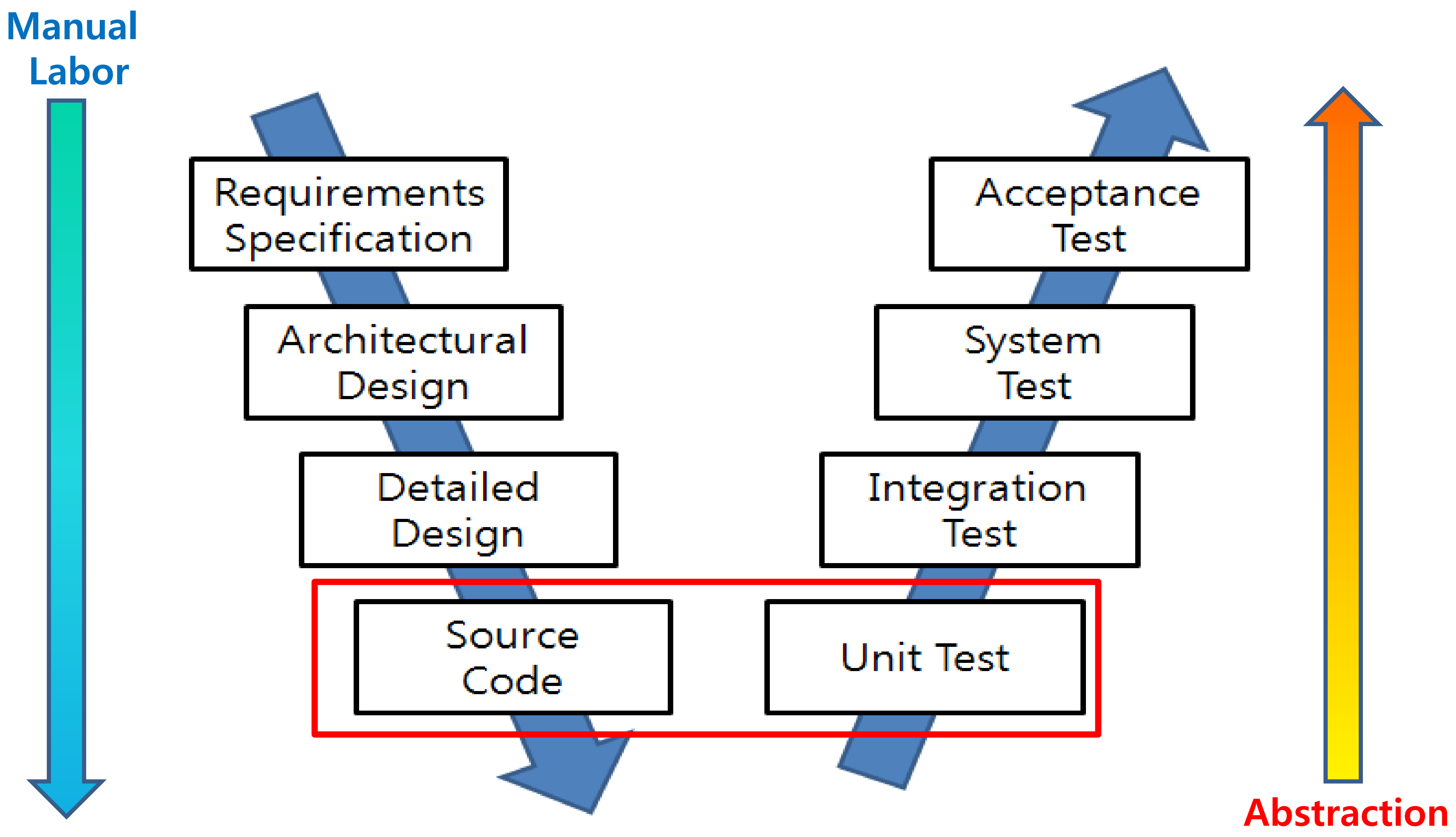
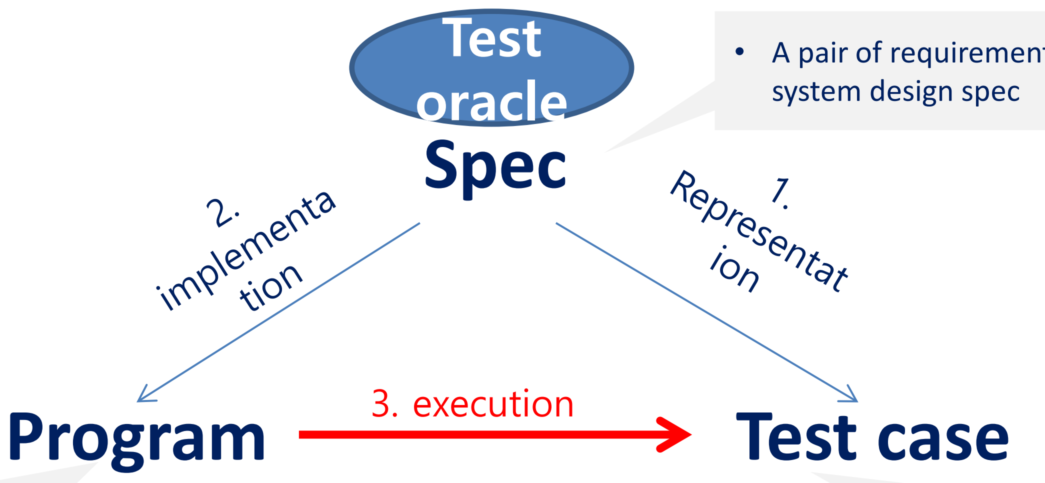
SW Development and Testing Model (a.k.a. V model)



Foundation of Software Testing



- A pair of requirement spec and system design spec

- Code that implements the system specification and satisfies the requirements

- A pair of test input and expected test output for the input

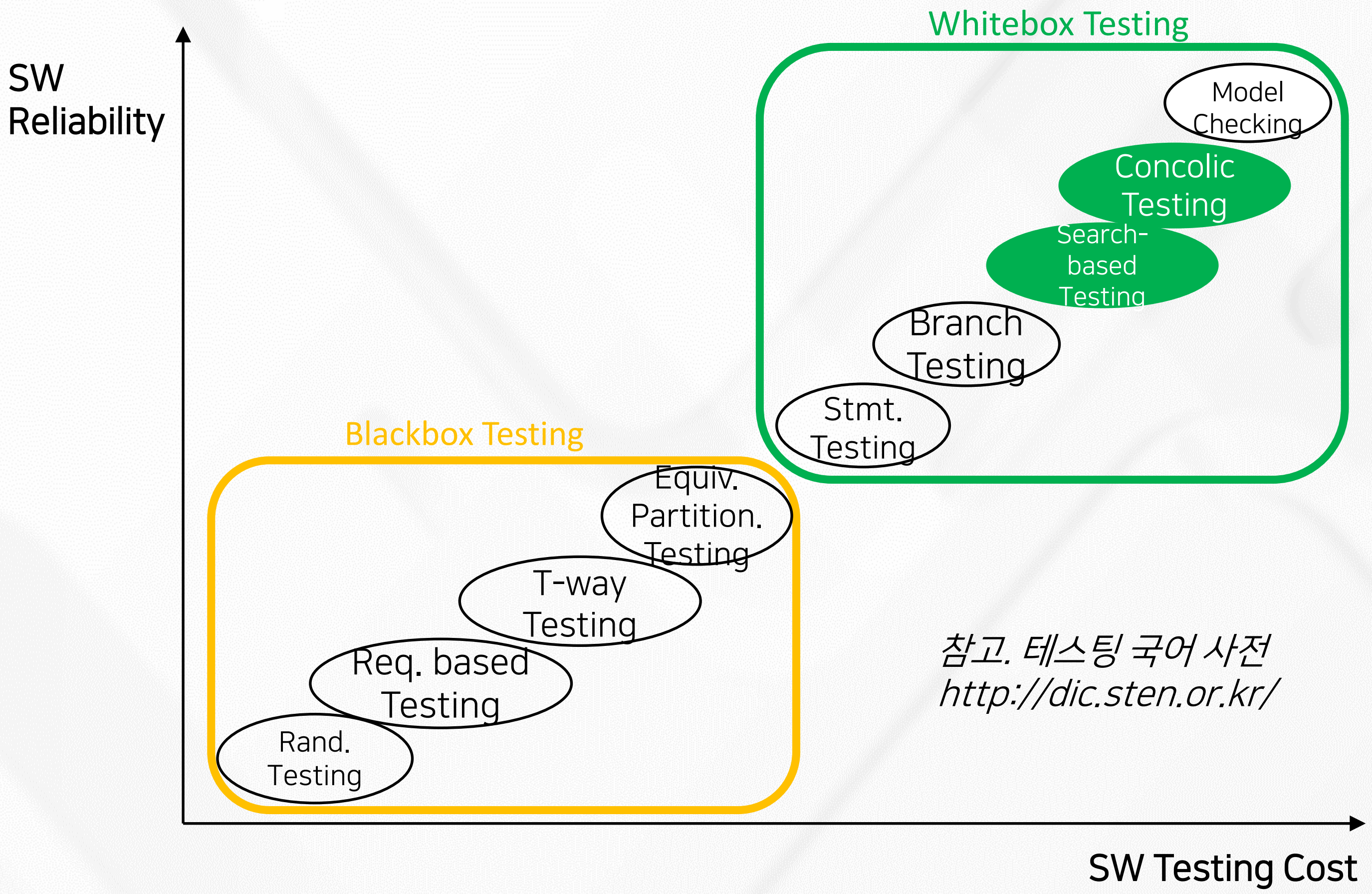
Multiple targets for software testing

1. Does the test cases represent the requirement spec correctly?
   → Scenario based testing (black-box testing)
2. Is the design spec implemented as program correctly?
   → Model-based testing (grey-box testing)
3. Does the program satisfy test cases correctly?
   → Code-based testing (white-box testing)

Black Box Testing

- A main goal of testing is <u>to generate multiple test cases</u>, one of which may reveal a bug.
- Black box testing concerns only input/output of a target program (i.e., ignore program code)
  - Ex1. Requirement specification based testing
  - Ex2. Random (input generation) testing
  - Ex3. Category partitioning method
  - Ex4. T-way testing
- Advantage of black box testing
  - Intuitive and simple
  - Requires little expertise on program/code analysis techniques
  - Requires less effort compared to white-box testing
    - cheaper but less effective

Various SW Testing Techniques w/ Different Cost and Effectiveness



참고. 테스팅 국어 사전
http://dic.sten.or.kr/

Requirement based Blackbox Testing VS Logic based Whitebox Testing

	Black Box Test	White Box Test
Def.	Functional test based on the requirement specification	Logical analysis based on target source code
View point	User	Developer
Bug detection criteria	Interface and/or performance problem	Logical problems
Verification & Validation level	High (user)	Low (testing)
Target bugs	Observable external errors	Internal errors due to logic problem, uncovered stmt.
Technique	Category partition, boundary value analysis, etc.	Loop, control structure test
Bug detection ability	Low	High
# of TC	Small	Large
Application	Beta test	Alpha test

도리의 디지털라이프 블로그에서 발췌/편집
http://blog.skby.net/%EB%B8%94%EB%9E%99%EB%B0%95%EC%8A%A4-%ED%85%8C%EC%8A%A4%ED%8A%B8-%ED%99%94%EC%9D%B4%ED%8A%B8%EB%B0%95%EC%8A%A4-%ED%85%8C%EC%8A%A4%ED%8A%B8/

If a requirement is specified as an `assert` statement requirement can be tested through whitebox texting

Example of Blackbox Testing Technique:

The Category-Partition Method for Specifying and Generating Functional Tests
(Thomas J. Ostrand and Marc J.Balcer [CACM,1988])

The original slides from Prof. Shmuel Sagiv's lecture notes
msagiv@post.tau.ac.il

<u>Content:</u>

The goal of functional testing

- To find discrepancies between the **actual beha vior** of the implemented system’s function and the **desired behavior** as described in the syste m’s functional specification.

How to achieve this goal ?

- Tests have to be execute for all the system fun ctions.

- Tests have to be designed to maximize the cha nces of finding errors in the software.

Functional test can be derived **from 3 sources:**

1. The software specification.

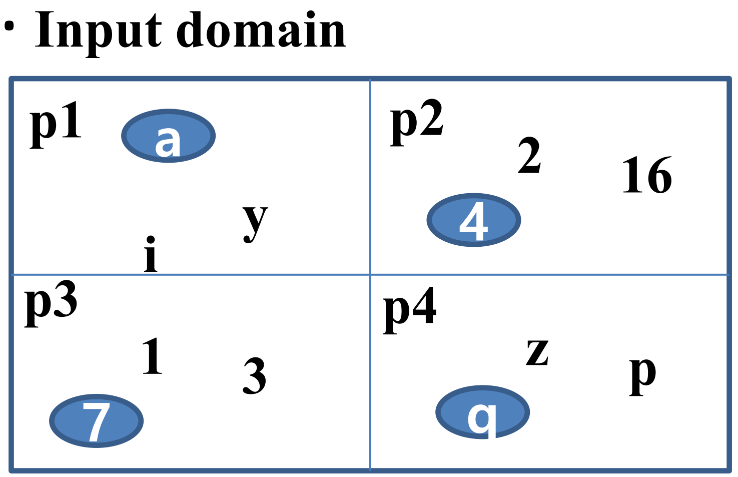
2. Design information.

3. The code itself.

Partition - The standard approach

- The main idea is to **partition the input domain** of function being tested, and then **select test data for each class** of the partition.

- The problem of all the existing techniques is the **lack of systematic steps.**



A strategy for test case generation

1. Transform the system’s specification to be more concise and structured.

2. Decompose the specification into functional unit - to be tested independently.

3. Identify the parameters and environment conditions.

A strategy for test case generation (cont)

4. Find categories that characterize each paramet er and environment condition.

5. Every category should be partitioned into distin ct choices .

$\Downarrow$

formal test specification

A strategy for test case generation (cont)

?

6.	test frames -	set of choices, one from each category.
	⇓	
	test cases -	test frame with specific values for each choices.
	⇓	
	test scripts -	sequence of test cases.

Example

Command: find

Syntax: find <pattern> <file>

Function: The find command is used to locate one or

more instance of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the ma ximum length of a line in the file .To include a blank in the pattern, the entire p attern must be enclosed in quotes (“).To include quotation mark in the pattern , two quotes in a row (“ “) must be used.

Example:

find john myfile

display lines in the file **myfile** which contain **john**

find “john smith” myfile

display lines in the file **myfile** which contain **john smith**

find “john”” smith” myfile

display lines in the file **myfile** which contain **john” smith**

Categories

Parameters:

Pattern size:

4
- empty
- single character
- many character
- *longer than any line in the file*

Quoting:

3
- pattern is quoted
- pattern is not quoted
- pattern is improperly quoted

Embedded blanks:

3
- no embedded blank
- one embedded blank
- several embedded blanks

Embedded quotes:

3
- no embedded quotes
- one embedded quotes
- several embedded quotes

File name:

3
- good file name
- no file with this name
- omitted

Total Tests frames:
1944 (=4*3*3*3*3*3*2)

Environments:

Number of occurrence of pattern in file:

3
- none
- exactly one
- more than one

Pattern occurrences on target line:

2
- one
- more than one

Adding Constraints between Categories to Reduce #of TC'S

Parameters:

Pattern size:

empty	[property Empty]
single character	[property NonEmpty]
many character	[property NonEmpty]
longer than any line in the file	[property NonEmpty]

Quoting:

pattern is quoted	[if NonEmpty] [property Quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[if NonEmpty]

Embedded blanks:

no embedded blank	[if NonEmpty]
one embedded blank	[if NonEmpty and Quoted]
several embedded blanks	[if NonEmpty and Quoted]

Embedded quotes:

no embedded quotes	[if NonEmpty]
one embedded quotes	[if NonEmpty]
several embedded quotes	[if NonEmpty]

File name:

good file name

no file with this name

omitted

Total Tests frames: 678

Environments:

Number of occurrence of pattern in file:

none	[if NonEmpty]
exactly one	[if NonEmpty] [property Match]
more than one	[if NonEmpty] [property Match]

Pattern occurrences on target line:

one	[if Match]
more than one	[if Match]

Parameters:

Pattern size:

empty	[property Empty]
single character	[property NonEmpty]
many character	[property NonEmpty]
longer than any line in the file	[error]

Quoting:

pattern is quoted	[property quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[error]

Embedded blanks:

no embedded blank	[if NonEmpty]
one embedded blank	[if NonEmpty and Quoted]
several embedded blanks	[if NonEmpty and Quoted]

Embedded quotes:

no embedded quotes	[if NonEmpty]
one embedded quotes	[if NonEmpty]
several embedded quotes	[if NonEmpty]

File name:

good file name	
no file with this name	[error]
omitted	

Total Tests frames: 125

Environments:

Number of occurrence of pattern in file:

none	[if NonEmpty]
exactly one	[if NonEmpty] [property Match]
more than one	[if NonEmpty] [property Match]

Pattern occurrences on target line:

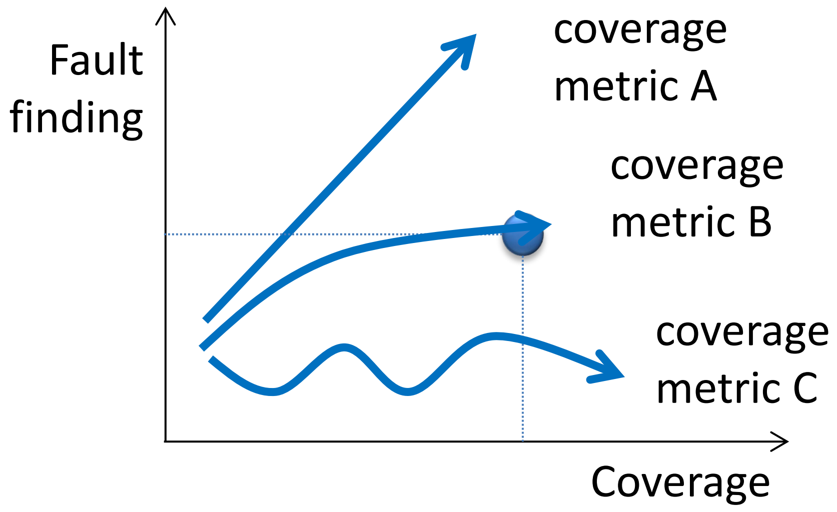
one	[if Match]
more than one	[if Match]

White Box Testing (1/2)

- White box testing concerns program code itself
- Many different viewpoints on “program code”
  1) program code as a graph (i.e., structural coverage)
  2) program code as a set of logic formulas (i.e., logical coverage)
  3) program code as a set of execution paths (i.e., behavioral/dynamic coverage)
- Advantages:
  – More effective than blackbox testing in general
  – Can measure the testing progress quantitatively based on coverage achieved
- Should be used with blackbox testing together for maximal bug detection capability
  – Blackbox testing and whitebox testing often explore different segments of target program space

White Box Testing (2/2)

- Coverage is a good predictor/indicator of testing effectiveness
  - Utilizing correlation between structural coverage and fault detection ability



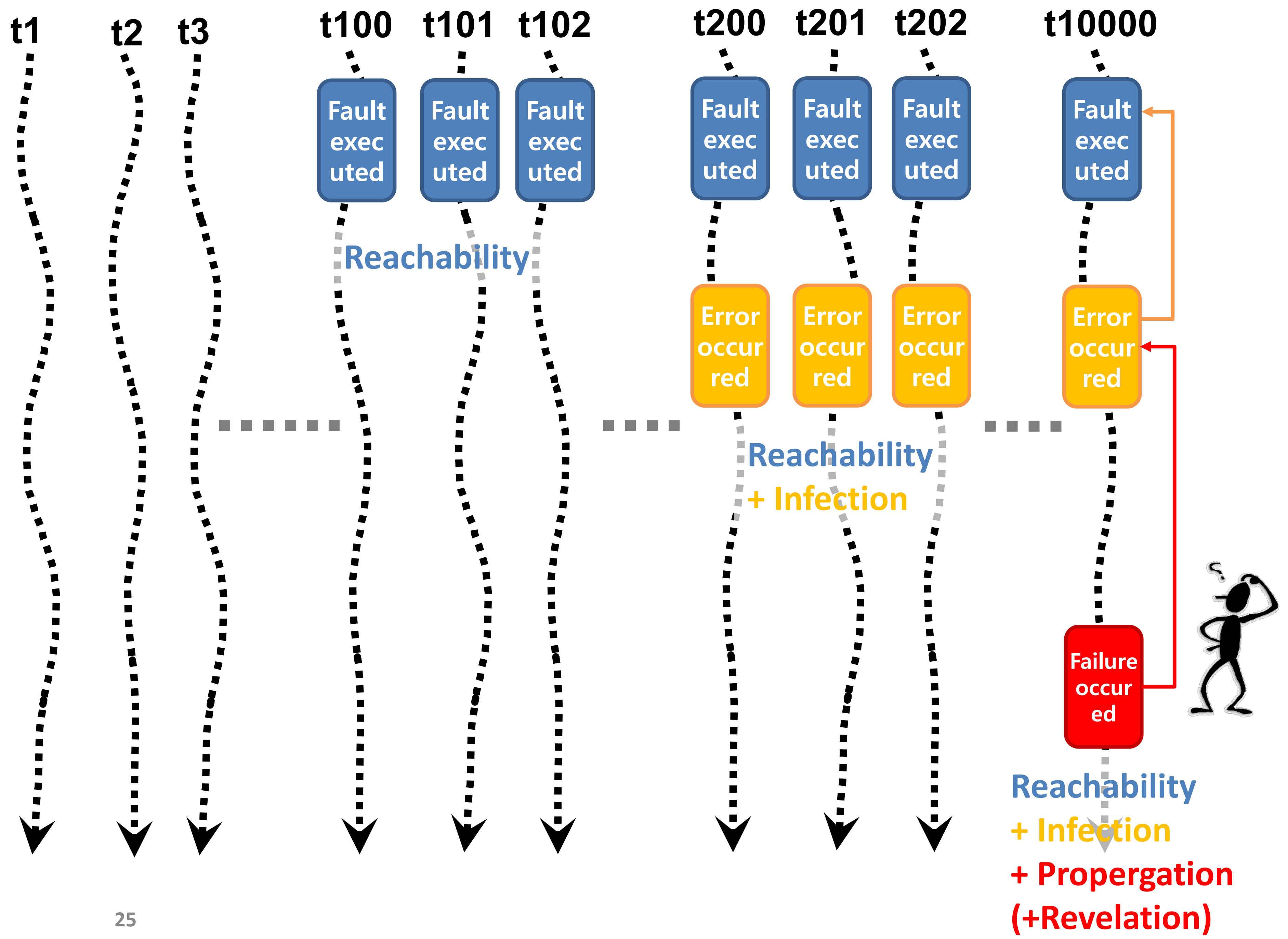
Bug Observability/Detection Model: **R**eachability, **I**nfection, **P**ropagation, and **R**evelation (RIPR)

- Terminology
  - Fault: static defect in a program text (a.k.a a bug)
  - Error: dynamic (intermediate) behavior that deviates from its (internal) intended goal
    - A fault causes an error (i.e. error is a symptom of fault)
  - Failiure: dynamic behavior which violates a ultimate goal of a target program
    - Not every error leads to failure due to error masking or fault tolerance

- Graph coverage
  - Test requirement satisfaction == Reachability
    - the fault in the code has to be reached
- Logic coverage
  - Test requirement satisfaction

    == Reachability +Infection
    - the fault has to put the program into an error state.
      - Note that a program is in an error state does not mean that it will always produce the failure
- Mutation coverage
  - Test requirement satisfaction

    == Reachability +Infection + Propagation
    - the program needs to exhibit incorrect outputs
- Furthermore, test oracle plays critical role to reveal failure of a target program (Revelation)


t1
t2
t3
t100
t101
t102
t200
t201
t202
t10000
Fault exec uted
Fault exec uted
Fault exec uted
Fault exec uted
Fault exec uted
Fault exec uted
Fault exec uted
Reachability
Error occur red
Error occur red
Error occur red
Error occur red
Reachability
+ Infection
Failure occur ed
Reachability
+ Infection
+ Propergation
(+Revelation)