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.
But, small system’s techniques cannot be scaled up easily.

Large systems must be developed by large teams.
But, many programmers tend to be lonely workers.

Sw systems are abstract, i.e. have no physical form.
- no constraints by manufacturing processes or materials governed by physical laws
- software engineering 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
  - manually
  - computer-aided validation
    - animation / simulation / testing
    - context checking (static analysis)
    - consistency checking (verification)
    - model checking

Operation phase
- Fault Tolerance
  - fault masking
    - defensive
    - diversity
  - fault recovery
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: \( \geq 50\% \)
  -> extreme availability demands: \( \approx 80\% \)

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.

- "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 is an inherently destructive task
  -> most programmers unable to test own programs
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:
  - \(\text{DSt (pn)} \rightarrow \text{not DSt (tpn)}\)
  - \(\text{live(pn)} \rightarrow \text{not live (tpn)}\)
  - \(\text{not BND (pn)} \rightarrow \text{BND (tpn)}\)

- **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

Monika.Heiner(at)b-tu.de
July 2017
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

- objective - reuse of certified components

---

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

- objective - reuse of certified components

---

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

- objective - reuse of certified components

---
MODEL-BASED SYSTEM VALIDATION

- **MODEL CLASSES**
  - model classes
  - analysis methods
  - analysis objectives

QUALITATIVE MODELS
- context checking
- verification by model checking

QUANTITATIVE MODELS
- worst-case evaluation
- performance prediction
- reliability 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**
  - finite prefix of branching process
  - concurrent automaton
CASE STUDY - PRODUCTION CELL

Dependability Engineering

feed belt (belt 1)

deposit belt (belt 2)

travelling crane

robot

arm 2

press

elevating rotary table

arm 1

14 sensors

34 commands

CASE STUDY - DINING PHILOSOPHERS

Dependability Engineering

BDD ANALYSIS RESULT, PHIL1000:

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

Number of states: ca. 1.1 * 10e667

113751760865620516280733543276768405854187694780000110922858323169918\n15995958812203133264112006909717907134071439603793701320514129462357710\n24420952273842421885324723962294300719880861927627555972033293946931\n334482712874090358795533181711372868591957997236895570937383074225421\n493299735059349871208726085116502627817824646762991281238722816835426\n439043702222227167126998740049615901200930144970216630268925118631696\n7921927976430854076756777224220660460294623534355683154921949034887\n413898510872615227535084646719457353408471086965332494805497753832942\n1717811011687205102115416900392117962799564222929032376885417450385275\n5124881924010536352551190474777411874

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

ANOTHER SUMMARY - SOME DOUBTS

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

---

fault tolerance - defensive software

memorising executed cases

- to prevent the execution of un-known paths
- only tested paths are reliable paths
- requires excessive testing
**N VERSION PROGRAMMING**

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

**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

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?