DEPENDABLE SOFTWARE
FOR EMBEDDED SYSTEMS

MONIKA HEINER

BTU Cottbus
Computer Science Institute
Data Structures & Software Dependability
PROLOGUE

- my new car!

- my new software toolkit?
There is no such thing as a complete task description.

Sw systems tend to be (very) large and inherently complex systems.
- mastering the complexity?
- But, small system’s techniques can not be scaled up easily.

Large systems must be developed by large teams.
- communication / organization overhead
- But, many programmers tend to be lonely workers.

Sw systems are abstract, i.e. have no physical form.
- no constraints by manufacturing processes or by materials governed by physical laws
- SE differs from other engineering disciplines
- But, human skills in abstract reasoning are limited.

Sw does not grow old.
- no natural die out of over-aged sw
- sw cemetery
- But, “sw mammoths” keep us busy.
OVERVIEW

- dependability taxonomy
- methods to improve dependability

SOFTWARE DEPENDABILITY

development phase

FAULT AVOIDANCE
- fault prevention
- fault removal
  - manuellly
  - computer-aided validation
    - animation / simulation / testing
    - context checking (static analysis)
    - consistency checking (verification)

operation phase

FAULT TOLERANCE
- fault masking
  - defensive
  - diversity
- fault recovery

February 2004
STATE OF THE ART

- natural fault rate of seasoned programmers - about 1-3 % of produced program lines

- undecidability of basic questions in sw validation
  - program termination
  - equivalence of programs
  - program verification
  - ...

- validation = testing

- testing portion of total sw production effort
  - standard system: ≥ 50 %
  - extreme availability demands: ≈ 80 %

Murphy’s law: There is always still another fault.

Cleanroom approach
LIMITATIONS OF TESTING

“Testing means the execution of a program in order to find bugs.” [Myers 79]
-> A test run is called successful, if it discovers unknown bugs, else unsuccessful.

- testing is an inherently destructive task
  -> most programmers unable to test own programs

“Program testing can be used to show the presence of bugs, but never to show their absence!” [Dijkstra 72]

- exhaustive testing impossible
  - all valid inputs
    -> correctness, . . .
  - all invalid inputs
    -> robustness, security, reliability, . . .
  - state-preserving software (OS/IS): a (trans-) action depends on its predecessors
    -> all possible state sequences

- systematic testing of concurrent programs is much more complicated than of sequential ones
Testing of Concurrent Software

- **State space explosion**, worst-case: product of the sequential state spaces

- **Probe Effect**
  - System exhibits in test mode other (less) behavior than in standard mode
    -> Test means (debugger) affect timing behavior
  - Result: Masking of certain types of bugs:
    - DSt (pn) -> not DSt (tpn)
    - live(pn) -> not live (tpn)
    - not BND (pn) -> BND (tpn)

- **Non-deterministic behavior**, -> pn: time-dependent dynamic conflicts

- Dedicated testing techniques to guarantee **reproducibility**, e.g. Instant Replay
MODEL-BASED SYSTEM VALIDATION

- General principle
- Modelling = abstraction
- Analysis = exhaustive exploration
- (Amount of) analysis techniques depend on model type

Diagram:
- System properties
- Model properties
- Modelling
- Analysis
- Conclusions
MODEL-BASED SYSTEM VALIDATION

- process and tools
- DFG project, PLC’s
- dedicated technical language for requirement spec
- error message = inconsistency between system model & requirement spec
- verification methods -> toolkit

Diagram:

- controller
- environment
- modelling (compiler)
- modelling
- library
- control model
- environment model
- composition
- system model
- verification methods
- errors / inconsistencies
- temporal logic
- set of temporal formulae
- safety requirements
- functional requirements

Note: The diagram illustrates the process and tools for model-based system validation, including the use of a dedicated technical language for requirement specification and verification methods.
MODEL-BASED SYSTEM VALIDATION

- objective - reuse of certified components
MODEL-BASED SYSTEM VALIDATION

- model classes
- analysis methods
- analysis objectives

MODEL CLASSES

QUALITATIVE MODELS
- context checking
- verification by model checking

QUANTITATIVE MODELS
- NON-STOCHASTIC MODELS
- worst-case evaluation
- performance prediction
- reliability prediction

- STOCHASTIC MODELS
  - performance prediction
STATE SPACE EXPLOSION, POSSIBLE ANSWERS

BASE CASE TECHNIQUES

- compositional methods
  -> simple module interfaces

- abstraction by ignoring some state information
  -> conservative approximation

ALTERNATIVES ANALYSIS METHODS

- structural analysis
  -> structural properties, reduction

- Integer Linear Programming

- compressed state space representations
  -> symbolic model checking (OxDD)

- lazy state space construction
  -> stubborn sets, sleep sets

- alternative state spaces (partial order representations)
  -> finite prefix of branching process
  -> concurrent automaton

PROOF ENGINEERING
CASE STUDY - PRODUCTION CELL

- deposit belt (belt 2)
- travelling crane
- feed belt (belt 1)
- elevating rotary table
- robot
- arm 1
- arm 2
- press

14 sensors
34 commands
CASE STUDY - DINING PHILOSOPHERS

BDD ANALYSIS RESULT, PHIL1000:

Number of places/marked places/transitions: 7000/2000/5000

Number of states: ca. $1.1 \times 10^{667}$

Time to compute P-Invariants: 45885.66 sec
Number of P-Invariants: 3000
Time to compute compact coding: 385.59 sec
Number of Variables: 4000
Time: 3285.73 sec ca. 54.75’
SUMMARY - SOFTWARE VALIDATION

- validation can only be as good as the requirement specification
  - readable <-> unambiguous
  - complete <-> limited size

- validation is extremely time and resource consuming
  - 'external' quality pressure?

- sophisticated validation is not manageable without theory & tool support

- validation needs knowledgeable professionals
  - study / job specialization
  - profession of “software validator”

- validation is no substitute for thinking

- There is no such thing as a fault-free program!
  - sufficient dependability for a given user profile
Ich wage zu bezweifeln, daß mich das, was ich bei Ihnen lerne, "da draußen" weiterbringt.
FAULT TOLERANCE

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

- part 7
  Overview of techniques & 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
MEMORISING EXECUTED CASES

- to prevent the execution of un-known paths
- only tested paths are reliable paths
- requires excessive testing
FAULT TOLERANCE - SOFTWARE DIVERSITY

N VERSION PROGRAMMING

- parallel execution of n program versions
- followed by majority test
- higher abstraction level, transitions:
  -> program versions
  -> voting algorithm

Diagram:

- Start
- Fork
- v1_start, v2_start, v3_start
- v1, v2, v3
- v1_end, v2_end, v3_end
- Voting
- Voting result: all equal, two equal, all unequal
- Ok, Warning, Ko
- Success, Fail
RECOVERY BLOCK SCHEME

- alternative execution of n program versions
- each followed by acceptance test
- high-level Petri net
SUMMARY - FAULT TOLERANCE

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

- fault tolerance is no substitute for fault avoidance

- fault tolerance is no substitute for thinking

- tailored amount of fault tolerance requires sound software reliability measures

Think twice before using fault tolerance!

Look twice for suitable module sizes!
ANOTHER SUMMARY - BEYOND THE LIMIT

Sugar Bowl Trail Ends Here

Beyond This Point You Are Entering A Wild Mountainous Area. You Should Be Prepared With Appropriate Clothing, Have A Map And Compass, And Know How To Use Them.
EPILOGUE

- Model-based software validation
  - waste of money?

- Fault-tolerant software
  - just another way to waste money?

- Dependable software
  - an unrealistic dream or just a reality far away?