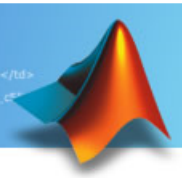
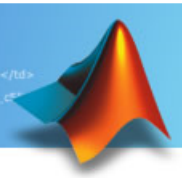


軟體程式碼驗證工具及實例應用

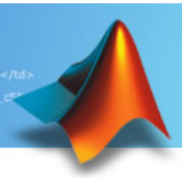
Terasoft, Inc.

CDA engineer : Kary Chang



Agenda

- 簡介
- 軟體驗証測試
- Demo

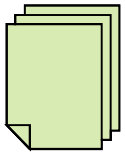


Agenda

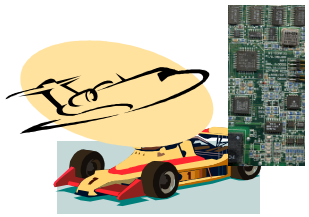
- 簡介
- 軟體驗証測試
- Demo

傳統嵌入式軟體開發流程的缺點

Requirements and Specs



Design



Implementation

Test and Verification

Text-based
- ambiguous
- difficult to track
- prevents rapid iteration

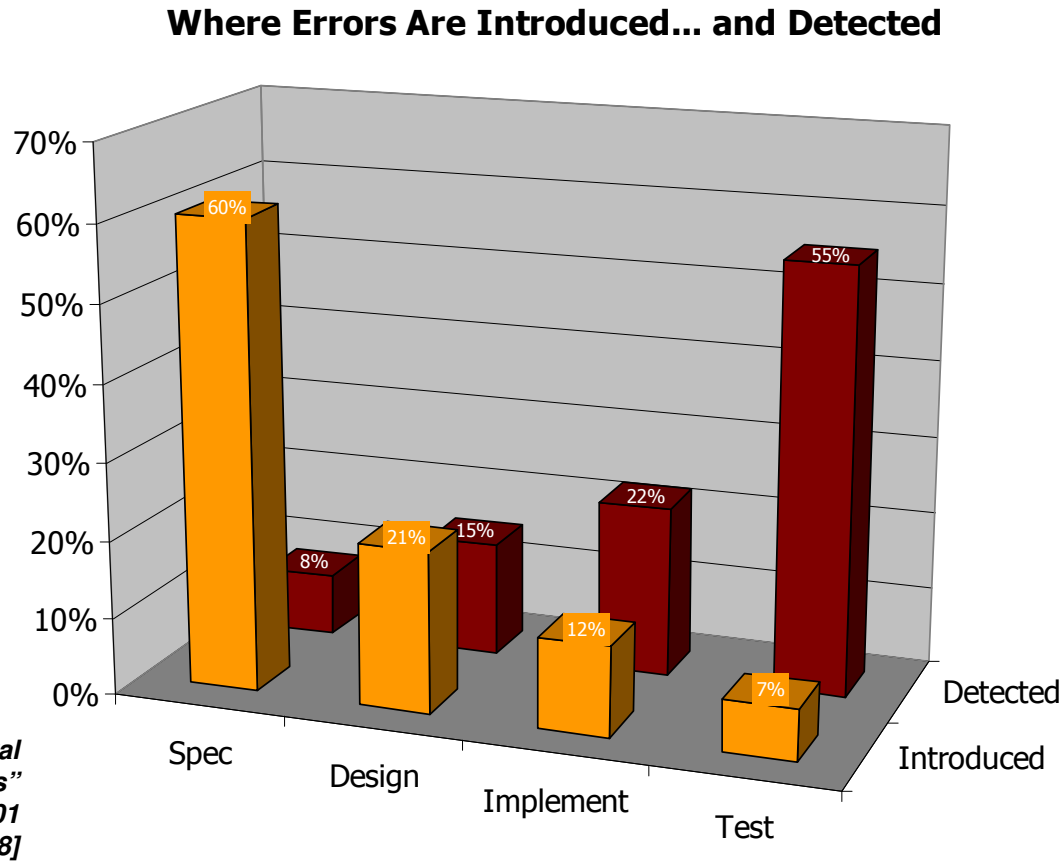
Physical prototypes
- aren't always available
- are typically expensive
- don't test the complete range of scenarios

Manual coding
- introduces human error
- takes more time
- doesn't promote reusability
- doesn't leverage the system-level design of the software and hardware

Traditional testing
- defined by (ambiguous) specs
- doesn't benefit from the design know-how
- errors detected late in the process

錯誤太晚被發現，造成大量的時間與金錢的花費

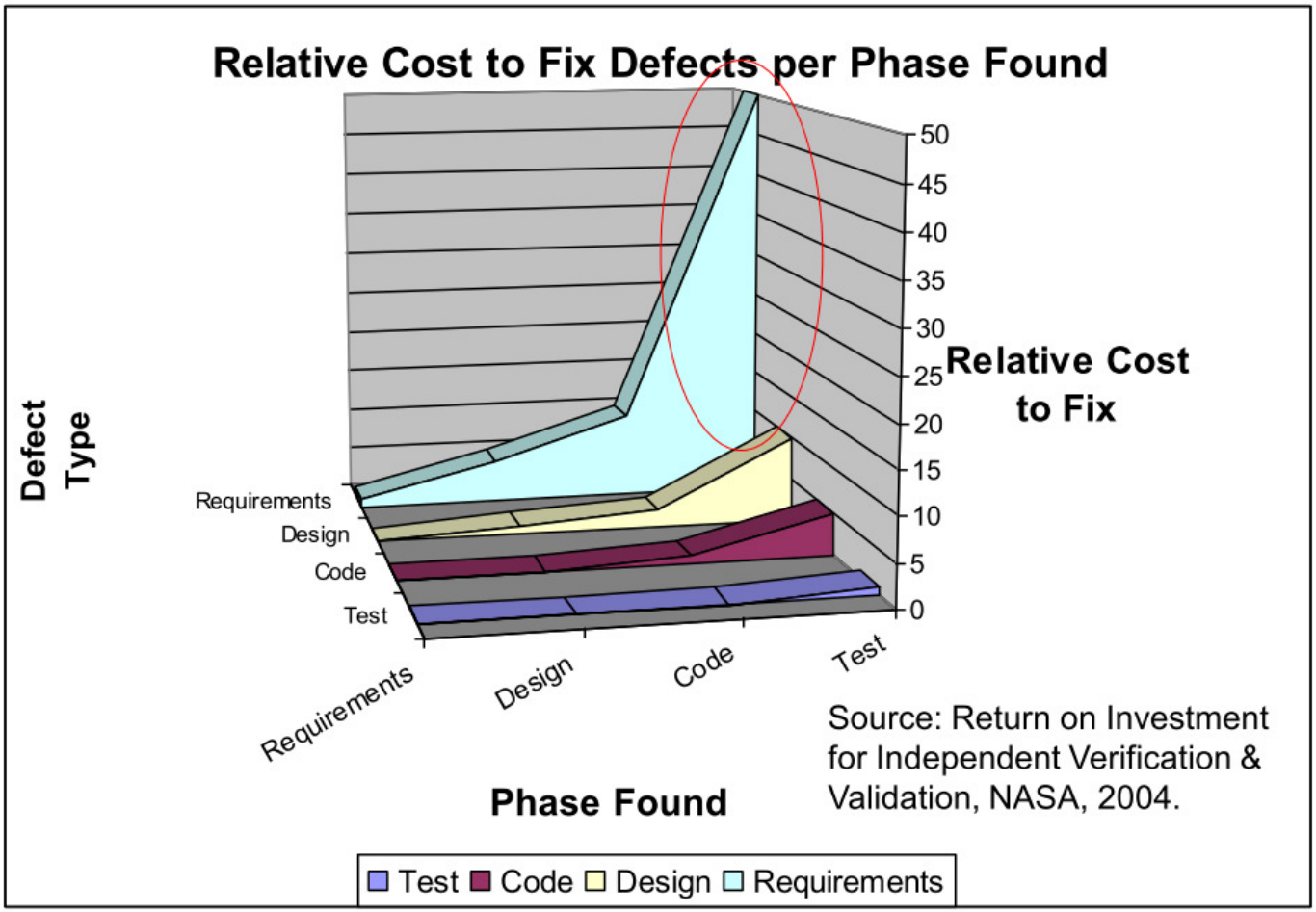
"…each delay in the detection and correction of a design error makes it an order of magnitude more expensive to fix…"



Clive Maxfield and Kuhoo Goyal
"EDA: Where Electronics Begins"
TechBites Interactive, October 1, 2001
ISBN: 0971406308]

Source: "Migration from Simulation to Verification with ModelSim®" by Paul Yanik. *EDA Tech Forum*, 2004 Mar 11, Newton MA

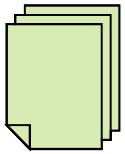
Challenge:
Early Verification Saves Time and Cost



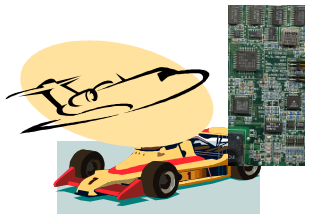
Source: "Migration from Simulation to Verification with ModelSim®" by Paul Yanik. *EDA Tech Forum*, 2004 Mar 11, Newton MA

Advantages of Model-Based Design

Requirements and Specs

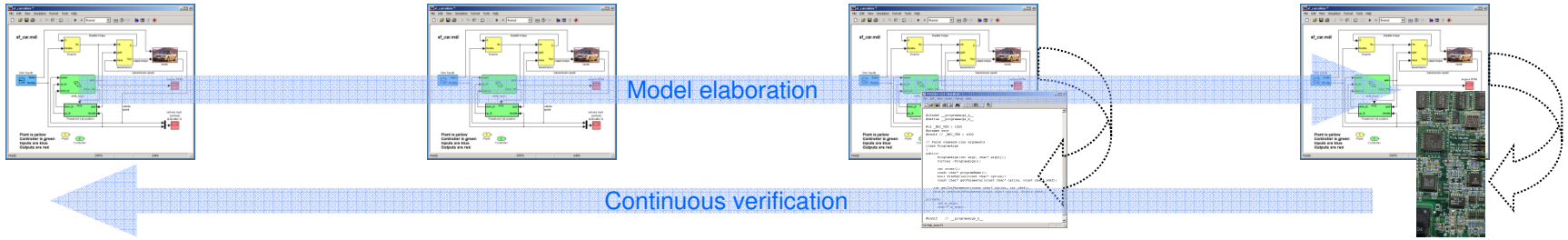


Design



Implementation

Test and Verification

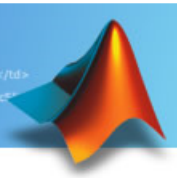


Executable models
- -intuitive
- -unambiguous
- -complete
- -always in sync

Simulation
- -defers or eliminates need for physical prototypes
- -explore broad range of scenarios

Automatic code generation
- -eliminates coding errors
- -saves time
- -promotes design reuse

Test coupled with Design
- -more comprehensive
- -leverages model information
- -detects errors earlier



MBD savings come from

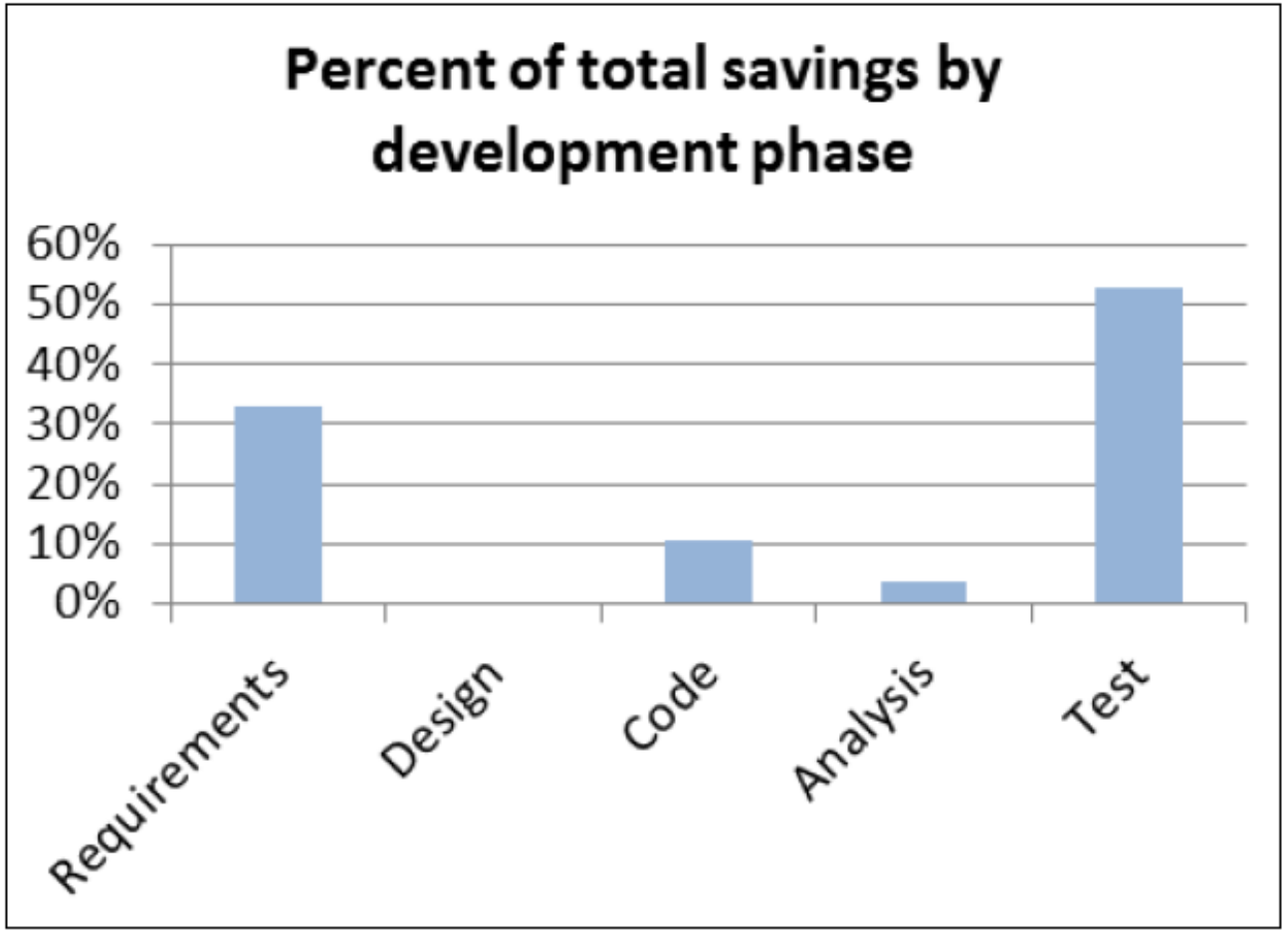
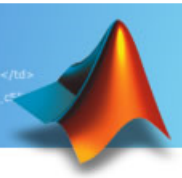
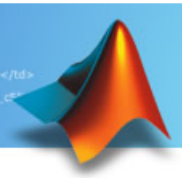


Figure 5. Savings in the requirements and testing phases accounted for most of the total savings.



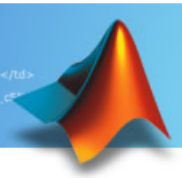
目前MBD被大量用於

- Aerospace and Defense
- Automotive industry
- Industry Automation and Machining
- Other safety critical applications, ex. Railway, Medical, Power System and Nuclear…



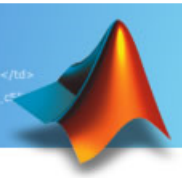
Agenda

- 簡介
- 軟體驗証測試
- Demo



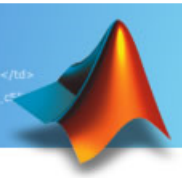
軟體驗証測試

- 測試分類方式
- 依測試方法分類
- 依測試過程分類
- 依程式狀態分類



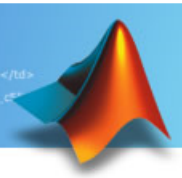
測試分類方式

- 劃分方式不同，有以下分類
  - 方法：黑箱、白箱、灰箱
  - 過程：單元測試、整合測試、系統測試、驗收測試
  - 程式狀態：靜態測試、動態測試



依測試方法分類

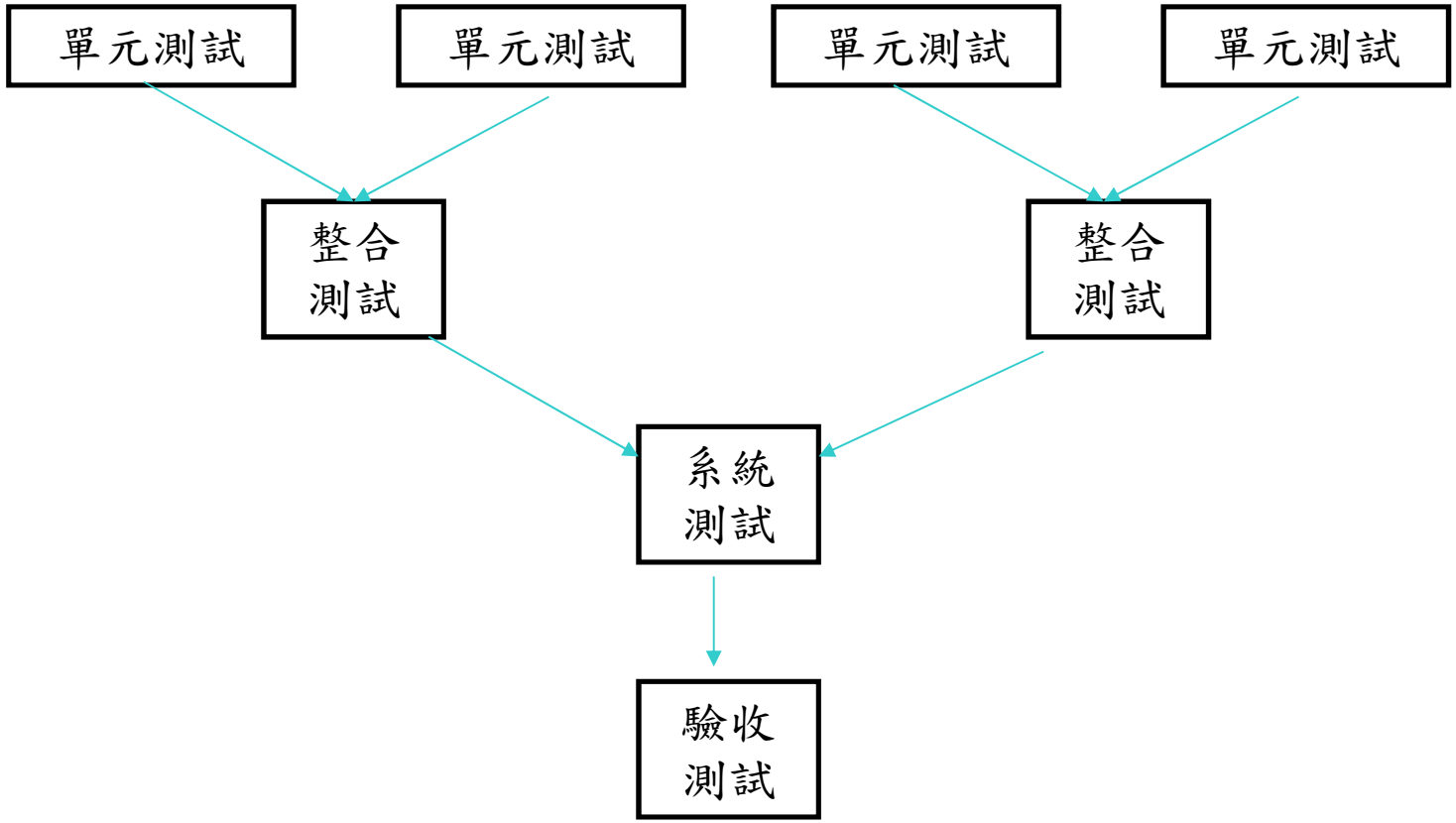
- 依測試方法大致分成三類:
  - 白箱
    - 測試人員直接在軟體程式上進行測試，這類測試包含了語法、邏輯、路徑等測試
  - 黑箱
    - 只管Input與Output，主要以產品的功能為指標的測試方法
  - 灰箱
    - 介於白箱與黑箱之間的測試

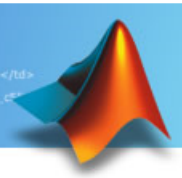


依測試過程分類

- 過程包含4測試步驟：
  - 單元測試
  - 整合測試
  - 系統測試
  - 驗收測試

測試步驟

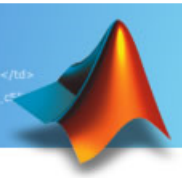




V&V

- 驗證(Verification)
  - 主要用來確定軟體內所定義出來的功能有正確的被實現，而其所執行的功能皆完成
  - Are we building the product right?
  - 單元測試、整合測試、系統測試

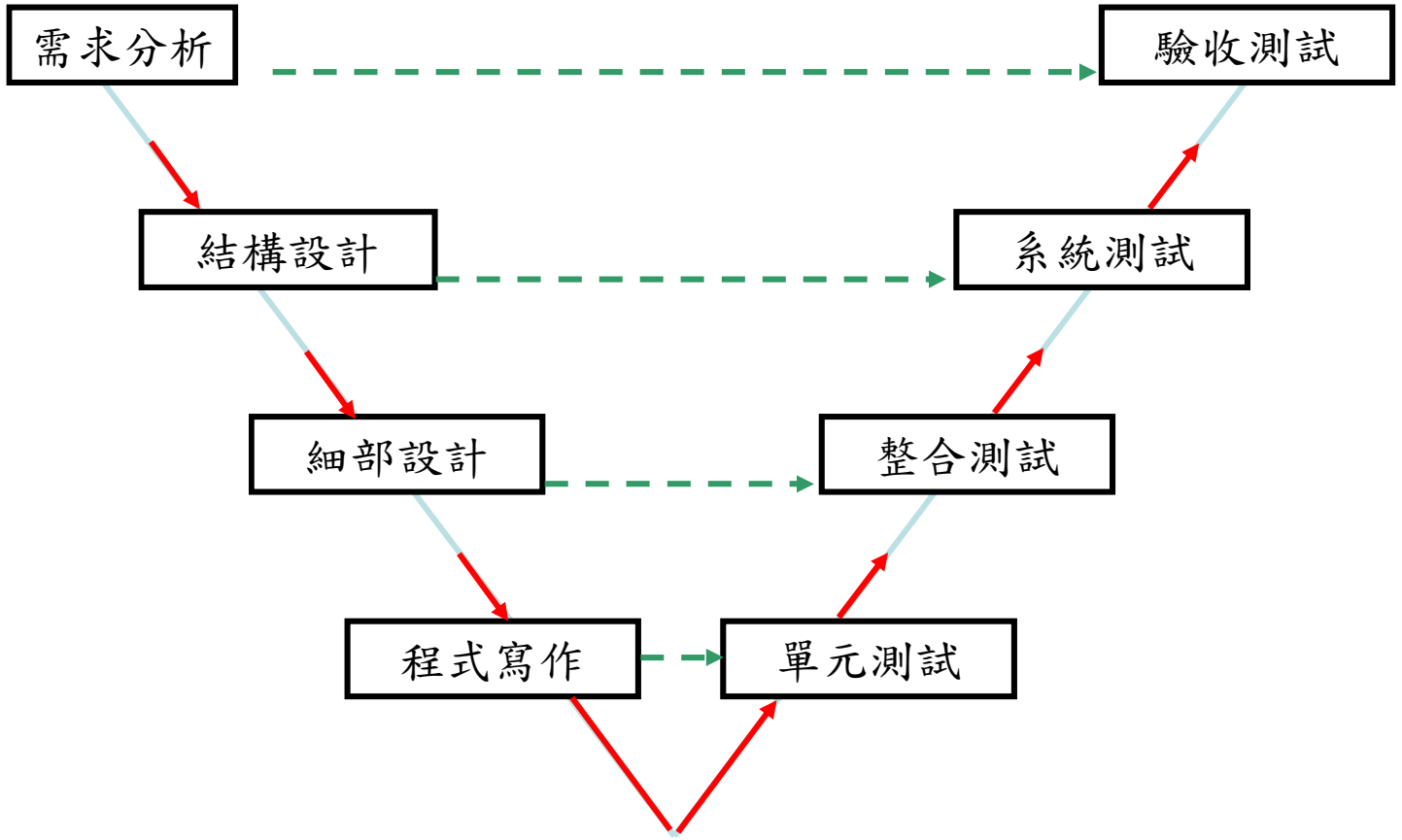
- 驗認(Validation)
  - 確認所被實現的功能是否符合當初使用者所要求，解決使用者想解決的事
  - Are we build the right product?
  - 驗收測試



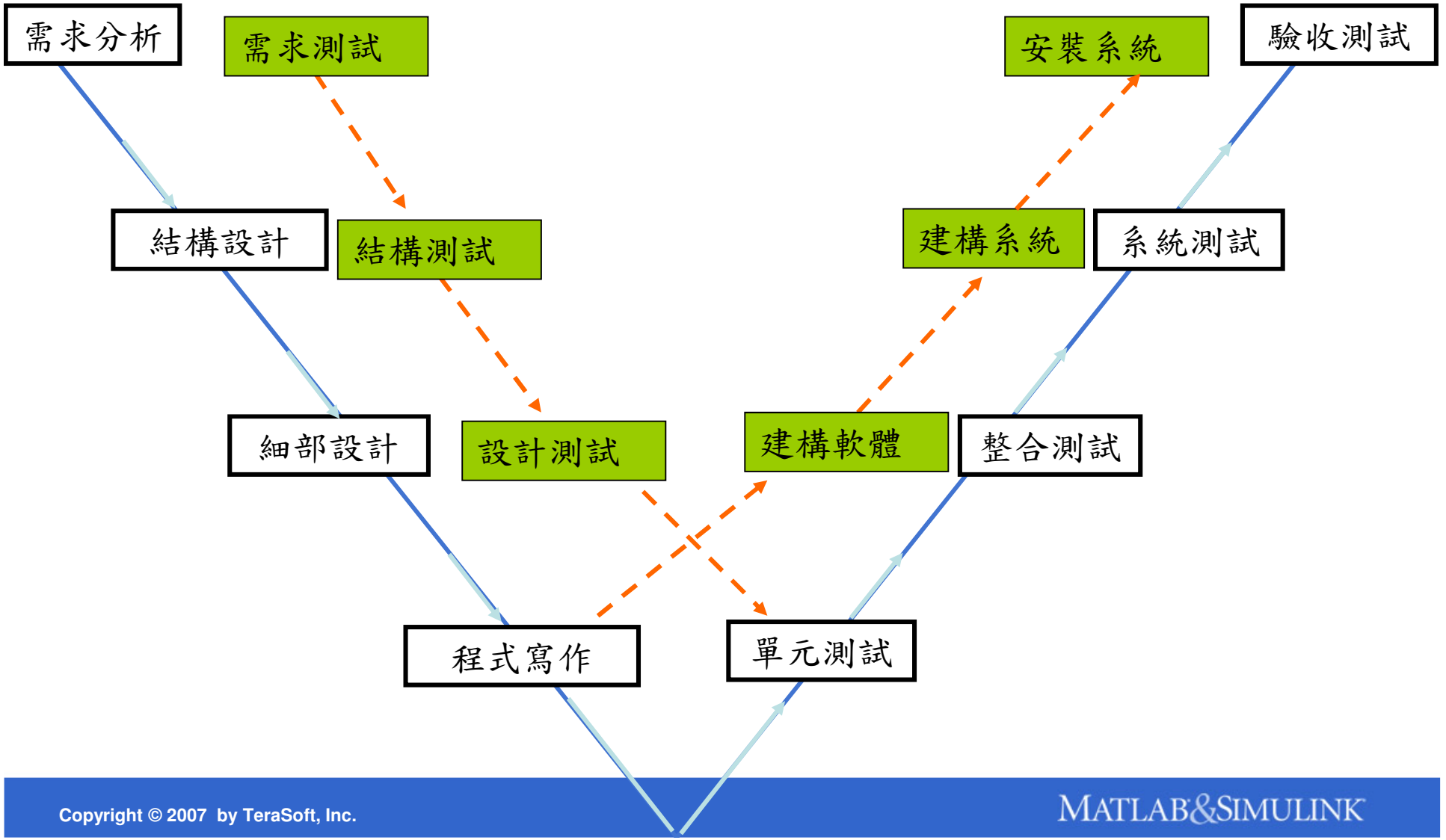
V模型[1]

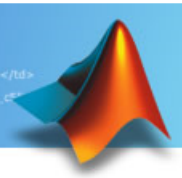
- V模型特色
  - 將測試表示成單一階段，分別為需求分析、高階設計、細部設計、程試寫作
  - V模型描述了數個不同的測試等級及每個層級的生命週期
  - V模型的左側表示的為開發階段的活動，右邊則是與其相對應的測試階段

V模型2



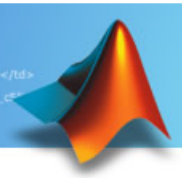
W模型





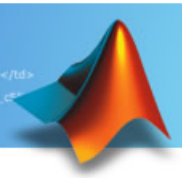
單元測試

- 低階測試
- 獨立測試
- 細節容易被看清楚
- 通常由程式設計師自行測試
- 又稱為元件測試、模組測試、程式測試



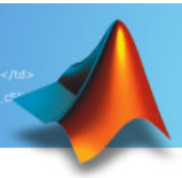
整合測試

- 多個單元測試
- 可用來測試無法單一測試的群組
- 單元間的溝通
- 非功能性觀點
- 測試策略
  - Top-down
  - Bottom-up
  - Functional



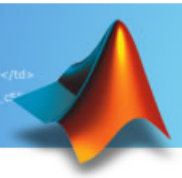
系統測試

- 系統測試是最後的整合步驟
- 功能性
  - 以功能或需求為基底做測試
  - 商業流程為基底做測試
- 非功能性與功能性測試一樣重要
  - 通常不被重視
  - 但此測試是必須被執行的
- 通常由測試人員進行測試



驗收測試

- 確認所被實現的功能是否符合當初使用者所要求，解決使用者想解決的事。

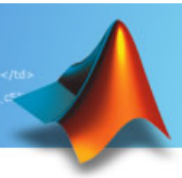


依程式狀態分類

- 依程式執行與否分類：
  - 動態測試

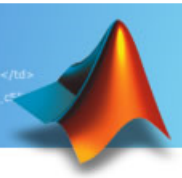
    測試執行中程式
  - 靜態測試 (原始程式分析)

    測試未執行的程式



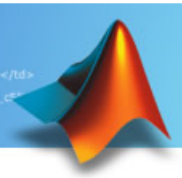
動態測試1

- 單元測試(Unit test)
- 白箱測試(White box test)
- 黑箱測試(Black box test)
- 灰箱測試(Gray box test)
- 功能測試(Function test)
- 系統測試(System test)



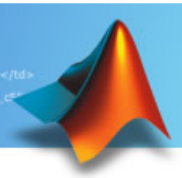
動態測試2

- 驗收測試 (Acceptance test)
- Alpha版測試 (Alpha version test)
- Beta版測試 (Beta version test)
- 平台測試(Platform test)
- 回應測試(Response test)
- 壓力測試(Stress test)
- 自動測試 (Auto Test)
- 回復測試(Recovery Test)



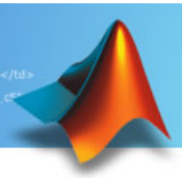
靜態測試 1

- 靜態檢查
- 人工檢查



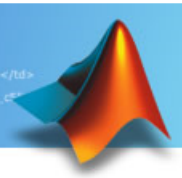
靜態測試 2

- 靜態檢查
  - 資料類型檢查
  - 錯誤路徑檢查
  - 運算式檢查
  - 語法檢查
  - 函數檢查
  - 邏輯檢查



靜態測試 3

- 人工檢查
  - 桌前檢查(Desk Check)
  - 同仁檢查(Peer Review)
  - 走訪(Walkthroughs)
  - 程式碼閱讀(Code reading)



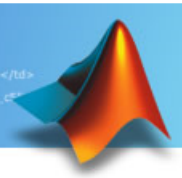
Our Solution in 測試

動態測試

- MIL/SIL/PIL測試 in MBD
- Code Coverage Measurement
- Equivalence Test

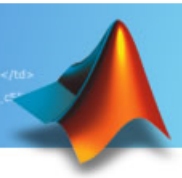
靜態測試

- SLDV Formal Verification Test on Simulink model
- Polyspace Code Verification on C/C++
- MISRA C coding rule check



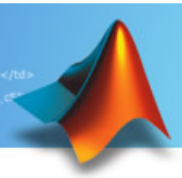
Agenda

- 簡介
- 軟體驗証測試
- Polyspace and Demo



Standards and Regulations

- PolySpace products help you to work with these standards:
  - MISRA-C
  - DO178-B
  - IEC 61508



Where are the errors in this code?

- There are none.
- Proven

where_are_the_errors.c

```
1     int where_are_the_errors(int input)
2     {
3     int x,y,k,l;
4
5     k = input / 100;
6     x = 2;
7     y = k + 5;
8     while (x < 10)
9             {
10            x++;
11            y = y + 3;
12            }
13
14    if ((3*k + 100) > 43)
15            {
16            y++;
17            x = x / (x - y);
18            }
19
20    return x;
21    }
```

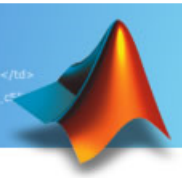
Why prove the absence of run-time errors?

- Exhaustive testing is out of reach, allowing errors to remain in the code.
- Consequences
  - Latent faults in released code
  - Crash during tests or during operations
  - *The Ariane 5 launcher*
- 30-40% of errors found in software are run-time errors.

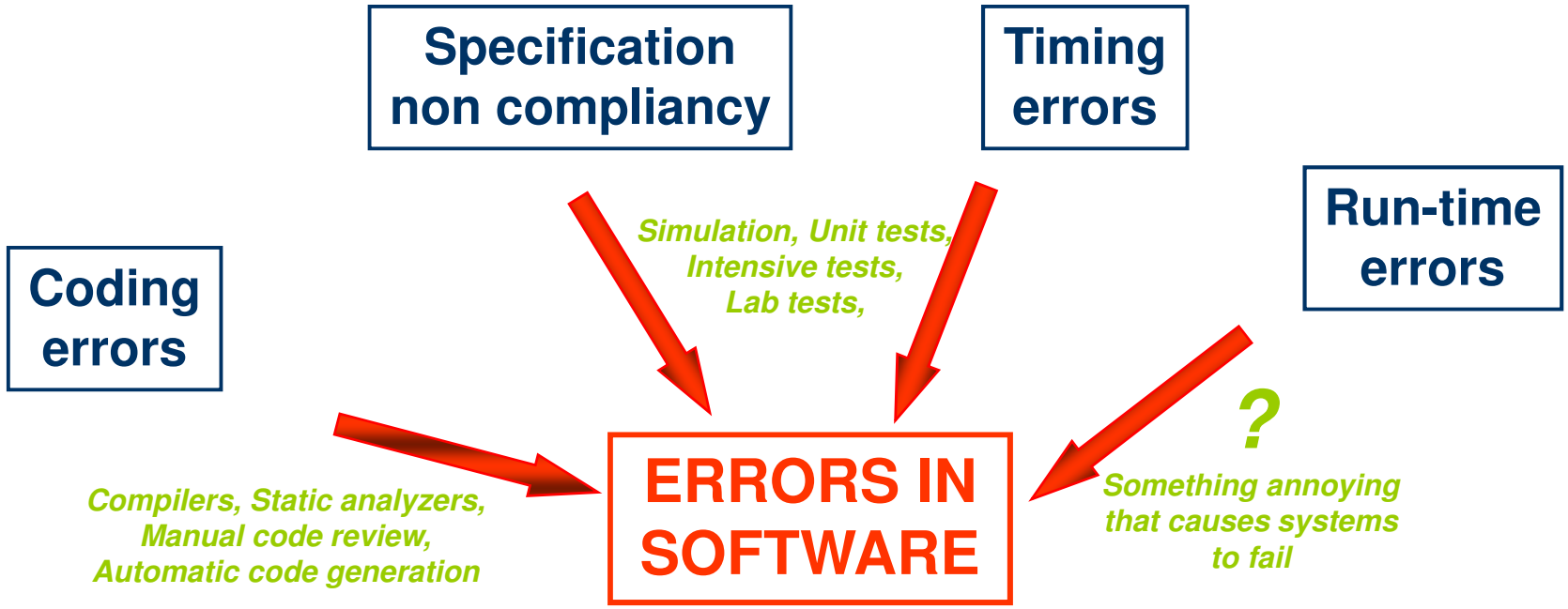
Ariane 501
European launcher.

Industries Using PolySpace™ Products

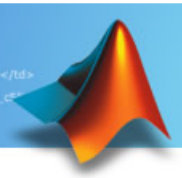
- Aerospace and defense
- Automotive
- Industrial automation and machinery
- Railway transportation
- Consumer electronics
- Medical devices



Old pain :
How to avoid releasing software errors?



Why proving the absence of run-time errors?

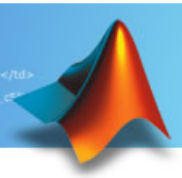


What are run-time errors?

- Also known as latent faults because they rarely manifest directly and frequently
- Effects include crashes, unexpected software behavior during testing and after deployment

Read access to non-initialized data
Out-of-bounds array access
Concurrent access on shared data
Dereferencing through null or out-of-bounds pointers

Incorrect computation
Overflow, underflow
Division by zero
Square root of negative value
Illegal type conversion
Unreachable code
And more…



PolySpace Products

For hand-written code

Formal method:
Abstract Interpretation

Proven

Green
reliable

Red
faulty

Grey
dead

Orange
unproven

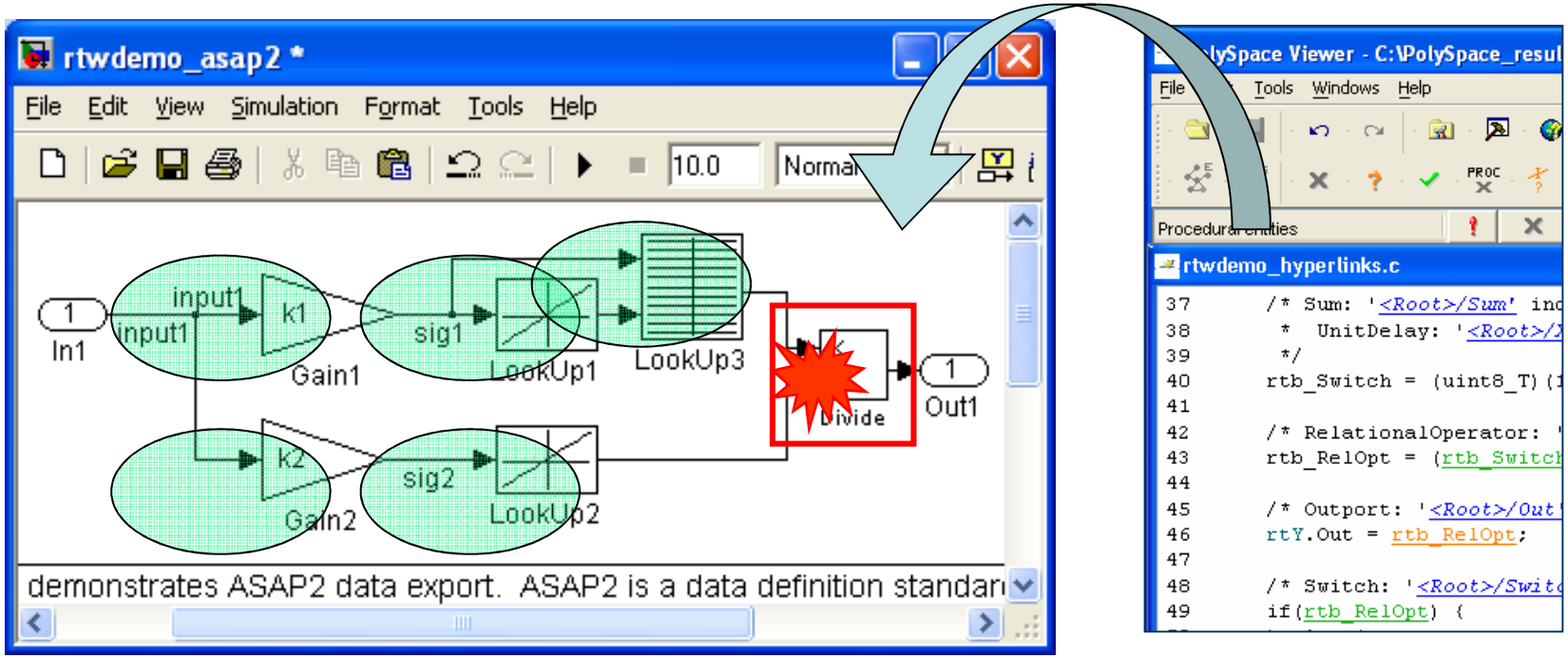
```
static void Pointer_Arithmetic ()
{
  int tab[100];
  int i, *p = tab;

  for(i = 0; i < 100; i++, p++)
    *p = 0;

  if(get_bus_status() > 0)
  {
     if(get_oil_pressure() > 0)
       *p = 5; /* Out of bounds */
     else
       i++;
  }

  i = random_int();
  if (random_int())  *(p-i) = 10;

  if (0<i && i<=100)
  { p = p - i;
    *p = 5;      /* Safe pointer access */
  }
}
```

Results are proven for all possible executions of the code

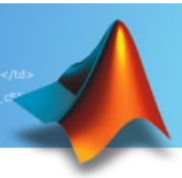
PolySpace Products

For modeling tools and automatic code generation

PolySpace results on generated code are traced back to the model.



Applicable to block-diagrams, Stateflow® charts, and legacy C code

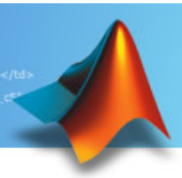


How to prove that no error occurred?

Verifying `x = x / (x - y)`

- Potential run-time errors:
  - Are `x` and `y` initialized?
  - Could a division by 0 occur?
  - Could there be an underflow or overflow on '-', '/' or '=' ?

The following slide focuses on the check for division by 0.



Executing All Test Cases

2 inputs: INT on 32 bits

1 file

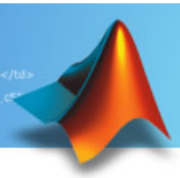
How many possible test cases?

… 2^{32} x 2^{32} = 2^{64}

If you execute 1,000 tests, it means you verify 0.00000000000005% of the possible executions.

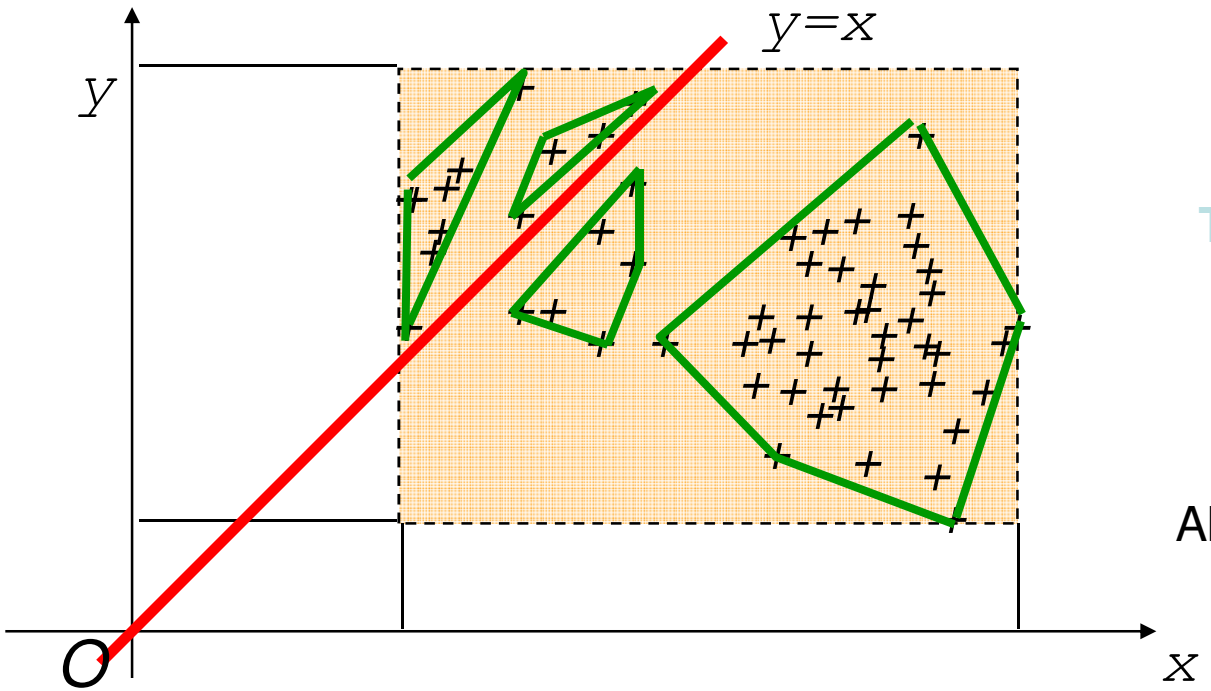
Usable for finding (functional) bugs…

…but not effective for run-time error verification



No execution
No simulation
No test cases to write

Verifying `x =/x / (x - y)`

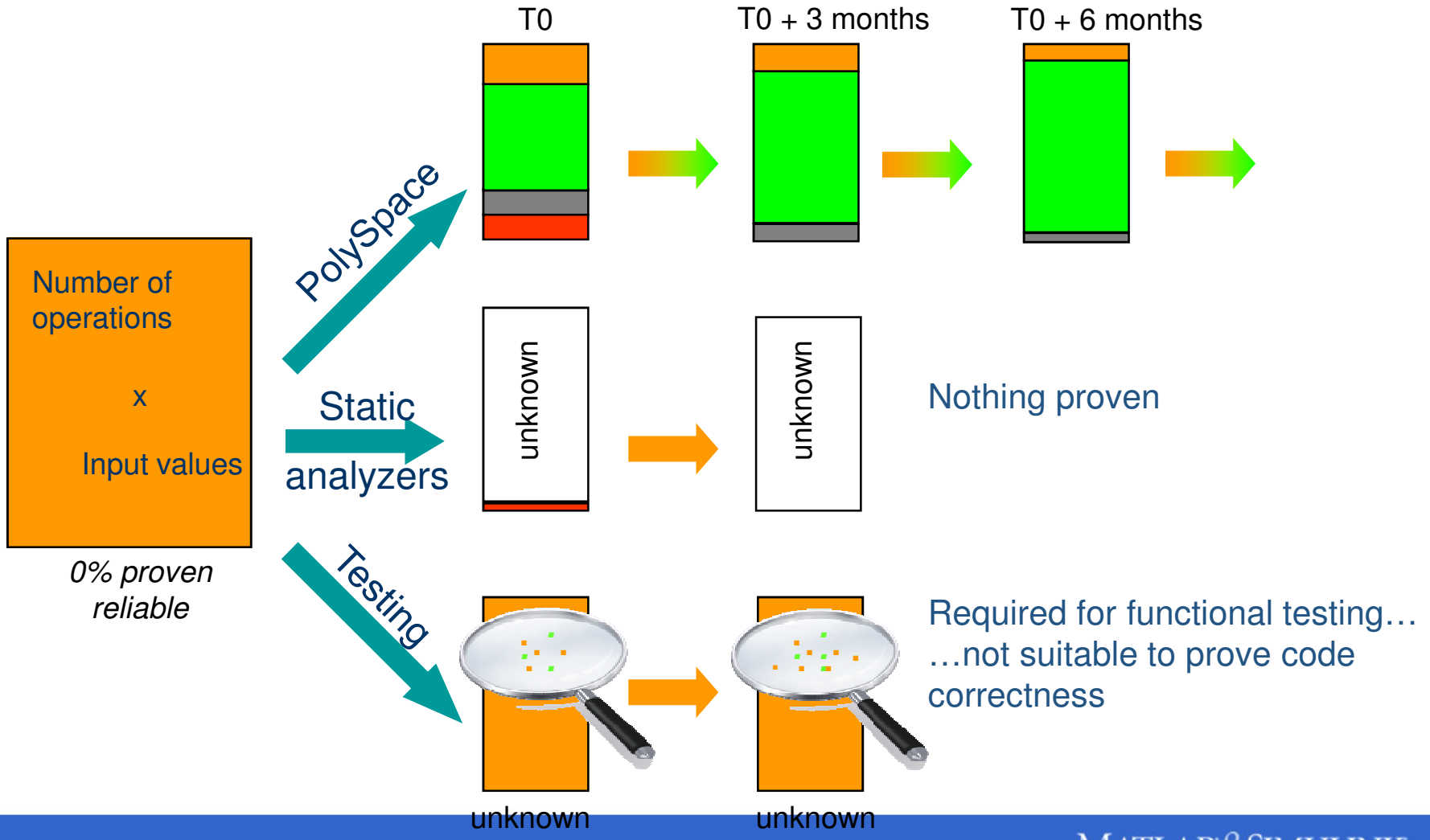


Type analysis

Abstract interpretation

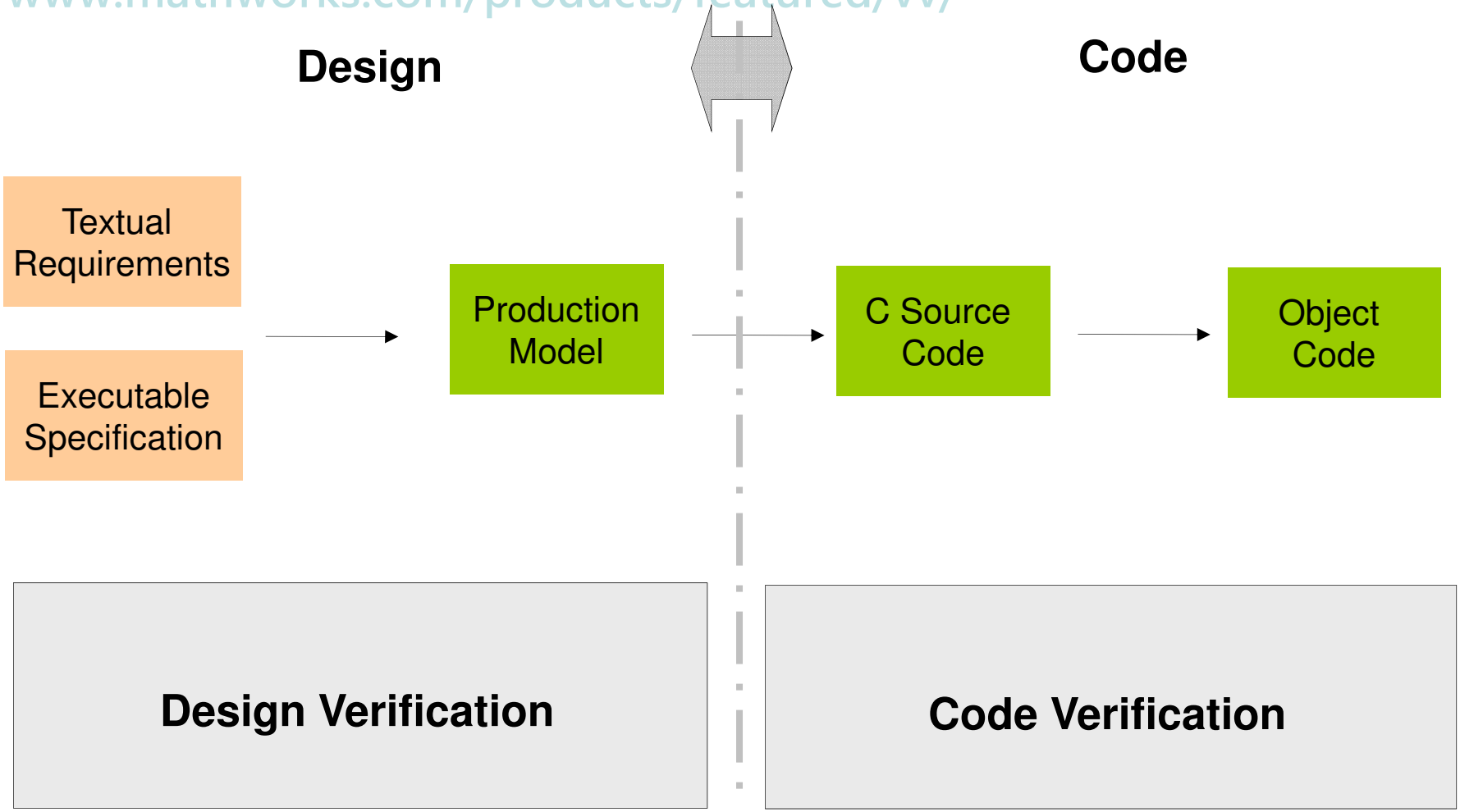
What color is your code today?

How do you prove code correctness?



Component Verification

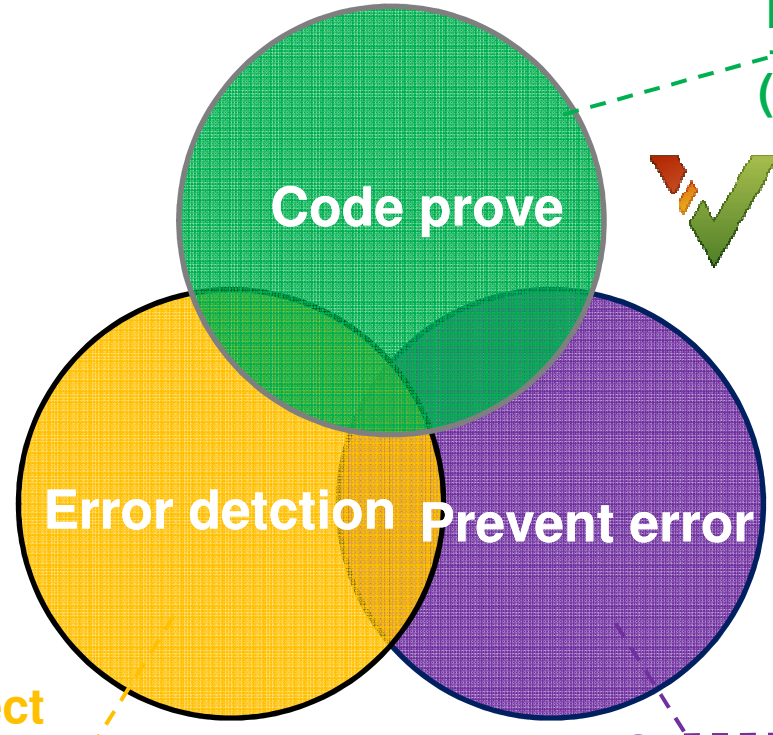
www.mathworks.com/products/featured/vv/



Polyspace provide code reliability

Auto code generation or hand code detection

File base or project base verification



Formal verification
(No False negative)

Polyspace Code Prover

Bug detect
(False negative)

Polyspace Bug Finder

Coding rule
Compiler warning code matrics

Polyspace Bug Finder™
Polyspace Code Prover™

Thank You For Your Attention