Iniciação à investigação científica

Ideias (aparentemente) absurdas e outras histórias de investigação científica

O que eu queria mesmo falar:

injecção de falhas de software

Henrique Madeira

University of Coimbra, Portugal

The research "business"
(my personal view)

- Identification of research problems (the key issue)
- Relevance of the research
- New/different is not enough
- Proving/showing that your proposal is better.
- The research approach (the experimental view)
- The Pygmalion effect
- Convincing other researchers and the world (getting papers accepted)

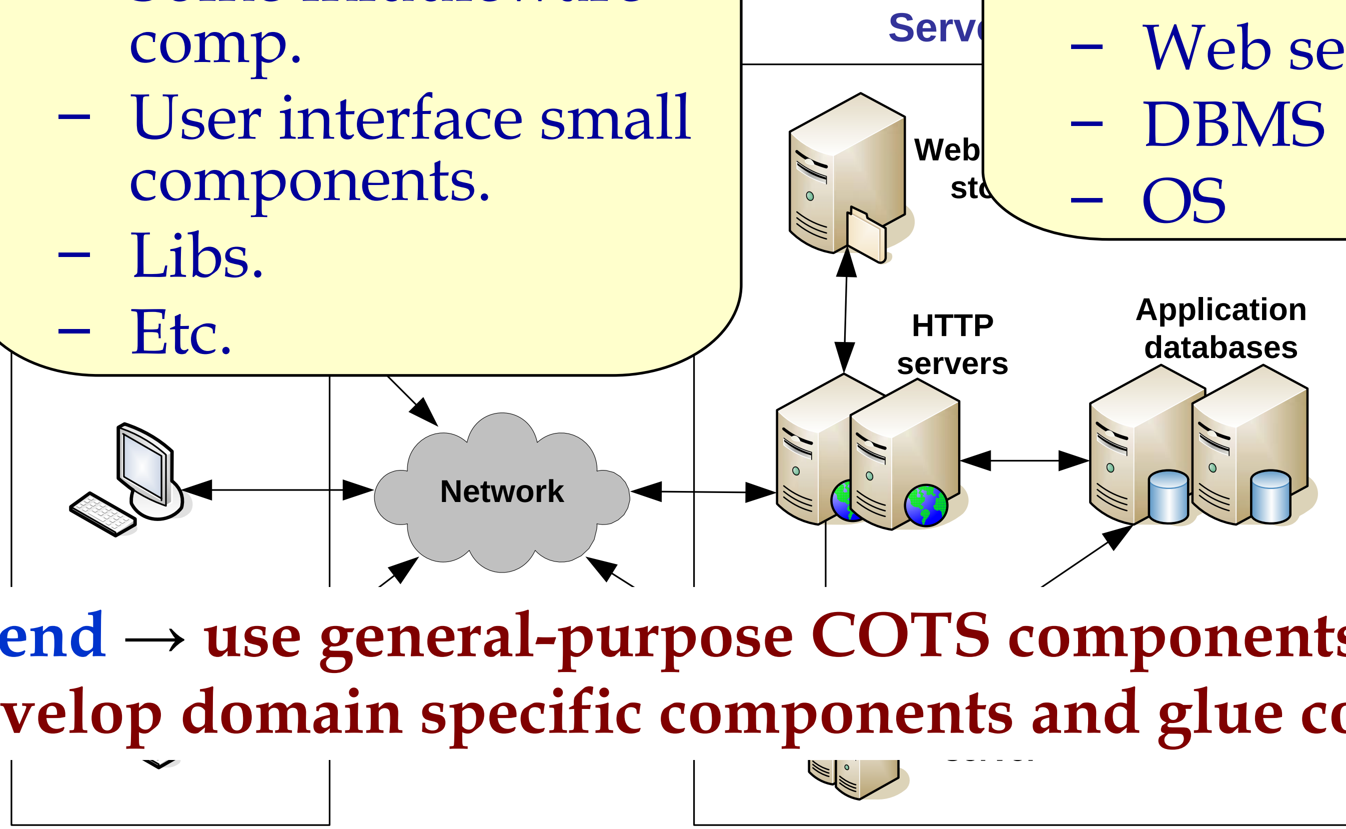
Background: COTS based software development

Fine grain COTS:

- Some middleware comp.
- User interface small components.
- Libs.
- Etc.

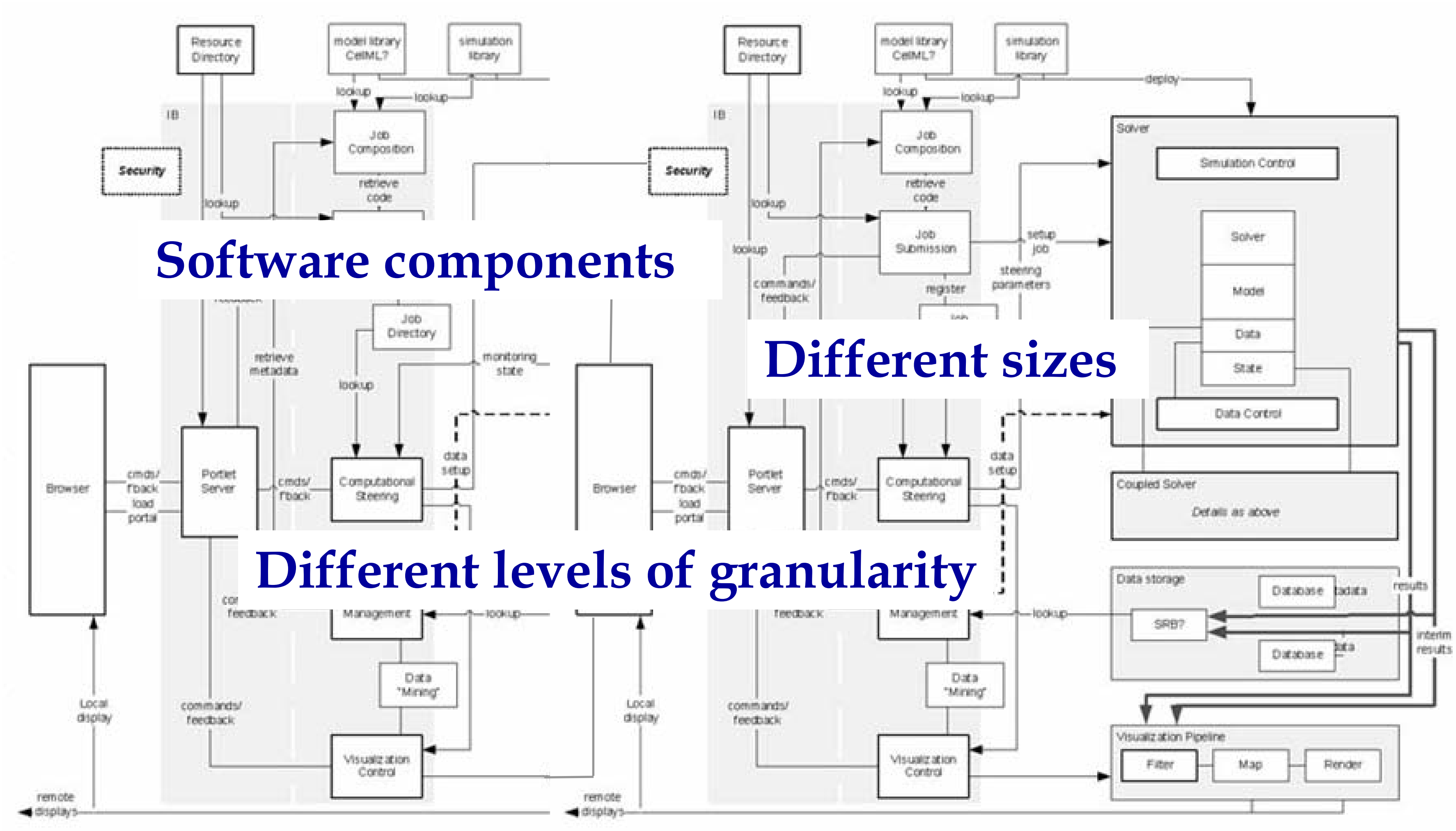
Coarse grain COTS:

- Middleware comp.
- Web servers
- DBMS
- OS

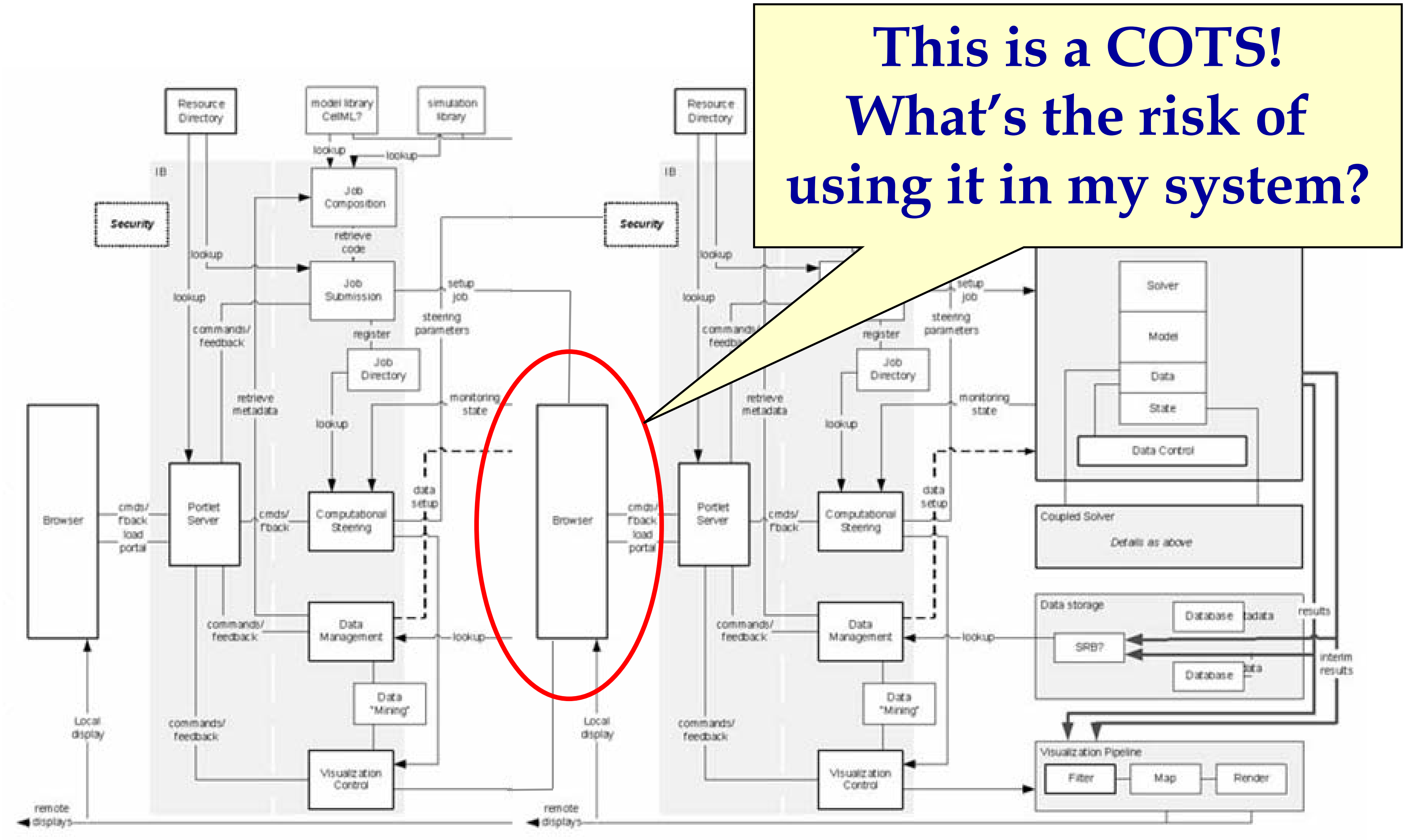


Trend → **use general-purpose COTS components and develop domain specific components and glue code.**

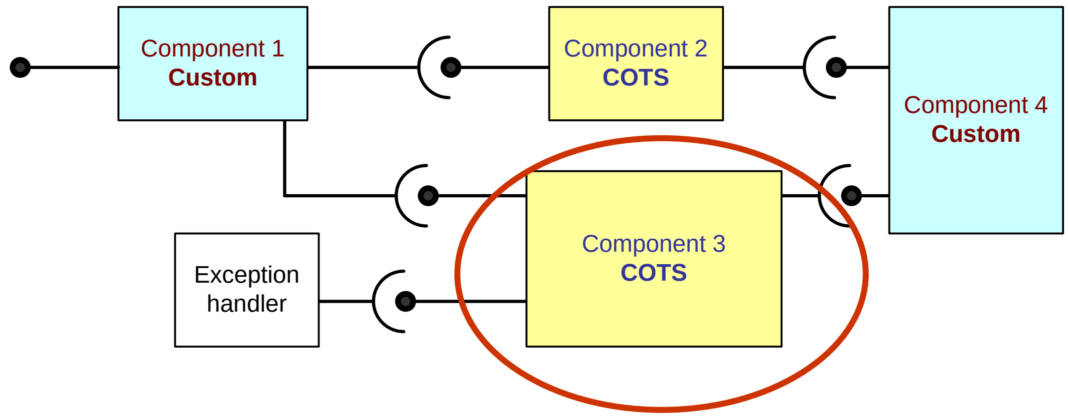
Research problem



Research problem



Research starting idea



Question:

What's the risk of using Component 3 in my system?

Risk = prob. of bug * impact of bug activation

Software complexity metrics

Injection of software faults

Research problem
(focusing the problem)

- How to inject realistic and representative software faults (bugs)
  - **Which faults are realistic?**
  - **Which ones are the most representative?**

- **Goals:**
  - Design the first software fault injector
  - Define practical methods/techniques for the evaluation of systems behavior in presence of faulty software components
  - Propose an experimental approach to estimate risk of using software.

What is a software fault?
(narrowing the problem)

Software development process (in theory...)

Requirements **OK**

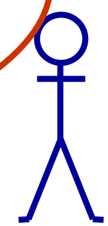
Specification **OK**

Design

Code development

Test

Deployment



The requirements + specification
are correct but the deployed code is not

Correctness from the end user point of view: too vague

Characterization of software faults (using previous work from IBM)

A SW fault is characterized by the change in the code that is necessary to correct it (Orthogonal Defect Classification from IBM).

Defined according two parameters:

- **Fault trigger** conditions that make the fault to be exposed
- **Fault type** type of mistake in the code

Types of software faults (ODC)

- **Assignment** values assigned incorrectly or not assigned
- **Checking** missing or incorrect validation of data, or incorrect loop, or incorrect conditional statement
- **Timing/serialization** missing or incorrect serialization of shared resources
- **Algorithm** incorrect or missing implementation that can be fixed without the need of design change
- **Function** incorrect or missing implementation that requires a design change to be corrected

Which are the most representative software faults?

- Field data on real software errors is the most reliable information source on which faults should be injected
- Typically, this information is not made public
- Open source projects provide information on past (discovered) software faults

Open source field data survey

Programs	Description	# faults
CDEX	CD Digital audio data extractor.	**11**
Vim	Improved version of the UNIX vi editor.	**249**
FreeCiv	Multiplayer strategy game.	**53**
pdf2h	pdf to html format translator.	**20**
GAIM	All-in-one multi-protocol IM client.	**23**
Joe	Text editor similar to Wordstar®	**78**
ZSNES	SNES/Super Famicom emulator for x86.	**3**
Bash	GNU Project's Bourne Again SHell.	**2**
LKernel	Linux kernels 2.0.39 and 2.2.22	**93**
Total faults collected		**532**

Characterization of software faults through an additional step over ODC

- ***Hypothesis*:**

  Faults are considered as programming elements (language constructs) that are either:

  - **Missing**

    E.g. Missing part of a logical expression

  - **Wrong**

    E.g. Wrong value used in assignment

  - **Extraneous**

    E.g. Surplus condition in a test

Fault characterization on top of ODC

ODC types	Nature	Examples
Assign	Missing	A variable was not assigned a value, a variable was not initialized, etc
	Wrong	A wrong value (or expression result, etc) was assigned to a variable
	Extraneous	A variable should not have been subject of an assignment
Checking	Missing	An "if" construct is missing, part of a logical condition is missing, etc
	Wrong	Wrong "if" condition, wrong iteration condition, etc
	Extraneous	An "if" condition is superfluous and should not be present
Interface	Missing	A parameter in a function call was missing
	Wrong	Wrong information was passed to a function call (value, expression result etc)
	Extraneous	Surplus data is passed to a function (one param. too many in function call)
Algorithm	Missing	Some part of the algorithm is missing (e.g. function call, a iteration construct)
	Wrong	Algorithm is wrongly coded or ill-formed
	Extraneous	The algorithm has surplus steps; A function was being called
Function	Missing	New program modules were required
	Wrong	The code structure has to be redefined to correct functionality
	Extraneous	Portions of code were completely superfluous

Fault distribution across ODC types

ODC Type	Number of faults	ODC distribution (our work)	ODC distribution (prev. research IBM)
Assignment	118	**22.1** %	21.98 %
Checking	137	**25.7 %**	17.48 %
Interface	43	**8.0 %**	8.17 %
Algorithm	198	**37.2 %**	43.41 %
Function	36	**6.7 %**	8.74 %

- There is a clear trend in fault distribution
  - Previous research (not open source) confirms this trend
  - Some faults are more representative (i.e. more interesting) than others: **Assignment**, **Checking**, **Algorithm**

Fault nature characterization across ODC

ODC types	Nature	# faults
Assign.	Missing	44
	Wrong	64
	Extraneous	10
Check.	Missing	90
	Wrong	47
	Extraneous	0
Interf.	Missing	11
	Wrong	32
	Extraneous	0
Alg.	Missing	155
	Wrong	37
	Extraneous	6
Func.	Missing	21
	Wrong	15
	Extraneous	0

- ***Missing*** and ***wrong*** elements are the most frequent ones
- This trend is consistent across the ODC types tested

Fault characterization across programs

Fault nature	CDEX	Vim	FCiv	Pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	Total
Missing cons.	3	157	35	11	17	34	1	0	63	321
Wrong cons.	8	85	18	9	6	41	2	2	24	195
Extraneous cons	0	7	0	0	0	3	0	0	6	16

1 – ***Missing constructs*** faults are the more frequent ones

2 – ***Extraneous constructs*** are relatively infrequent

3 – This trend is **consistent** across the programs tested

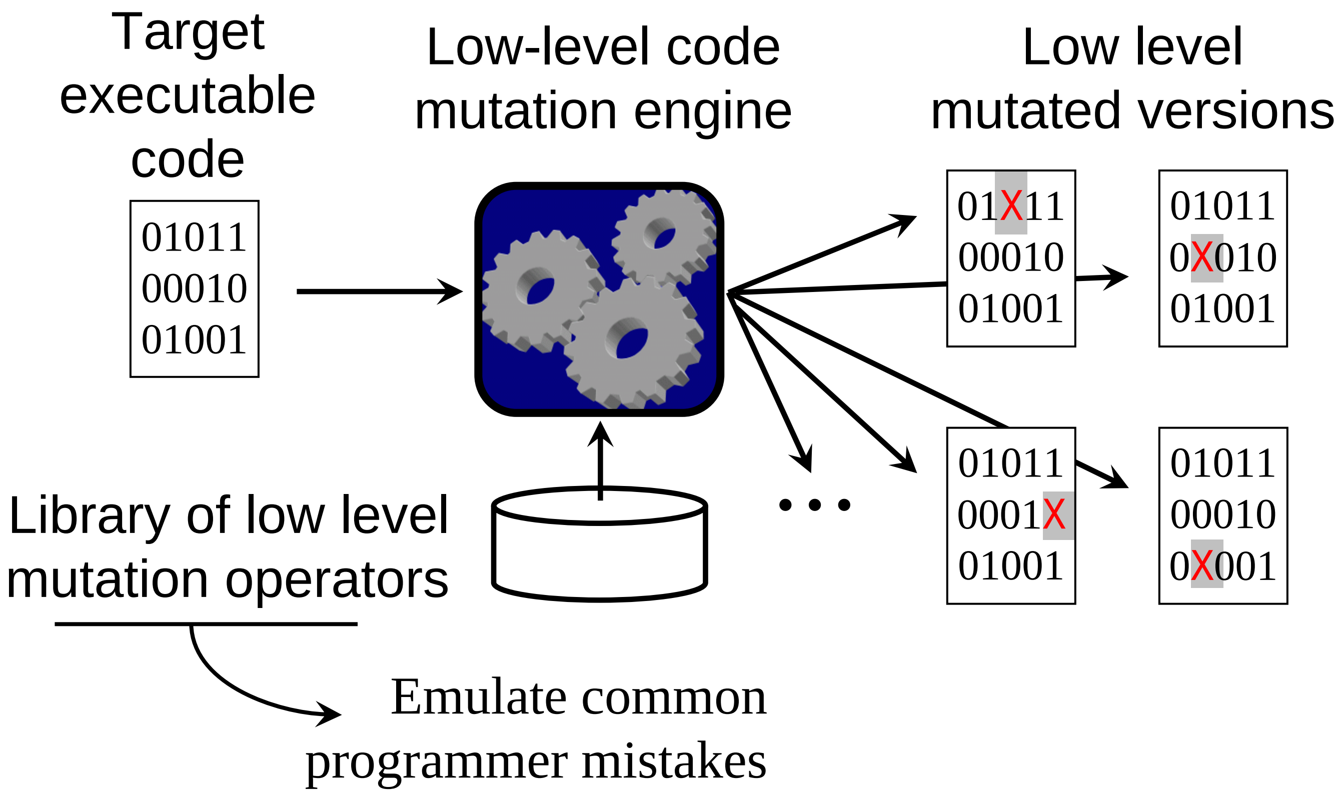
The most frequent software faults

Fault nature	# Faults	ASG	CHK	INT	ALG	FUN
Missing variable initialization	12	✓				
Missing variable assignment using a value	12	✓				
Missing variable assignment using an expression	16	✓				
Missing "if (*cond*)" surrounding statement(s)	23		✓			
Missing "AND EXPR" in expression used as branch condition	42		✓			
Missing function call	46				✓	
Missing "If (*cond*) { statement(s) }"	53				✓	
Missing "if (*cond*) statement(s) else" before statement(s)	17				✓	
Missing small and localized part of the algorithm	17				✓	
Missing functionality	21					✓
Wrong value assigned to variable	13	✓				
Wrong logical expression used as branch condition	16		✓			
Wrong arithmetic expression in param. of func. Call	12			✓		
Wrong variable used in parameter of function call	8			✓		
Wrong algorithm - large modifications	15					✓
Wrong data types or conversion used	12	✓				
Extraneous variable assignment using another variable	8	✓				
Total faults for these types in each ODC type	343	73	81	20	133	36
Fault coverage relative to each ODC type (%)	64.5	61.9	59.1	46.5	67.2	100

"Top-N" of software faults

Fault types	Description	Observed in field study	ODC classes
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

G-SWFIT
Generic software fault injection technique



The technique can be applied to binary files prior to execution or to in-memory running processes

Fault/operator example 1
Missing and-expression in condition

Target source code (avail. not necessary)

```
if ( a==3 && b==4 )
{
  do something
}
```

Code with intended fault

```
if ( a==3 && b==4 )
{
  do something
}
```

Original target code (executable form)

```
cmp dword ptr off_a[ebp],3
jne short ahead
cmp dword ptr off_b[ebp],4
jne short ahead
; ... do something ...
ahead:
...

; remaining prog. code
```

Target code with emulated fault

```
cmp dword ptr off_a[ebp],3
jne short ahead
nop
nop
nop
; ... do something ...
ahead:
...
; remaining prog. code
```

The actual mutation is performed in executable (binary) code. Assembly mnemonics are presented here for readability sake

Fault/operator example 2: Assignment instead equality comparison

Target source code (avail. not necessary)

```
if (v1 == v2)
{
    ...
}
```

→

Code with intended fault

```
if (v1 = v2)
{
    ...
}
```

Original target code (executable form)

```
MOV reg, mem1
CMP reg, mem2
JNE ahead
; ...
ahead:
; ...
```

→

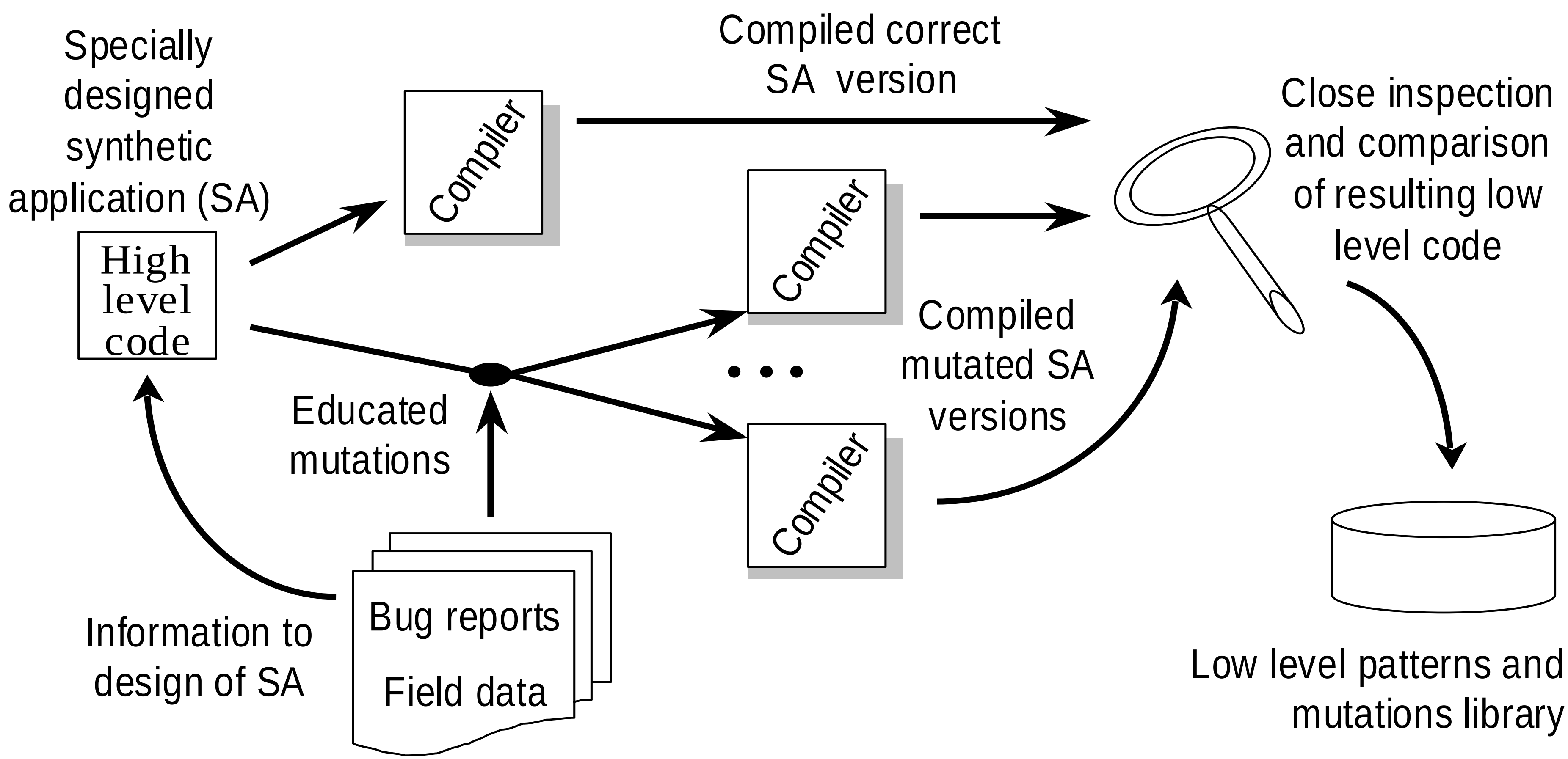
Target code with emulated fault

```
MOV reg, mem2
MOV mem1, reg
CMP reg, 0
JE ahead
; ...
ahead:
```

Some restrictions are enforced (e.g. it must not be preceded by a function call pattern to avoid **func() = =** ***val*** becoming **func() =** ***val***)

This fault is not the most common one, but it illustrates a mutation more complex than the previous one

Definition of the low-level mutation operators library



Validation of the technique

- **Accuracy?**

  Are the low-level faults actually equivalent to the high-level bugs?

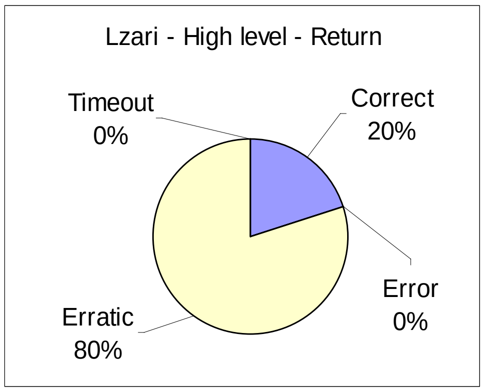
- **Generalization and portability**

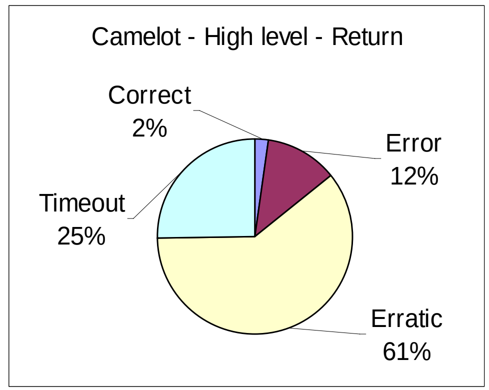
  Is the technique dependent on the compiler, optimization settings, high-level language, processor architecture, etc?

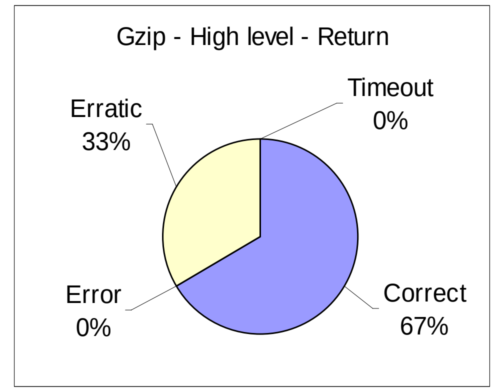
Example of results on accuracy validation: Missing or bad return statement

LZari | Camelot | GZip

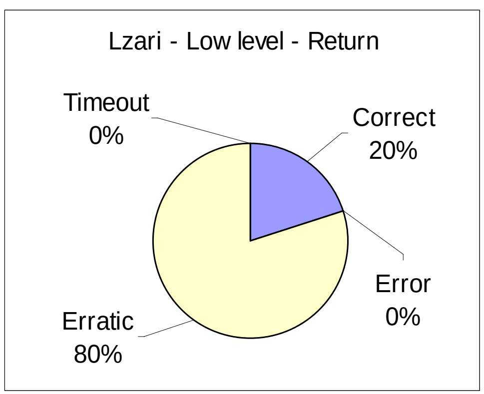
High-level

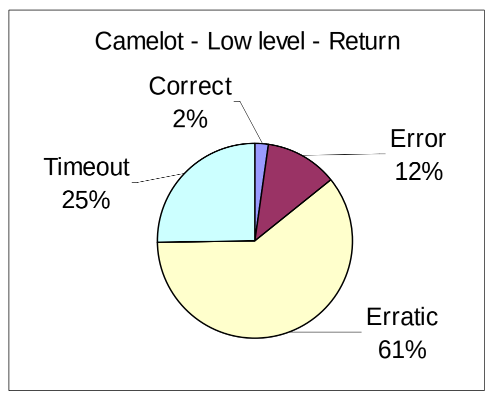


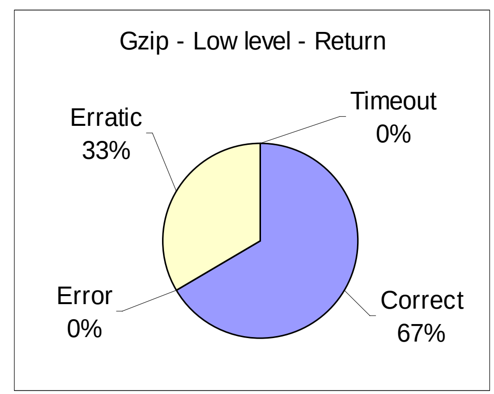




Low-level



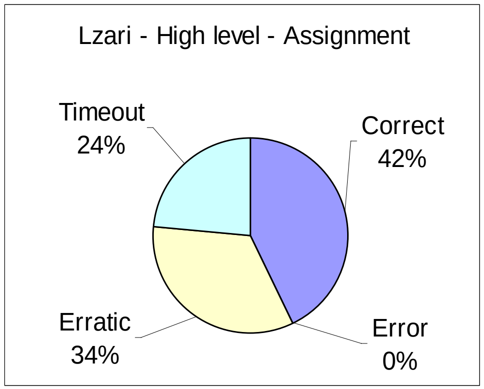


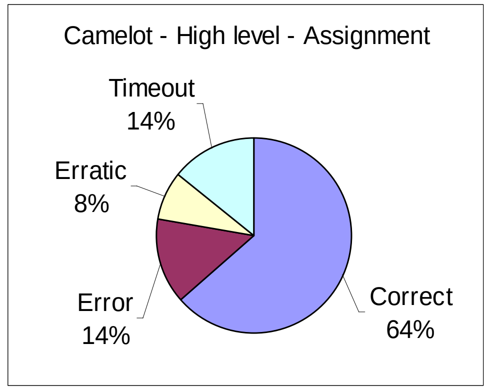


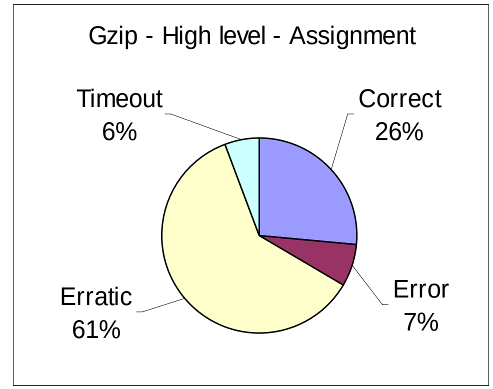
Example of results on accuracy validation: Assignment instead equality comparison

LZari **Camelot** **GZip**

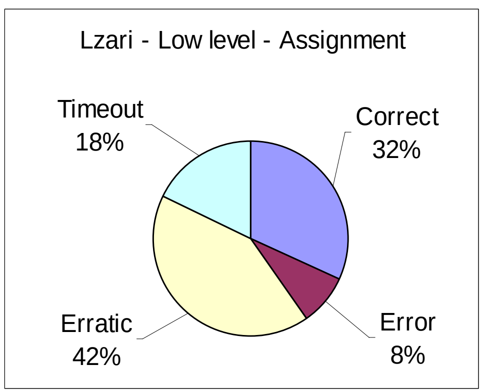
High-level

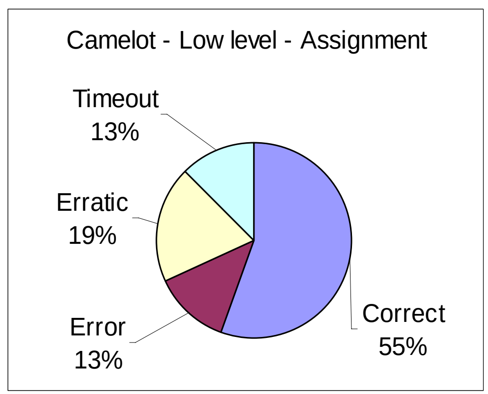


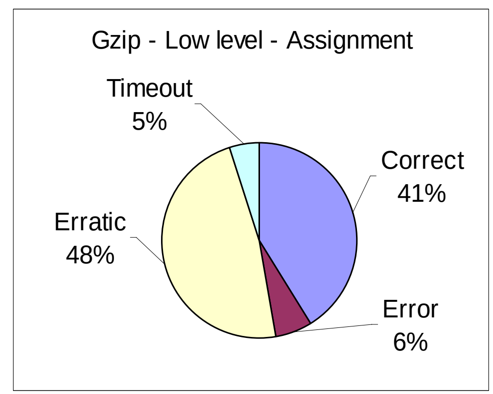




Low-level







Generalization of the technique

- Use of different compiler optimization settings
- Use of different compilers (Borland C++, Turbo C++, Visual C++)
- Use of different high-level languages (C, C++, Pascal)
- Different host architectures (Intel 80x86, Alpha AXP).

The library of fault operators (code patterns + mutations) depends essentially on the target architecture.

Current use of G-SWFIT

- Dependability benchmarking
  - DBench-OLTP: database and OLTP systems

    Already used to benchmark Oracle 8i, Oracle9i, and PostgreSQL running on top of Windows 2K, Windows XP, and Linux.
  - WEB-DB: web servers

    Already used to benchmark Apache and Abyss web servers running on top of Windows 2K, Windows XP, and Windows 2003.
- Independent verification and validation in NASA IV&V case-studies (project started on Feb. 2005).