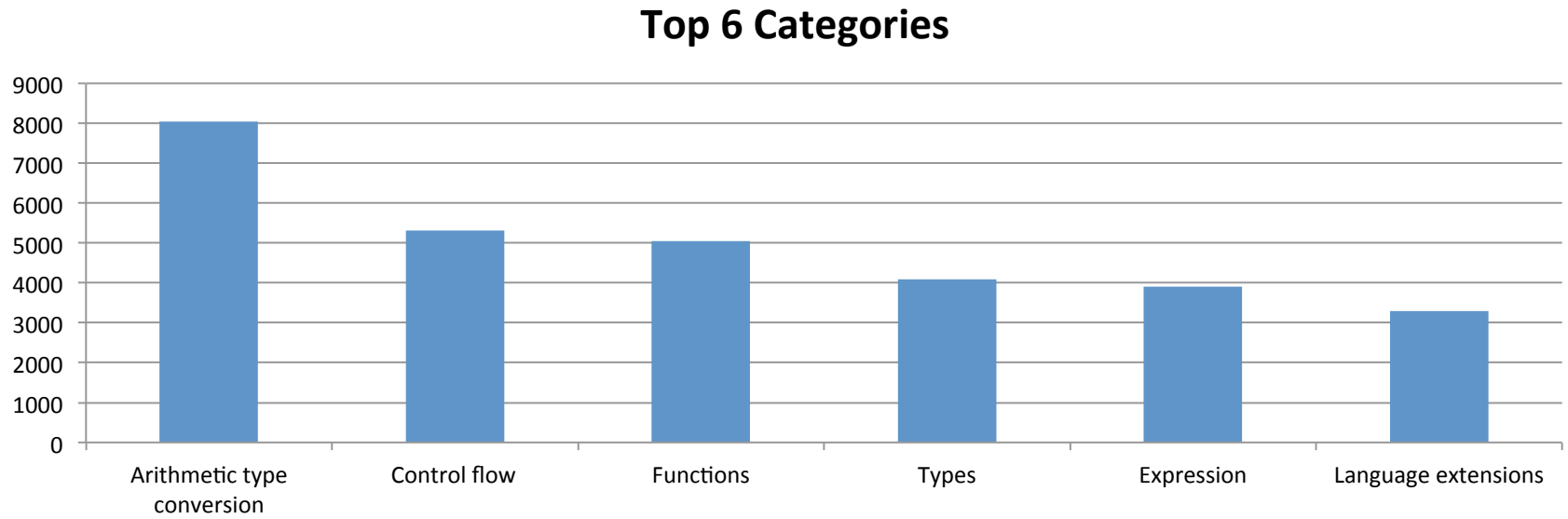


Distribution of the Violations per Category



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)

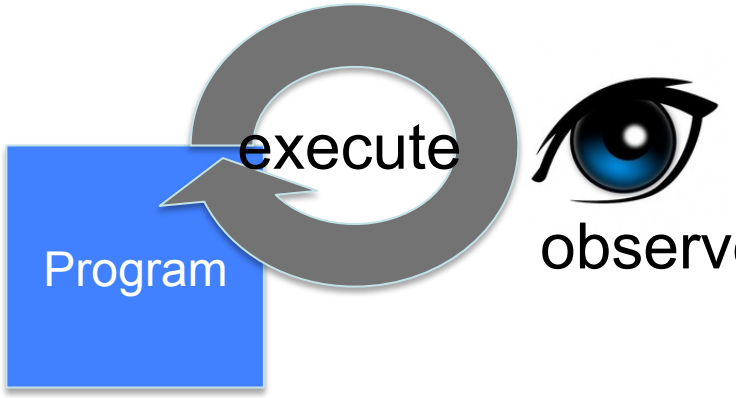
- Code Complexity = how hard is to maintain, test, debug, … the software
- Thus do no write complex code!

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

Dynamic Analysis



Easy to execute but hard to fail cases:

- Memory Leak
- Data races

…

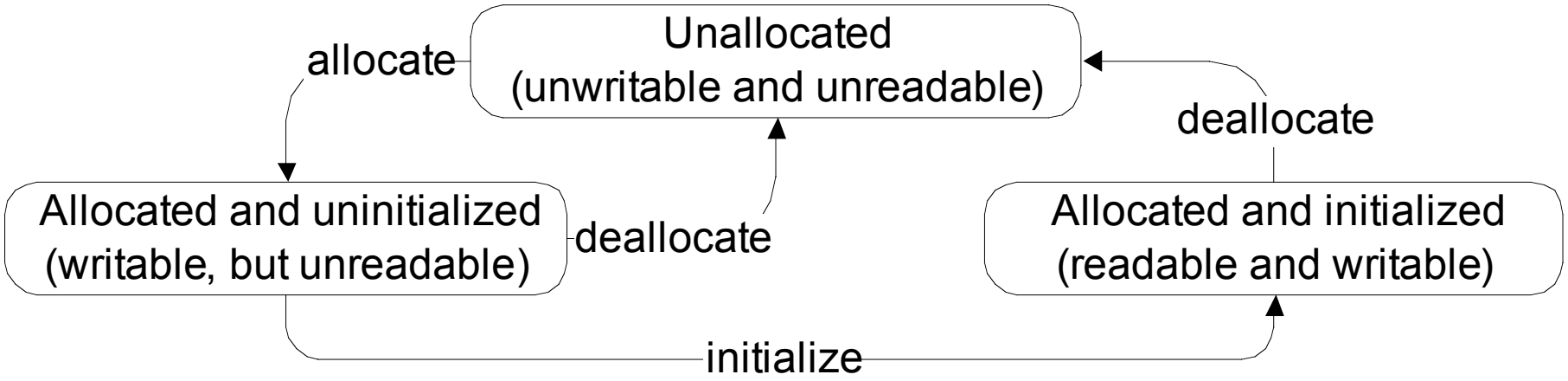
Do you see any fault in this piece of code?

```
void f(void) {
    int* x = malloc(10 * size
    x[10] = 0;
}
```

Heap block overrun
- sporadic failures

Memory leak
- Slow down and crashes in long running executions

(Dynamic) Memory Analysis



Data Race

```
#include <thread>
#include <iostream>
#include <vector>

unsigned const increment_count=2000000;
unsigned const thread_count=2;

unsigned i=0;

void func()
{
   for(unsigned c=0;c<increment_count;++c)
   {
      ++i;
   }
}
```

```
int main()
{
   std::vector<std::thread> threads;
   for(unsigned c=0;c<thread_count;++c)
   {
      threads.push_back(std::thread(func));
   }
   for(unsigned c=0;c<threads.size();++c)
   {
      threads[c].join();
   }

   std::cout<<thread_count<<" threads, Final i="<<I
   <<", increments="<<(thread_count*increment_count)
   <<std::endl;
}
```

What is the output of this program?

2 threads, Final i=2976075, increments=4000000
2 threads, Final i=3097899, increments=4000000
2 threads, Final i=4000000, increments=4000000
2 threads, Final i=3441342, increments=4000000
2 threads, Final i=2942251, increments=4000000

Data Race

```
#include <thread>
#include <iostream>
#include <vector>

unsigned const increment_count=2000000;
unsigned const thread_count=2;

unsigned i=0;

void func()
{
   for(unsigned c=0;c<increment_count;++c)
   {
      ++i;
   }
}
```

```
int main()
{
   std::vector<std::thread> threads;
   for(unsigned c=0;c<thread_count;++c)
   {
      threads.push_back(std::thread(func));
   }
   for(unsigned c=0;c<threads.size();++c)
   {
      threads[c].join();
   }

   std::cout<<thread_count<<" threads, Final i="<<I
   <<", increments="<<(thread_count*increment_count)
   <<std::endl;
}
```

Data races can compromise the correctness of the program!
Serious problem in concurrent (and long running) software

Simple lockset analysis: example

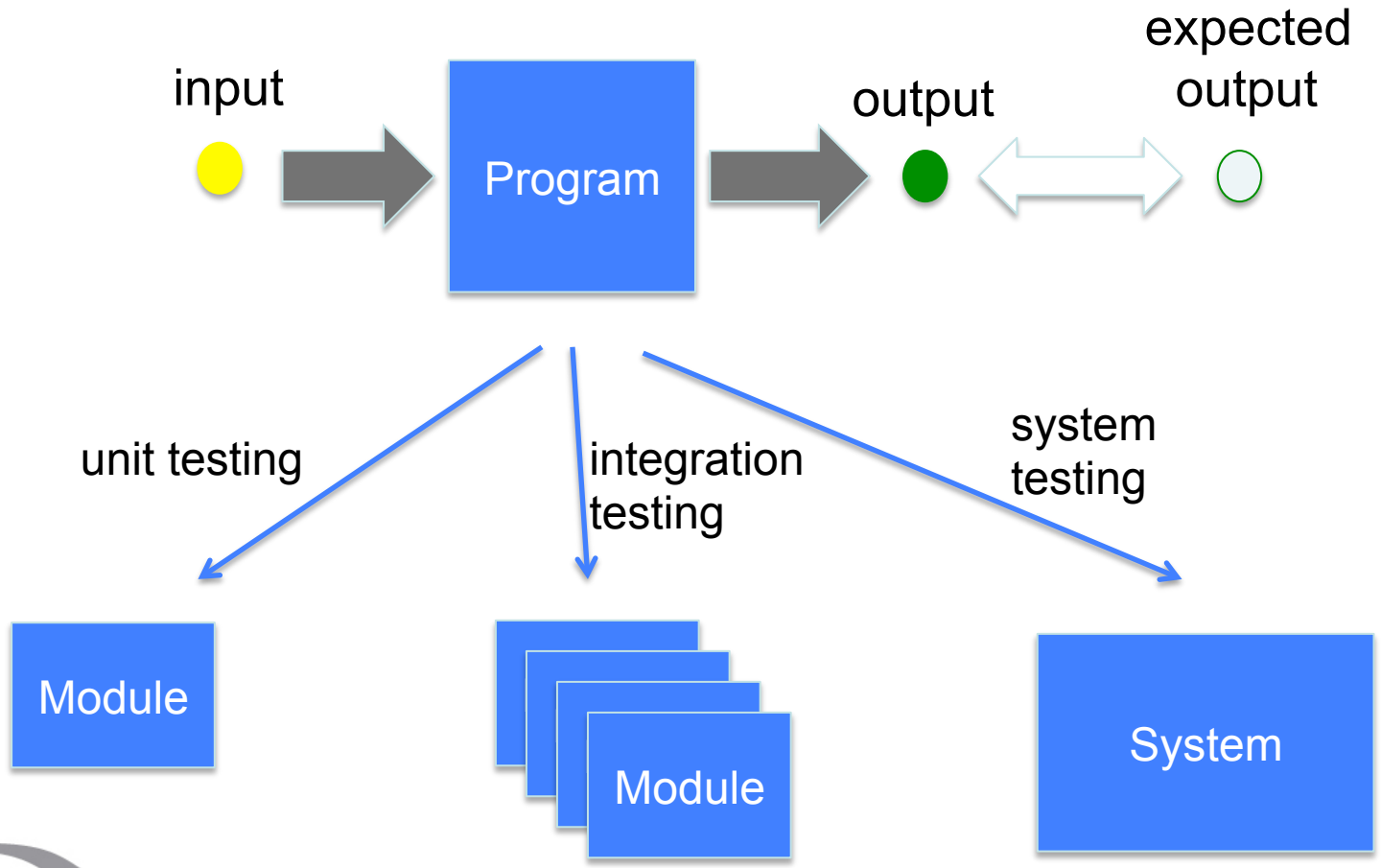
Thread	Program trace	Locks held	Lockset(x)	
		{}	{lck1, lck2}	INIT:all locks for x

Example

- ValGrind
  - provides several dynamic analysis tools
    - Memcheck most popular tool
  - Compile with -g

Testing

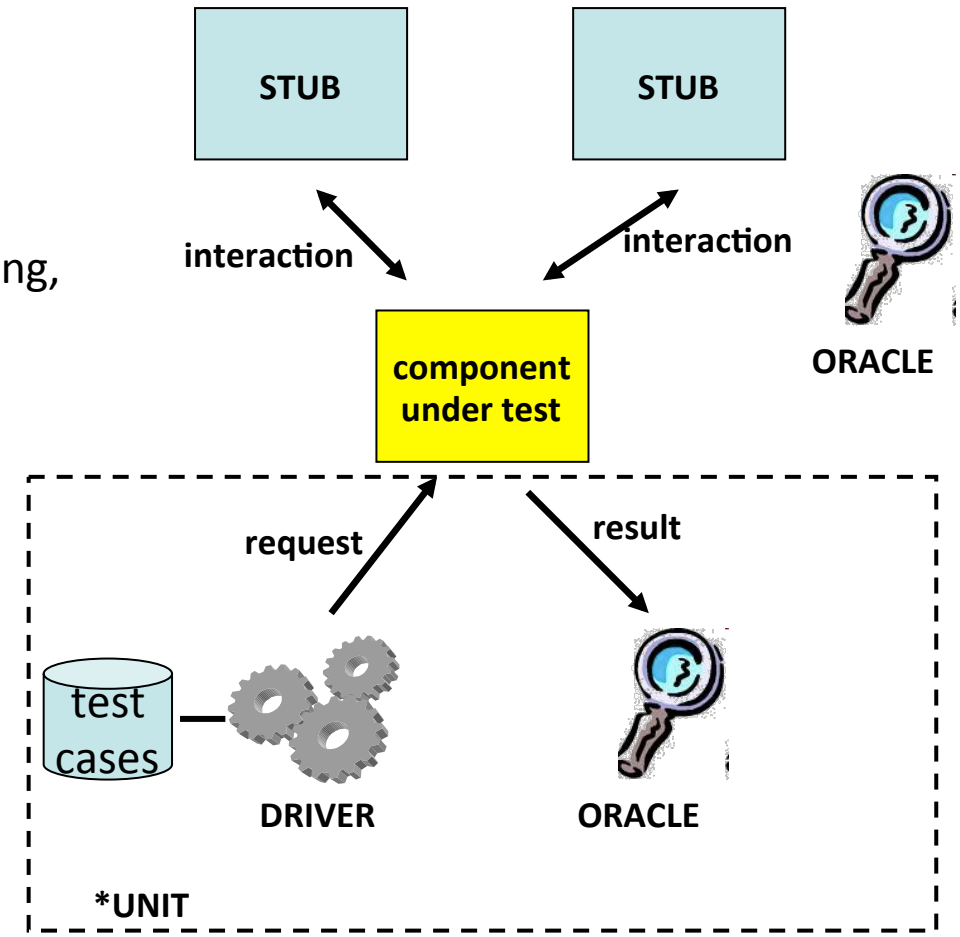
Testing Levels



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

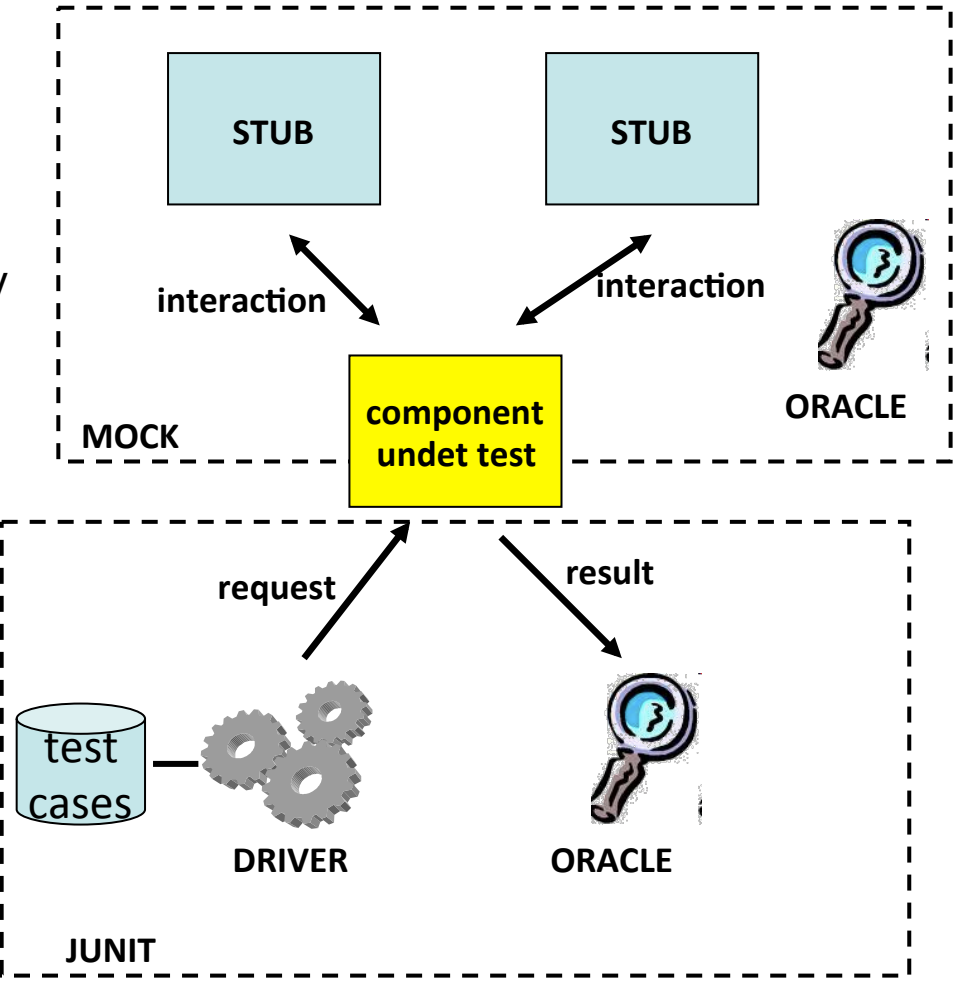
```
int add( int i, int j ) { return i + j; }

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
}
```

Example

- BOOST Unit testing with Eclipse CDT

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`)



Example

- Gmock + BOOST Unit Testing with Eclipse CDT

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

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

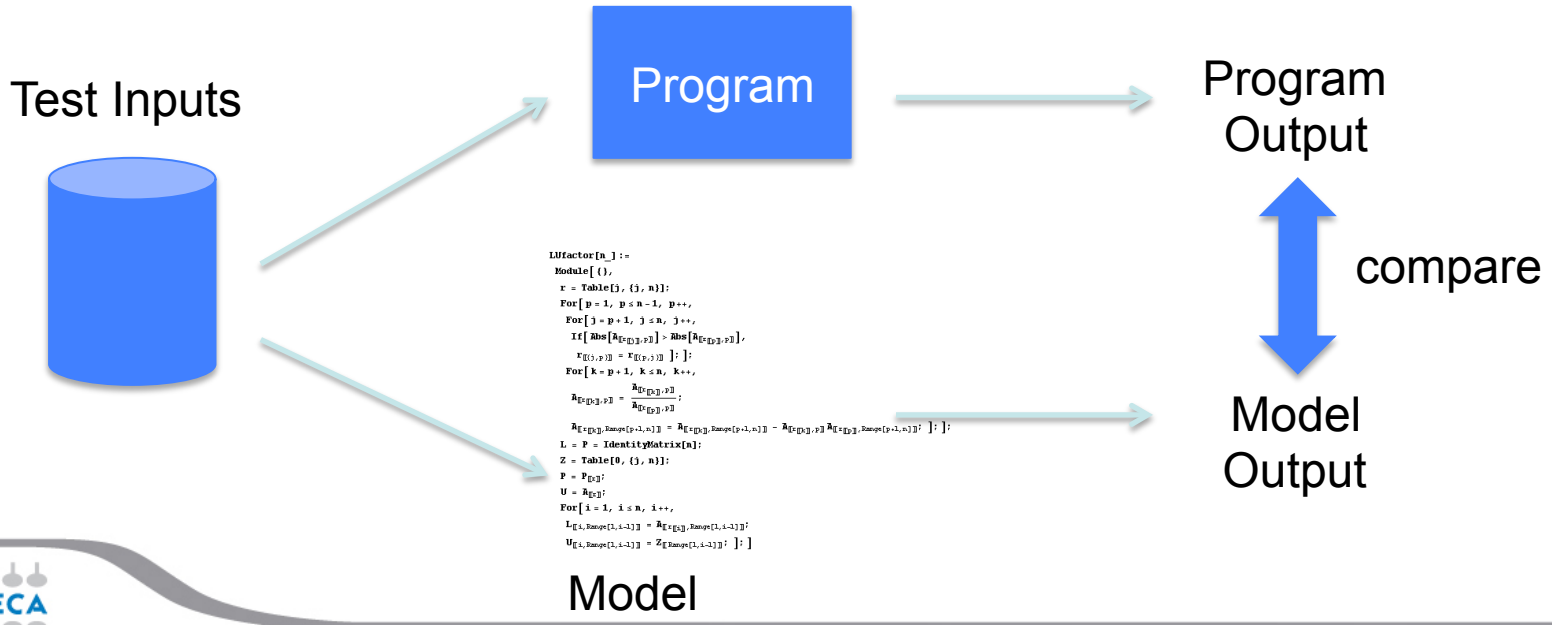
- You do not know the precise result of a simulation but you may know the **properties that must hold for the simulation**
  - *Every item must be part of a travel plan*
  - *The total money in the stock does not change as a consequence of stock exchanges*
  - *Items hit by a light cannot be darker than 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 (code-based) 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

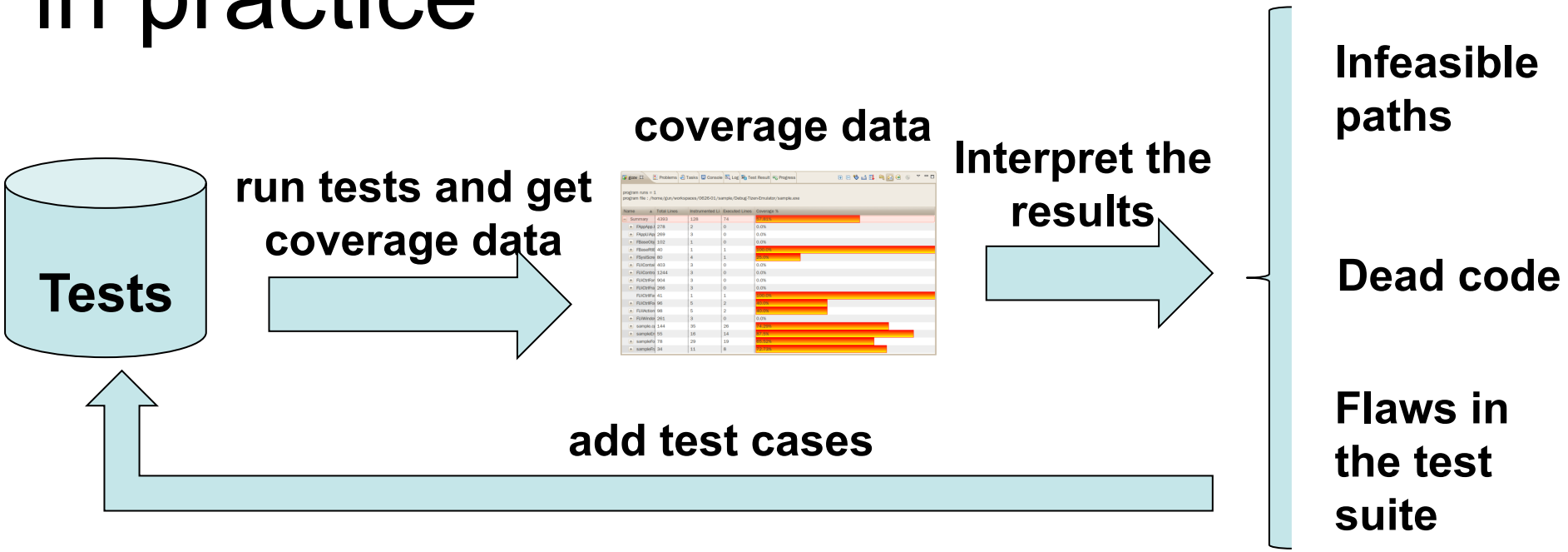
No guarantees

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:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

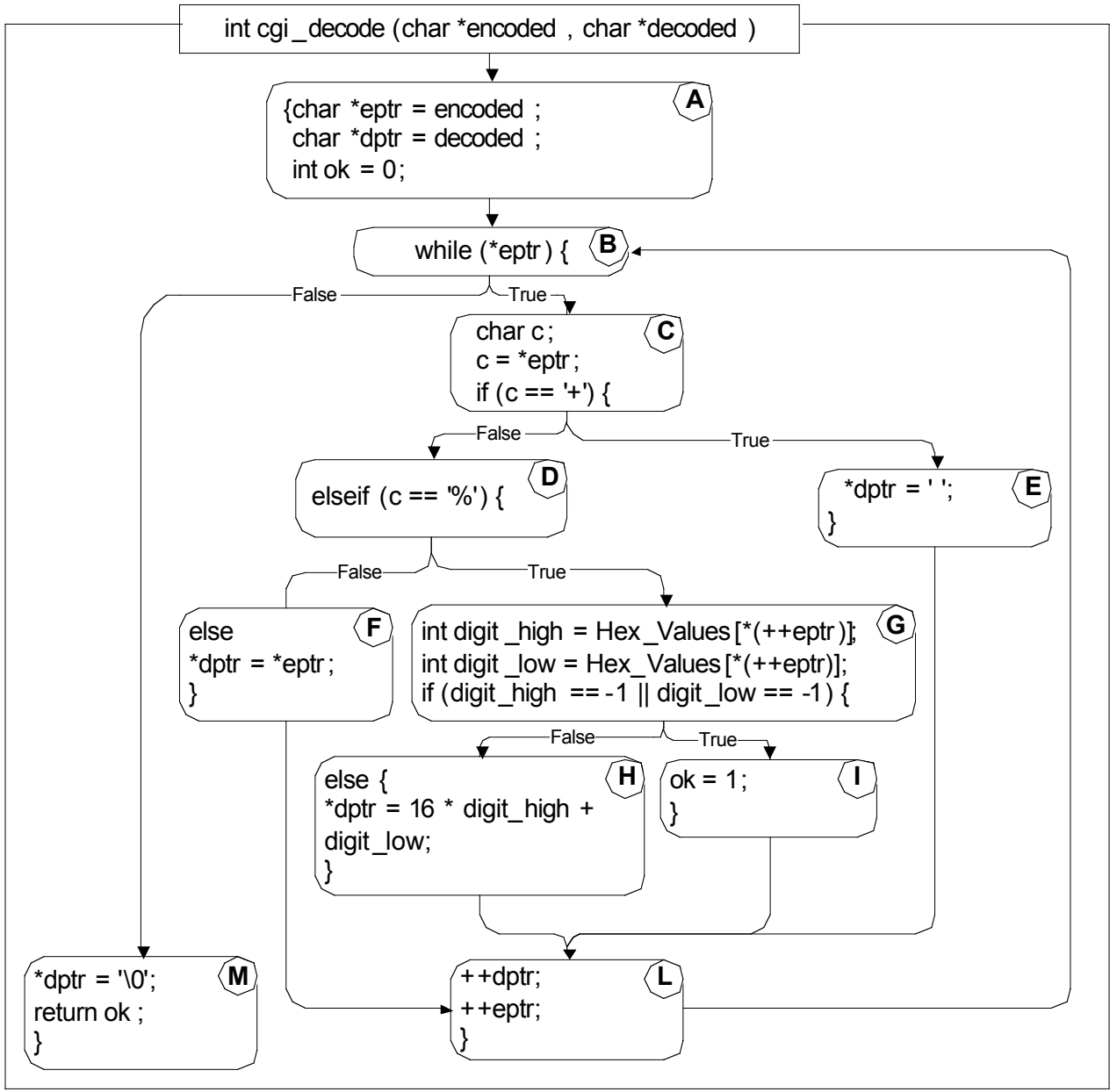
- Rationale: a fault in a statement can only be revealed by executing the faulty statement

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

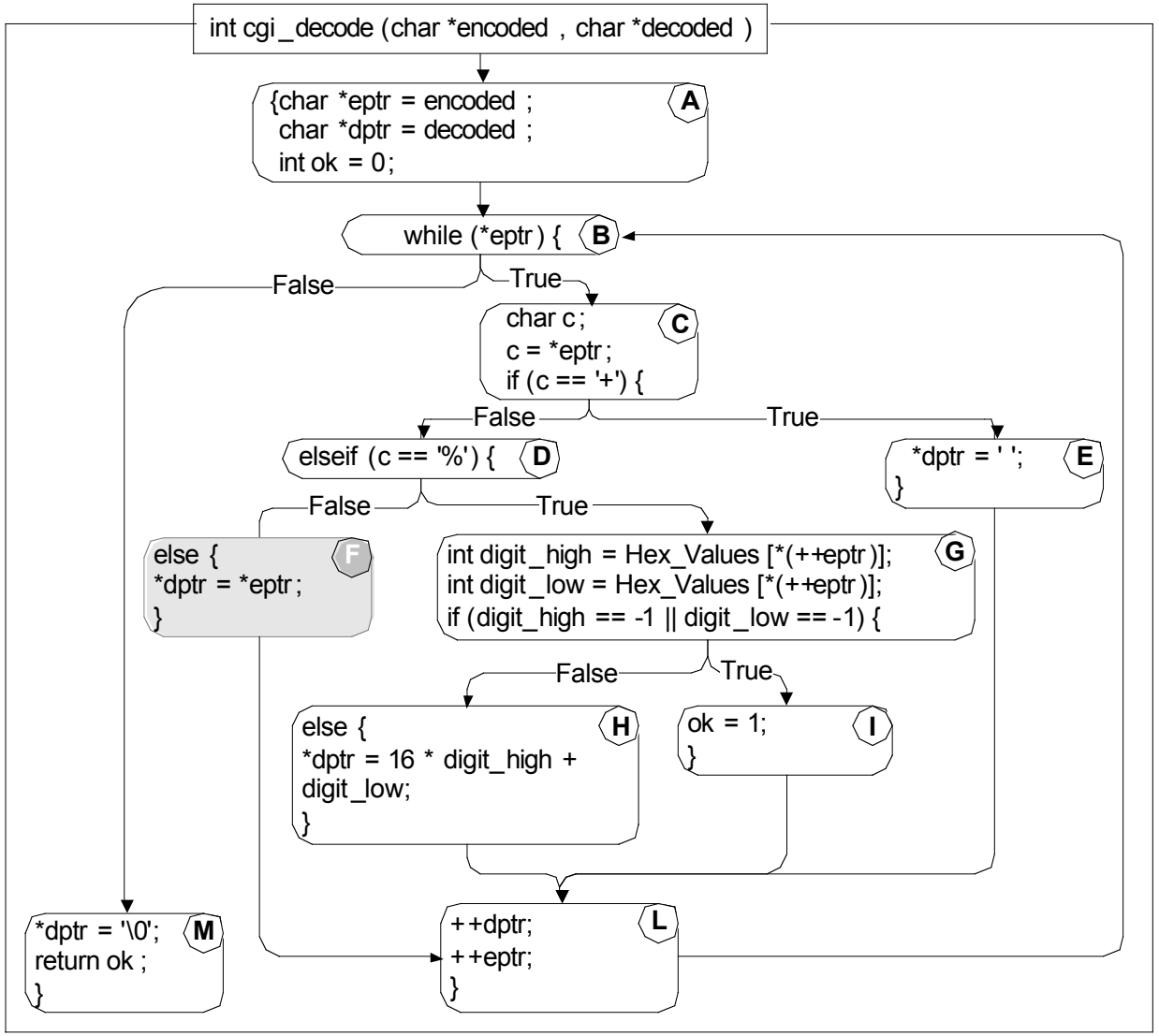
- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F were missing
  - Statement adequacy would not require *false* branch from D to L

T_3 =

{“”, “+%0D+%4J”}

100% Stmt Cov.

No *false* branch from D



Branch testing

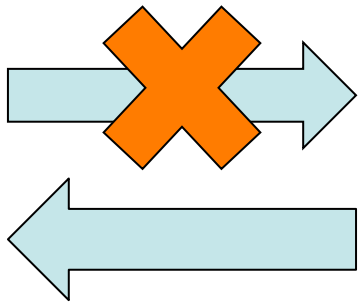
- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

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

Example

- Collecting coverage information with gcov

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:*

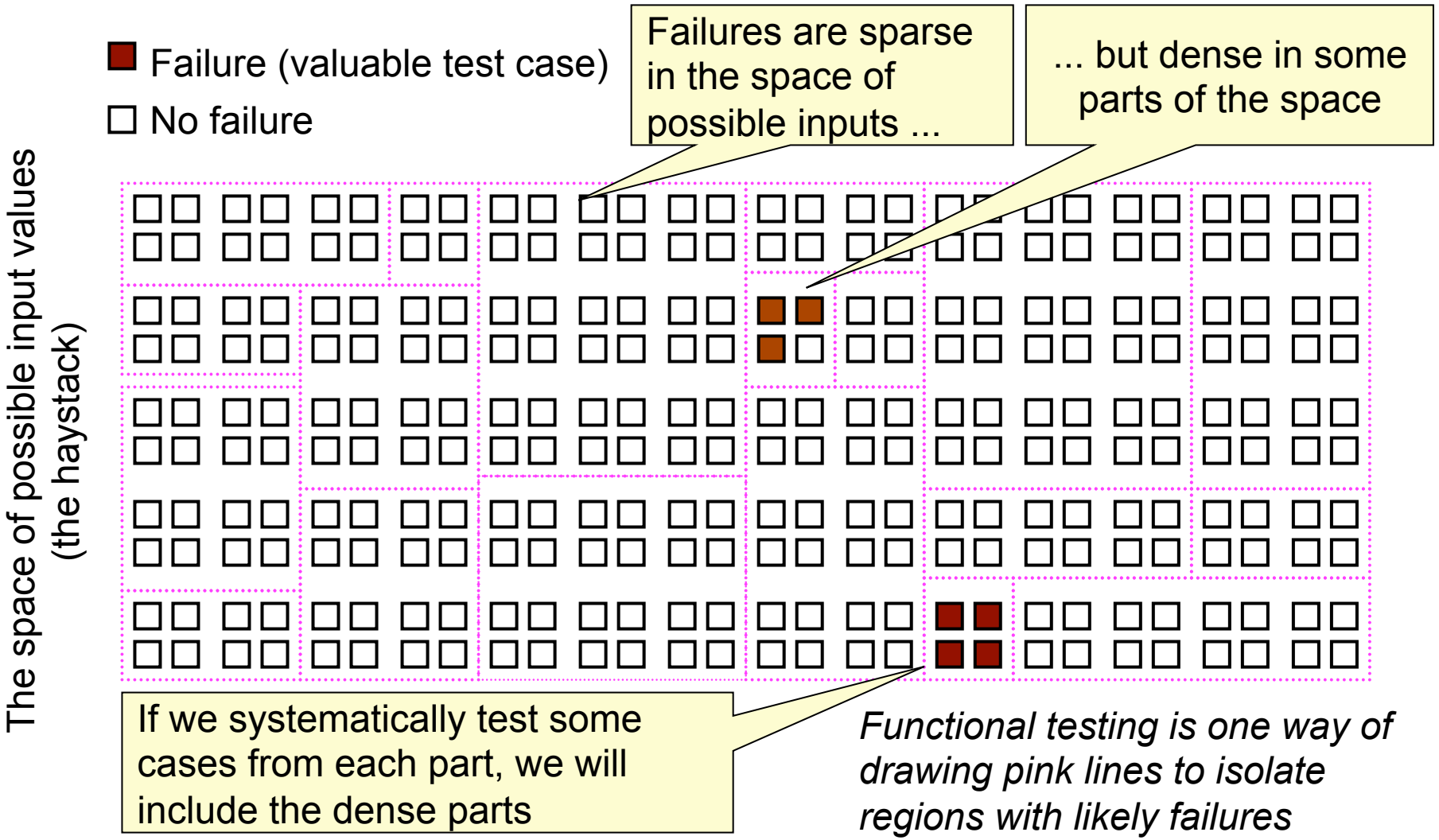
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

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

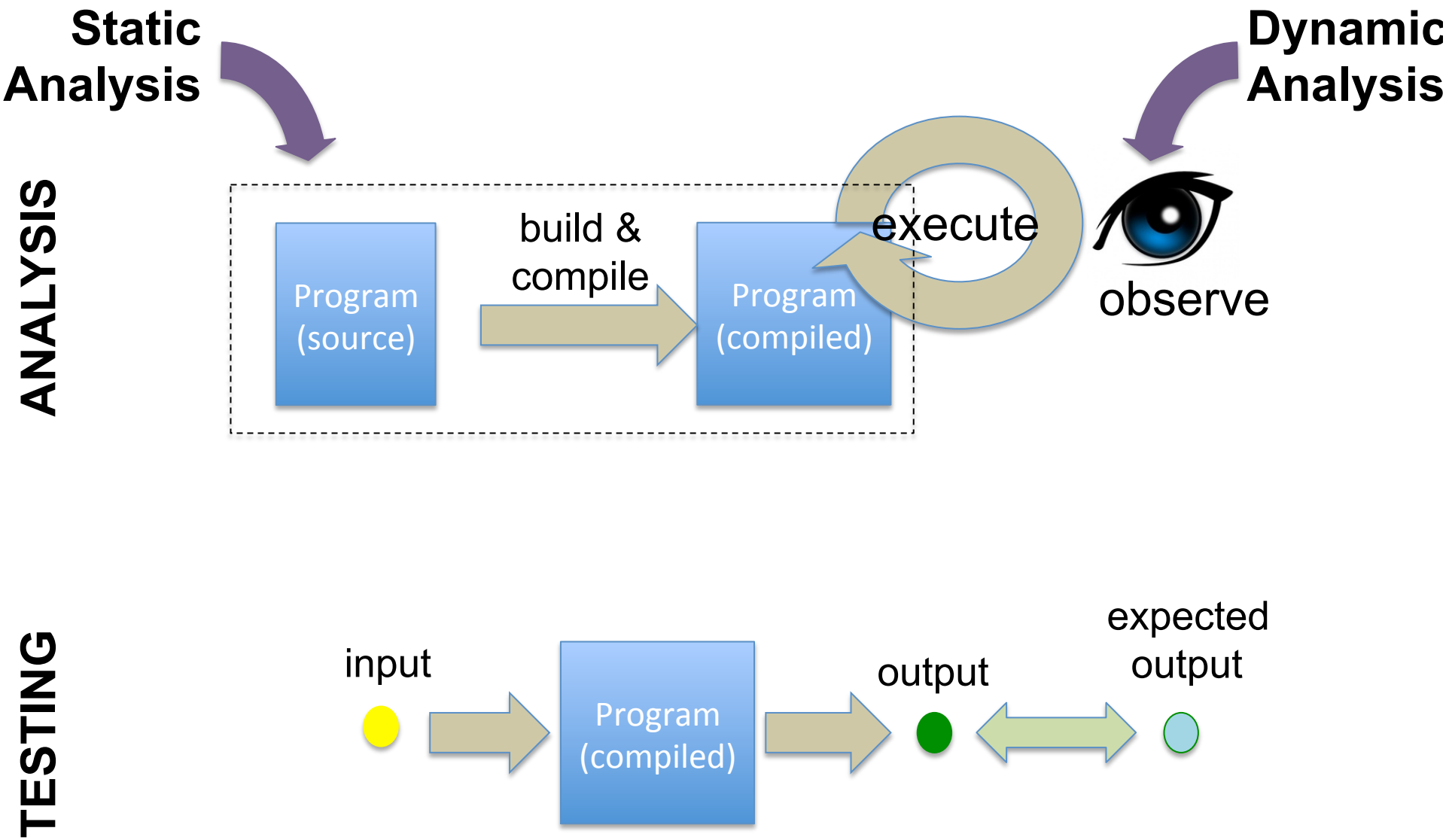
Systematic Partition Testing



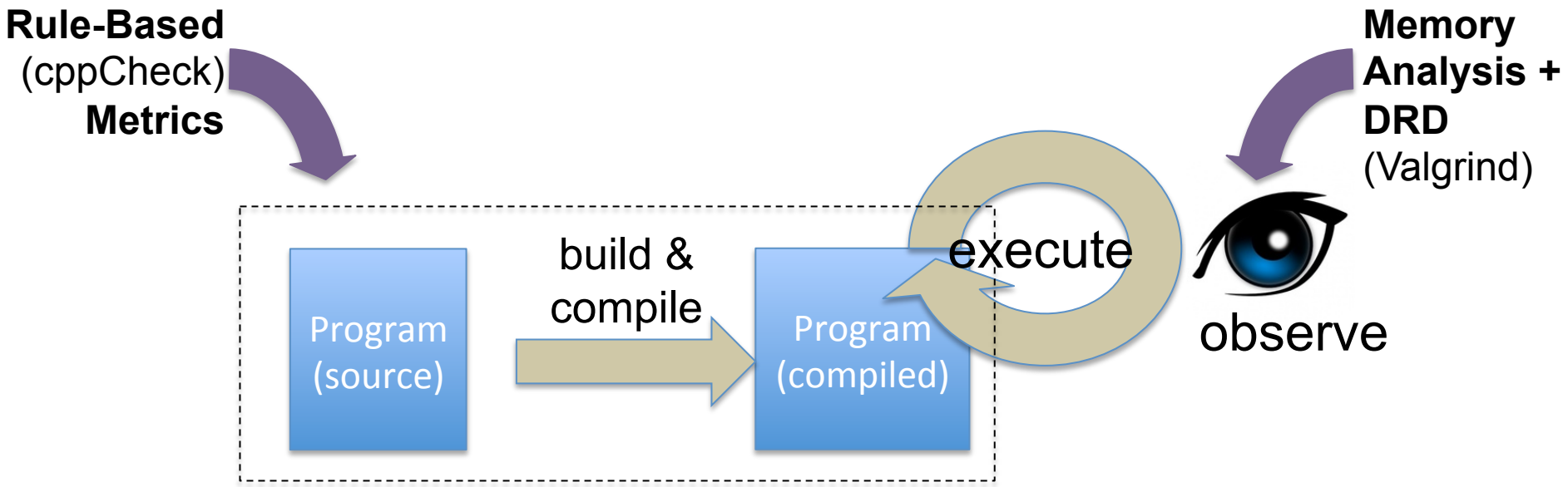
Steps: From specification to test cases

- 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

Take Home


Static Analysis
Dynamic Analysis
ANALYSIS
Program (source)
build & compile
Program (compiled)
execute
observe
TESTING
input
Program (compiled)
output
expected output

Take Home



Unit testing
(Boost)
Mocking
(GMock)
Coverage
(gcov)