INCREASED SAFETY BY FAULT TOLERANT SOFTWARE

JUST ANOTHER WAY TO WASTE MONEY?

MONIKA HEINER

mh@informatik.tu-cottbus.de
http://www.informatik.tu-cottbus.de

HIGHLY COMPETITIVE COMPETITION

- my new car!

- my new software toolkit?

ASR
ABSE
BV
ESP
USC

MTBF
MTTF
MTTR

BOOP
ASPECT
CORE
TL
ADT
OOP
VDM++
SADT
HOL
JSD
LOTOS
MASCOT
VDM
CSP
CCS
CSP
HOL
OOP
Z

MTBF
MTTF
MTTR
INTERNATIONAL STANDARD

- IEC 61508
  Functional safety of electrical/electronic/programmable electronic safety-related systems

- part 7
  Overview of techniques and measures, first edition August 2002

- Annex C
  Overview of techniques and measures for achieving software safety integrity

- C.2 Requirements and detailed design
  - C.2.5 Defensive programming

- C.3 Architecture design
  - C.3.1 Fault detection and diagnosis
  - C.3.2 Error detecting and correcting codes
  - C.3.3 Failure assertion programming
  - C.3.4 Safety bag
  - C.3.5 Software diversity
  - C.3.6 Recovery block
  - C.3.7 Backward recovery
  - C.3.8 Forward recovery
  - C.3.9 Re-try fault recovery mechanisms
  - C.3.10 Memorising executed cases
  - C.3.11 Graceful degradation
  - C.3.12 Artificial intelligence fault correction
  - C.3.13 Dynamic reconfiguration

METHODS TAXONOMY

SOFTWARE DEPENDABILITY

- FAULT AVOIDANCE - - - development phase
  - FAULT PREVENTION
  - FAULT REMOVAL
    - MANUAL
    - COMPUTER-AIDED
      - animation / simulation / testing
      - context checking (static analysis)
      - consistency checking (verification)

- FAULT TOLERANCE - - - operation phase
  - FAULT MASKING
    - DEFENSIVE
    - DIVERSITY
  - FAULT RECOVERY
**SYSTEM FAILURE / RELIABILITY - BASICS**

**RELIABILITY BLOCK DIAGRAM**

- [V1] -> [V2]  
- [V1]  
- [V2]  

**FAULT TREE**

- SF
  - >=1
  - f(v1) f(v2)  
- SF
  - &
  - f(v1) f(v2)

**LOGICAL FUNCTION**

- SF = f(v1) or f(v2)  
- SF = f(v1) and f(v2)

**PROBABILITY FUNCTION**

- p(SF) = 1 - [1 - p(v1)] * [1 - p(v2)]  
- p(v1) * p(v2)

- e.g. p(v1) = 0.1  
- p(v2) = 0.1  
- p(SF) = 0.19  
- p(SF) = 0.01

**CAUTION:** v1 must be statistically independent

---

**DEFENSIVE PROGRAMMING**

- **ERROR DETECTING/CORRECTING CODES**
  - -> sensitive information  
    - parity bits, Hamming codes, . . .

- **TRISTATE BRANCHING**
  - -> integrity check on (finite discrete) data types

- if b then thenAction  
  - else elseAction

- if b = true then thenAction  
  - else if b = false then elseAction

- endif

- if b = true then thenAction  
  - else if b = false then elseAction

- endif

- **SIGNATURES**
  - -> sequence errors in data/instruction streams
  - -> compile time:  
    - compute signature Sc for each instruction block,  
    - insert signature Sc at reference point
  - -> run time:  
    - re-compute signature Sr,  
    - at each reference point: Sc = Sr ?

- **TIMING CHECKS**
  - -> time outs, watchdog timers
FAILURE ASSERTION PROGRAMMING

- plausibility checks on (input/internal/output) variables
  -> robust programming

```c
if x >= 0
  then y := my_sqrt(x)
  else exceptionHandling
endif
```

```c
assert x >= 0;
y := my_sqrt(x);
```

- logical checks on system states
  -> detection of design faults

```c
assert x > 0; // pre-condition
  y := my_sqrt(x);
assert y*y = x; // post-condition
```

- run-time evaluation of assertions
  from thorough program verification approaches

MEMORISING EXECUTED CASES

- to prevent the execution of un-known paths

```plaintext
if x >= 0
  then y := my_sqrt(x)
  else exceptionHandling
endif
```

```c
assert x >= 0;
y := my_sqrt(x);
```

exception
when assertFail => exceptionHandling
end exception

- execution
- comparison
- ok
- ko
- fail

program source
instrumentation
compilation
executable
testing
operation
tested executable
tested paths
development phase
operation phase

memoisation

tested executable

case

execution

pre-case

current data

current path

result

compilation

instrumentation

program source

- test data

tested executable

tested paths
**N Version Programming**

-> parallel execution of n program versions, followed by majority test

**Recovery Block Scheme**

-> alternative execution of n program versions, each followed by acceptance test
**SUMMARY SOFTWARE DIVERSITY**

- **comparison**
  - n version programs
  - recovery block scheme
  - parallel versions
  - alternative versions

- **key assumptions**
  - error-free program frame
  - comparable results by identical interpretation of (semi-formal) requirement specification
  - independent faults in all versions / testers

- **versions are embedded in program frame**
  - flow control between versions and tester
  - reliability bottle-neck

- **equal functionality necessary**
  - gracefully degraded functionality possible

- **relative result test (voter)**
  - absolute result test (acceptance test)

- **hardware cheaper than run time**
  - run time cheaper than hardware

- **hot redundancy**
  - cold redundancy

**DIVERSITY ASPECTS**

- **versions are developed by different persons/teams of**
  - different education / universities
  - different professional background
  - different cultural circles

- **versions are developed using**
  - different software process models
  - different designs
  - different specification/programming languages
  - different programming styles
  - different tools (compiler, translator, debugger, ...)

- **versions are produced for different environments**
  - different hardware
  - different operating systems
  - different libraries

- **versions are based on**
  - different algorithms
  - different data structures
  - different (graceful degraded) functionality
FAULT RECOVERY

❑ BACKWARD RECOVERY
  reset to an earlier consistent internal state
  -> regular checkpointing necessary
  -> danger: domino effect!

❑ FORWARD RECOVERY
  obtain a state, which will be consistent some time later
  -> real-time systems
  with fast rate of internal state changes

❑ RE-TRY FAULT RECOVERY MECHANISMS
  re-executing the same code
  (re-boot the whole system ... re-start a procedure)
  -> exploiting non-reproducability

❑ GRACEFUL DEGRADATION
  maintaining the more critical system functions
  by dropping the less critical functions

❑ DYNAMIC CONFIGURATION
  remapping the logical architecture
  onto the currently available (restricted) resources

SUMMARY

❑ fault tolerance allows **basically**
  higher system reliability than components' reliability

❑ software fault tolerance = redundancy + DIVERSITY

❑ (diverse) fault tolerance is extremely expensive
  -> development & operation phase
  -> time & human/hardware resources
  -> what is more expensive:
  thorough validation or fault tolerance?

❑ fault tolerance involves increased complexity
  -> complexity <-> fault avoidance
  -> fault tolerance <-> reuse of trustworthy components
  -> advanced software engineering skills

❑ fault tolerance is no substitute for fault avoidance

❑ tailored amount of fault tolerance requires
  sound software reliability measures
  fault tolerance is no substitute for thinking

  -> **THINK TWICE BEFORE USING FAULT TOLERANCE !**

  **LOOK TWICE FOR SUITABLE MODULE SIZES !**
Disclaimer’s

- “The references should be considered as basic references to methods and tools or as examples, and may not represent the state of the art.” p. 23

- “The overview of techniques contained in this annex [C] should not be regarded as either complete or exhaustive.” p. 115

- “Currently, at the time of developing this standard, it is not clear whether object-oriented languages are to be preferred to other conventional ones.” p. 169

- “If a specific language is not listed in the table, it must not be assumed that it is excluded.” p. 173

Further Reading

  Wenn es auch der Titel nicht vermuten läßt, es werden sowohl fehlervermeidende als auch fehlertolerierende Verfahren für zuverlässige Software diskutiert.

  Guter Überblick über einschlägige (teilweise v. a. in den USA) übliche Standards, sowohl Spezifische für bestimmte Industriezweige als auch solche mit einer breiteren Allgemeingültigkeit.

  Konzentriert sich auf sicherheitsorientierte Systeme.

  Leider keine Monographie, sondern eine eher lose Sammlung von Themen-Blöcken, die irgendwie zur Software-Fehlertoleranz gehören.

  "Das zentrale Problem dieses ganzen Bandes ist die Umwandlung der Booleschen Funktion phi des Fehlerbaumes in irgendeine Form, aus der man einigermassen leicht die Wahrscheinlichkeit des Wertes phi=1, der dem Systemausfall entspricht, bestimmen kann." - der Autor. Eine eindrucksvolle, aber optimistische Falldarstellung zahlreicher Fußangeln der Wahrscheinlichkeitsrechnung. Nebenwirkung: glaube (fast) keinem Tool zur automatischen Fehlerbaum-Analyse.