Experimental Risk Assessment Using Software Fault Injection

Henrique Madeira
University of Coimbra
Portugal

University of Coimbra, Portugal

Component-based software development

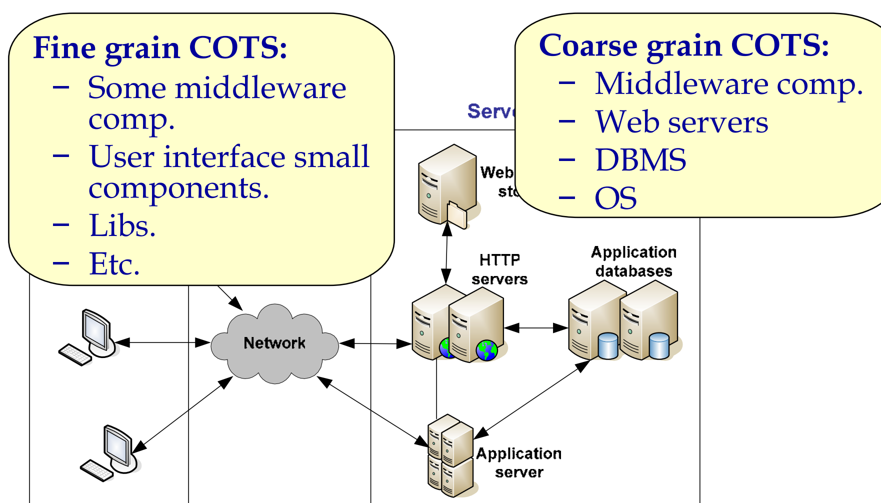
- **Vision:** development of systems using pre-fabricated components. Reuse custom components or buy software components available from software manufactures (Commercial-Off-The-Shelf: COTS).
- **Potential advantages:**
  - Reduce development effort since the components are already developed, tested, and matured by execution in different contexts
  - Improve system quality
  - Achieve of shorter time-to-market
  - Improve management of increased complexity of software
- **Trend** → **use general-purpose COTS components and develop domain specific components**.

Some potential problems

- COTS
  - In general, functionality description is not fully provided.
  - No guarantee of adequate testing.
  - COTS must be assessed in relation to their intended use.
  - The source code is normally not available (makes it impossible white box verification & validation of COTS).
- Reuse of custom components in a different context may expose components faults.

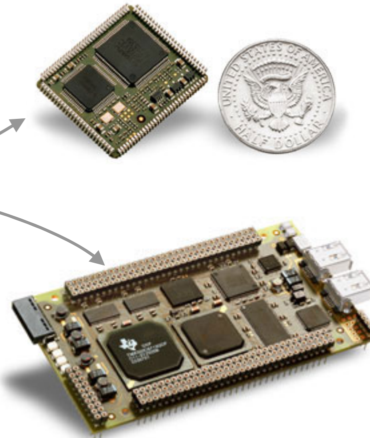
Using COTS (or reusing custom components) represent a risk!
How to assess (and reduce) that risk?

COTS in very large scale systems

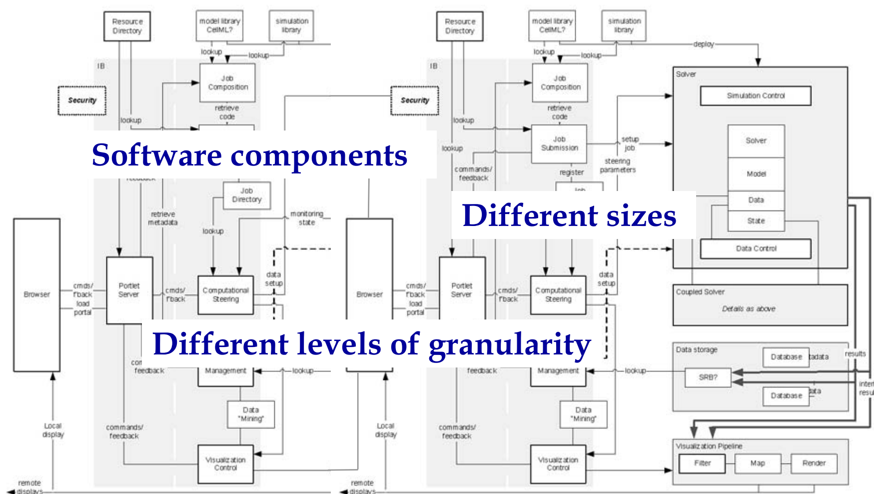


COTS in small scale systems

Control application
+
Real-time operating system



Software architecture diagram (dummy example)

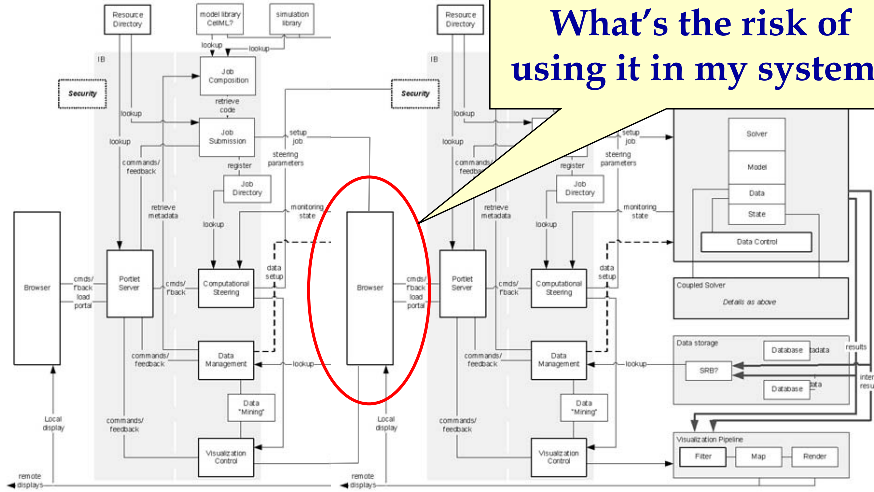


Software components

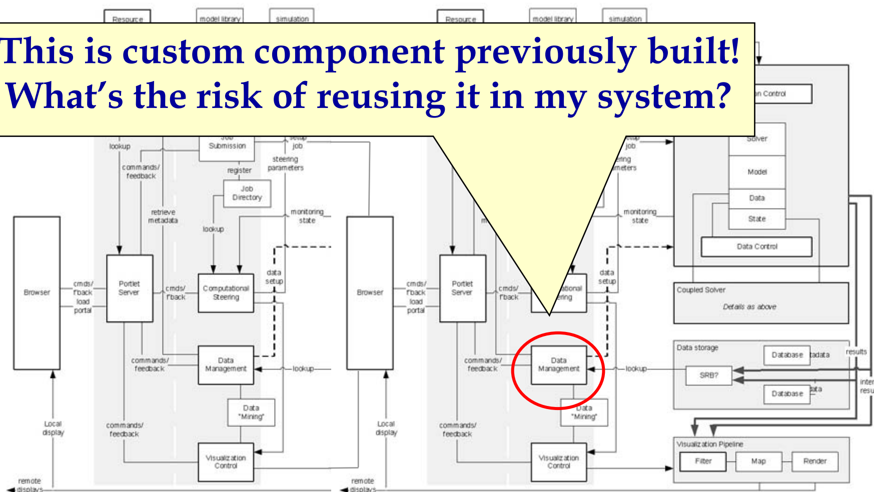
Different sizes

Different levels of granularity

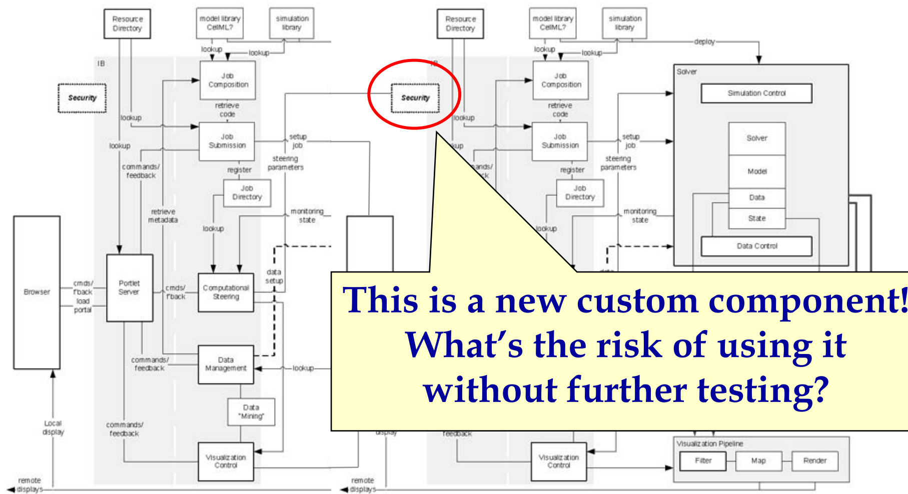
Question 1



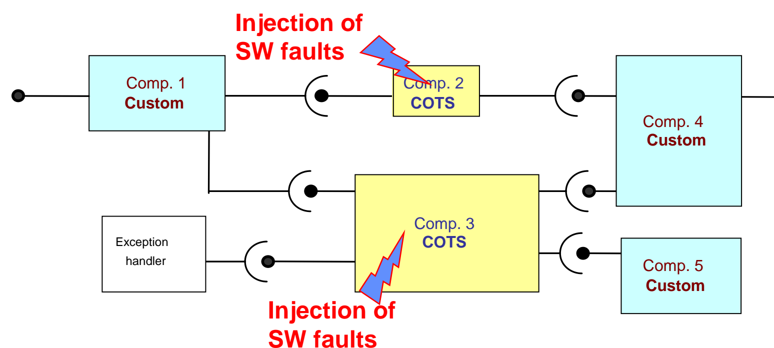
Question 2



Question 3

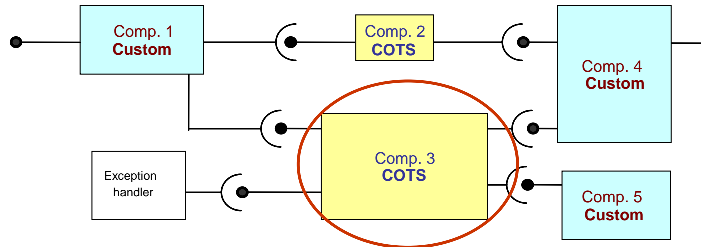


Software fault injection: Measuring <u>impact</u> of faults in SW components



1. Inject SW faults in the component
2. Measure system response (failure modes) to failures in the target component.
3. Evaluate how critical component failures are

Measuring risk of faults in SW components



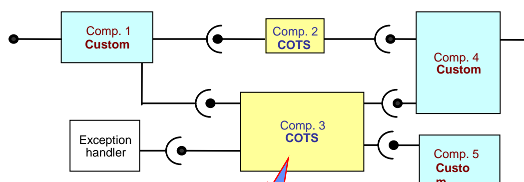
What's the risk of using Component 3 in my system?

Risk = prob. of bug * prob. of bug activation * impact of bug activation

Software complexity metrics | **Injection of software faults**

Software fault injection

- **Goal:** improve system dependability through:
  1. Measuring the impact of software faults in software components.
  2. Evaluating the risk of faults in software components
- Used when there is a systems prototype (real or even simulated).



Require representative:

- **Operational profile**
- **Injected faults**

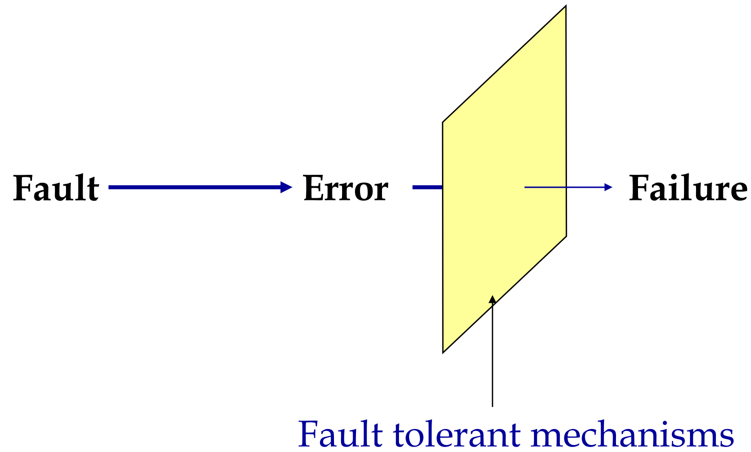
Research problems

1. How to emulate software faults realistically?
2. Is it really possible to estimate meaningful risk using fault injection + software complexity metrics?

What is fault injection?

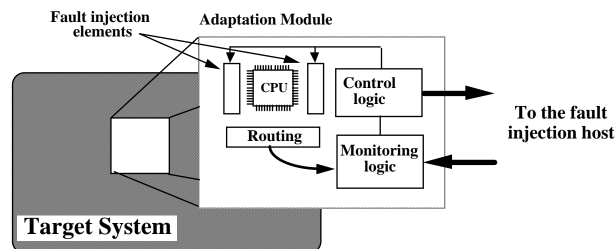
Deliberate insertion of upsets (faults or errors) in computer systems to evaluate its behavior in the presence of faults or validate specific fault tolerance mechanisms in computers.

Faults, Errors, and Failures



Examples of fault injection approaches

- Pin-level fault injection **(e.g., RIFLE)**



- Software Implemented Fault Injection **(e.g., Xception)**

  Reproduce pin-level fault inject

  Injection of hardware faul

What about injecting software faults?

What is a software fault?

Software development process (in theory...)

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

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

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

- ***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

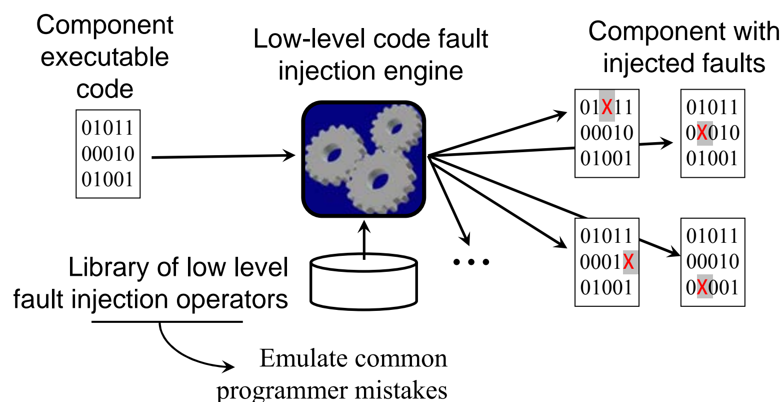
2 – ***Extraneous constructs*** are relatively infrequent

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

The "Top-N" software faults

Fault types	Perc. Observed in field study	ODC classes
Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
Missing function call	8.64 %	Algorithm
Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
Missing small and localized part of the algorithm	3.19 %	Algorithm
Missing variable assignment using an expression	3.00 %	Assignment
Wrong logical expression used as branch condition	3.00 %	Checking
Wrong value assigned to a value	2.44 %	Assignment
Missing variable initialization	2.25 %	Assignment
Missing variable assignment using a value	2.25 %	Assignment
Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
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
}
```

(the "&& b==4" part is crossed out)

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:
; ...
```

→

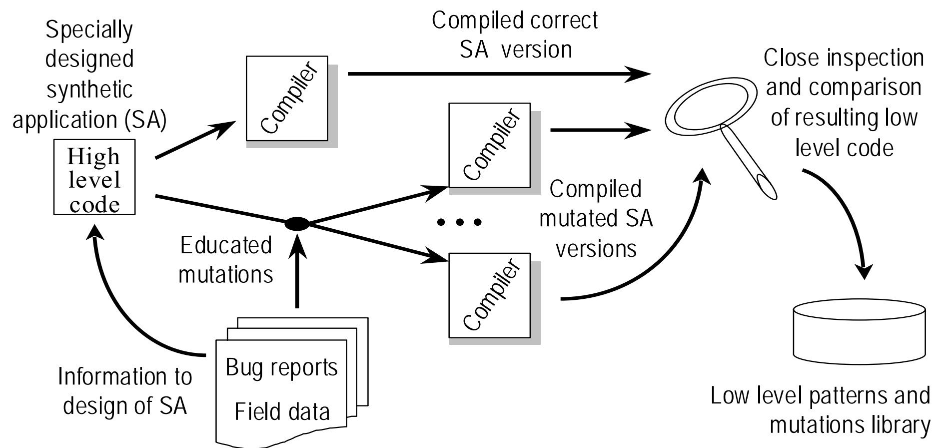
Target code with emulated fault

```
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

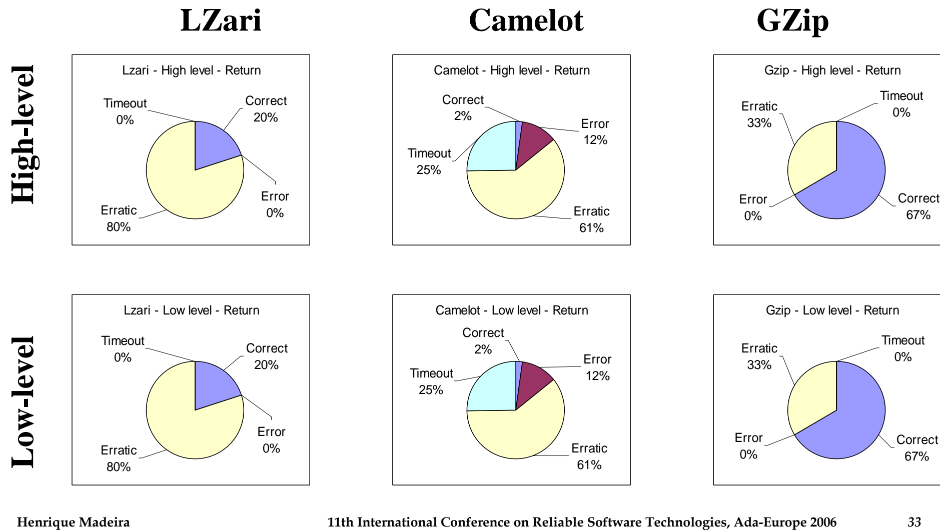
- **Accuracy?**

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

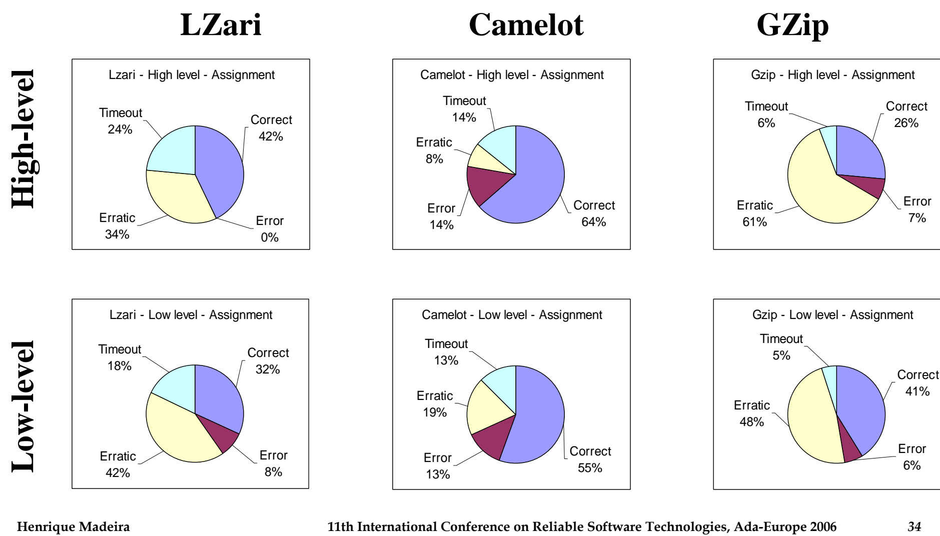
- **Generalization and portability**

  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



Example of results on accuracy validation: Assignment instead equality comparison



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 + code changes) depends essentially on the target processor architecture and programming model.

Current use of G-SWFIT

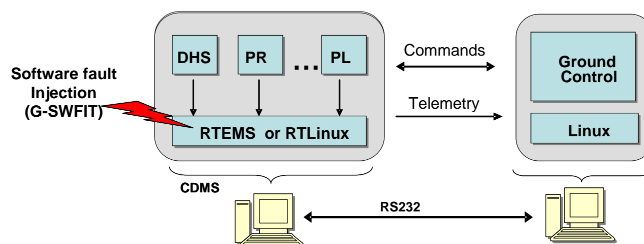
- Dependability benchmarking
  - DBench-OLTP: database and OLTP systems
    Already used to benchmark Oracle 8i, Oracle9i, and PostgreSQL running on top of Windows 2K, Windows XP, and Linux.
  - WEB-DB: web servers
    Already used to benchmark Apache and Abyss web servers running on top of Windows 2K, Windows XP, and Windows 2003.
- Independent verification and validation in NASA IV&V case-studies (project started on Feb. 2005).

Research problems

1. How to emulate software faults realistically?

2. Is it really possible to estimate meaningful risk using fault injection + software complexity metrics?

Case study: Satellite data handling system



- Evaluate feasibility of risk assessment approach
- Compare measured risk of using RTEMS versus RTLinux

Complexity analysis and estimation of the probability of residual fault

Application	# Module	LoC			C. Complexity (Vg)			Global
		< 100	100 - 400	> 400	< 25	25-40	> 40	*prob (f)*
RTEMS	1257	87,0%	11,0%	2,0%	80,0%	6,0%	14,0%	**7,5%**
RTLinux	2212	90,0%	9,0%	1,0%	84,0%	6,0%	10,0%	**6,5%**

More metrics can be used:

- number of parameters.
- number of returns.
- maximum nesting depth
- program length and vocabulary size (Halstead)

Probability of residual fault

$$prob(f) = \frac{\exp(\alpha + \beta_1 X_1 + \beta_2 X_2 + \dots \beta_n X_n)}{1 + \exp(\alpha + \beta_1 X_1 + \beta_2 X_2 + \dots \beta_n X_n)}$$

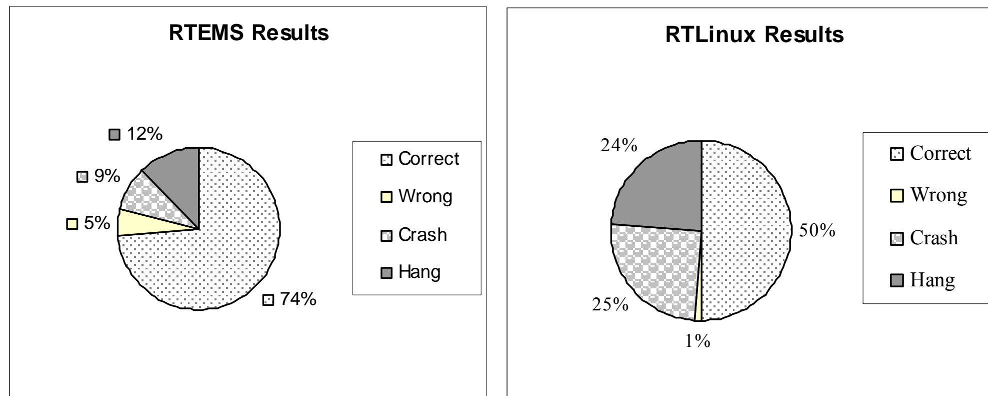
- Xi represents the product metrics
- α and β are estimated logistic regression coefficients
- exp is the base of the natural logarithms (2.718 ...)

From:

- M.-H. Tang, M.-H. Kao, M.-H. Chen, "An Empirical Study on Object-Oriented Metrics", In: Proceedings of the Sixth International Software Metrics Symposium, pp. 242-249, 1999.

- D. Hosmer, S. Lemeshow, "Applied Logistic Regression". John Wiley & Sons, 1989

Impact of software faults in the each operating system



Risk evaluation example

Comp. under analysis	prob(*f*)	Crash		Wrong		Hang		Incorrect Behavior	
		imp(*f*)	risk	imp(*f*)	risk	imp(*f*)	risk	imp(*f*)	risk
RTEMS	0.0749	0.09	0.67%	0.05	0.37%	0.12	0.89%	0.26	1.94%
RTLinux	0.0650	0.25	1.62%	0.01	0.06%	0.24	1.56%	0.50	3.25%

$$\textbf{Risk} = \underbrace{\text{prob. of bug}}_{prob(f)} * \underbrace{\text{prob. of bug activation} * \text{impact of bug activation}}_{impact(f)}$$

Conclusion

- Measure impact of components faults is ready to be used
  - Assess the impact of component faults (experimental criticality analysis).
  - Evaluate dependability improvements after system changes or patches.
- Experimental risk assessment is still under research. Validation of the approach is necessary.