DEPENDABLE SOFTWARE FOR EMBEDDED SYSTEMS

MONIKA HEINER

BTU Cottbus
Computer Science Institute
Data Structures & Software Dependability

- **my new car !**

MSR
ASR
ABS EBV
ESP USC

- **my new software toolkit ?**

BOOP ASPECT
CORE TL ADT OOP VDM++
SADT HOL JSD LOTOS
MASCOT VDM
RBD DFD CCS CSP
FTA OBJ Z
SA
NVP CTL/LTL RBS
MTBF MTTF MTTR

Dependable Software - Alligators

- **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*

  -> *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**

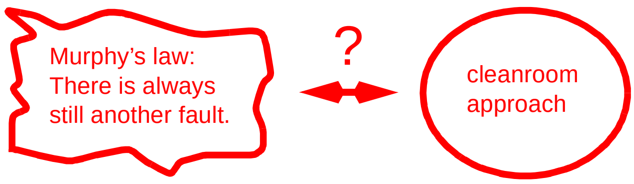
SOFTWARE DEPENDABILITY

- *development phase*: FAULT AVOIDANCE
  - fault prevention
  - fault removal
    - manuelly
    - 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: ≥ 50 %*

  -> *extreme availability demands: ≈ 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]**
- **testing is an inherently destructive task**

  -> *most programmers unable to test own programs*

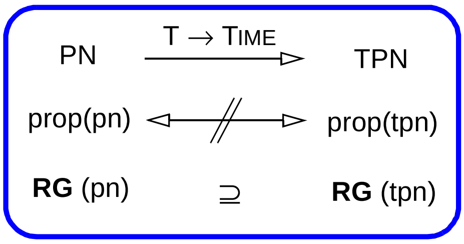
- **exhaustive testing impossible**
  - all valid inputs
    -> correctness, . . .
  - all invalid inputs
    -> robustness, security, reliability, . . .
  - state-preserving software (OS/IS):
    a (trans-) action depends on its predecessors
    -> all possible state sequences
- **systematic testing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

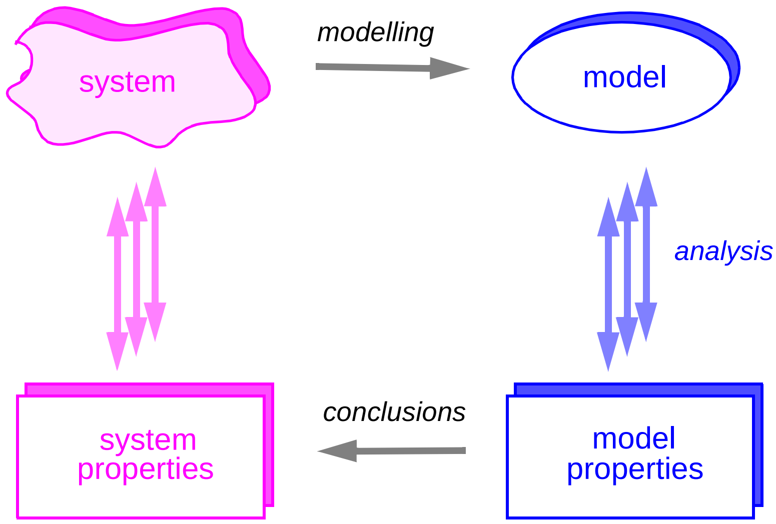


- **non-deterministic behavior,**
  -> pn: time-dependent dynamic conflicts

- **dedicated testing techniques to guarantee reproducibility,**
  e. g. Instant Replay

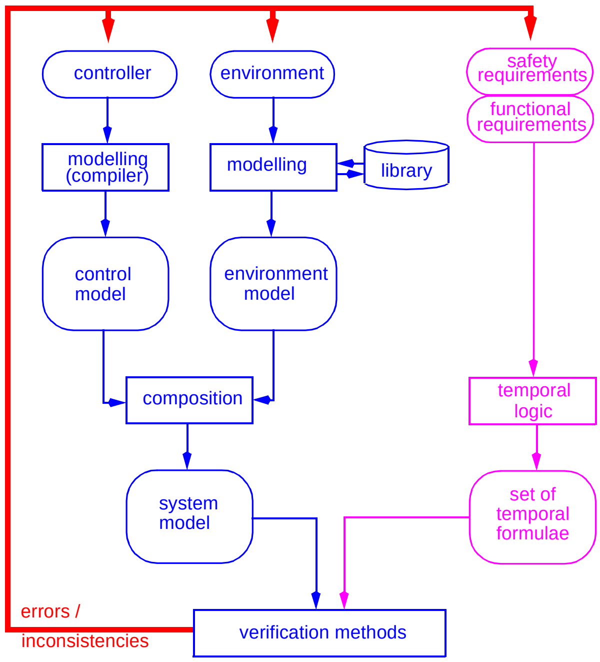
MODEL-BASED SYSTEM VALIDATION

- **general principle**
- **modelling = abstraction**
- **analysis = exhaustive exploration**
- **(amount of) analysis techniques depend on model type**



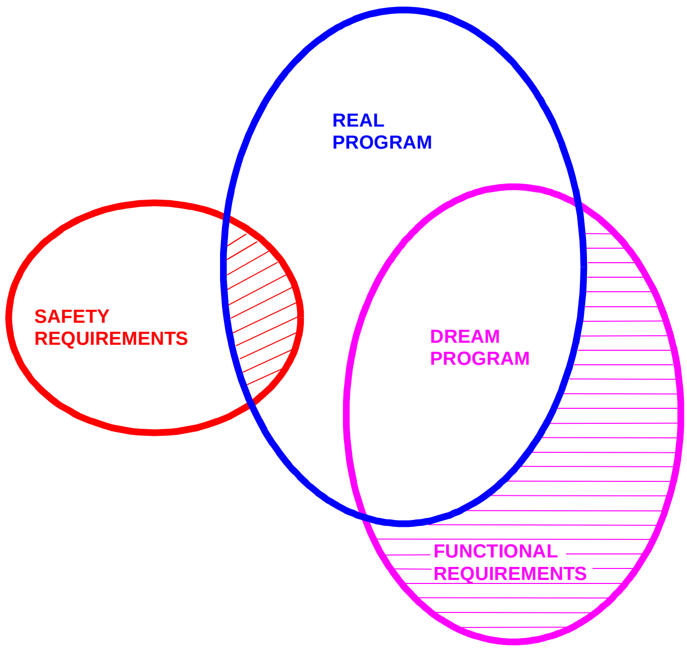
Model-based System Validation

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



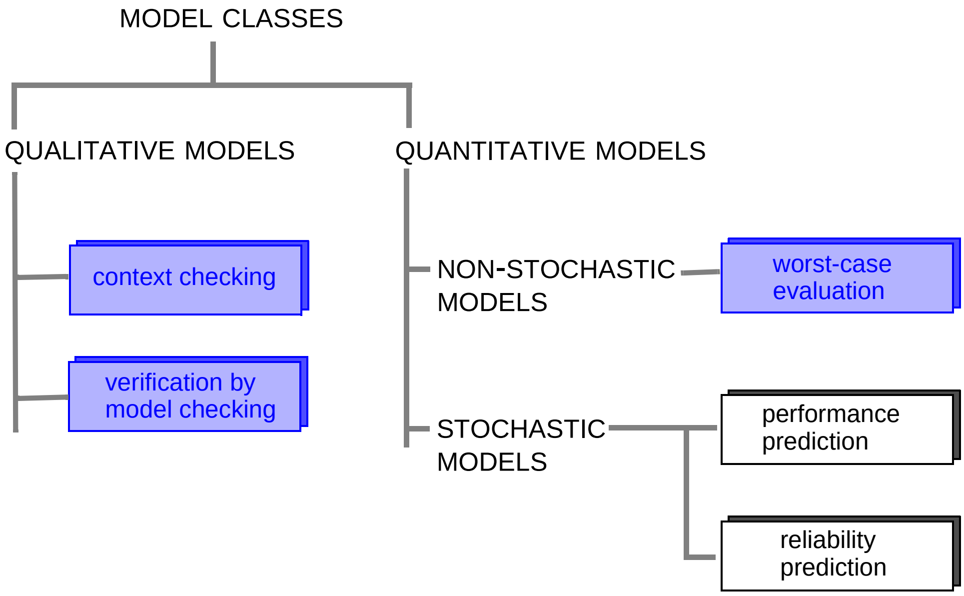
Model-based System Validation

- objective - reuse of certified components



MODEL-BASED SYSTEM VALIDATION

- ❑ **model classes**
- ❑ **analysis methods**
- ❑ **analysis objectives**



STATE SPACE EXPLOSION, POSSIBLE ANSWERS

BASE CASE TECHNIQUES

- ❑ **compositional methods**
  -> simple module interfaces

- ❑ **abstraction by ignoring some state information**
  -> conservative approximation

⇊

PROOF ENGINEERING

ALTERNATIVES ANALYSIS METHODS

- ❑ **structural analysis**
  -> structural properties, reduction

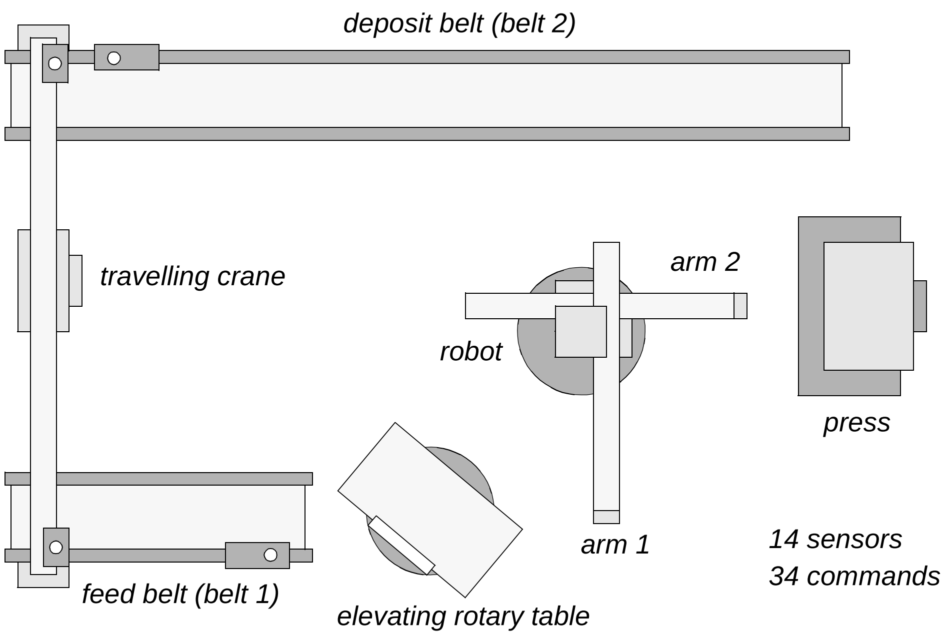
- ❑ **Integer Linear Programming**

- ❑ **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

Case Study - Production Cell



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 * 10e667

1137517608656205162806720354362767684058541876947800011092858232169918\\
1599595881220313326411206909717907134074139603793701320514129462357710\\
2442895227384242418853247239522943007188808619270527555972033293948691\\
3344982712874090358789533181711372863591957907236895570937383074225421\\
4932997350559348711208726085116502627818524644762991281238722816835426\\
4390437022222227167126998740049615901200930144970216630268925118631696\\
7921927977564308540767556777224220660450294623534355683154921949034887\\
4138935108726115227535084646719457353408471086965332494805497753382942\\
1717811011687720510211541690039211766279956422929032376885414750385275\\
51248819240105363652551190474777411874

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

FAULT TOLERANCE

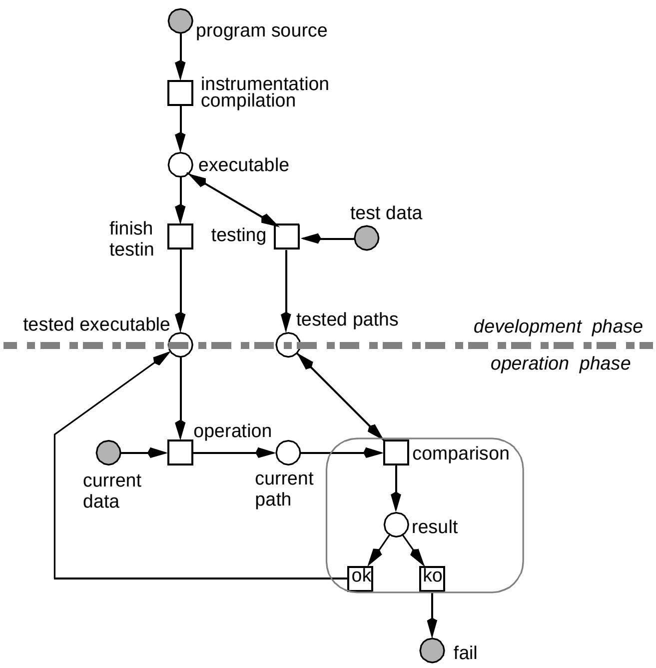
- **International Standard IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems**
- **part 7 Overview of techniques & measures, first edition August 2002**
- **Annex C Overview of techniques and measures for achieving software safety integrity**

- **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

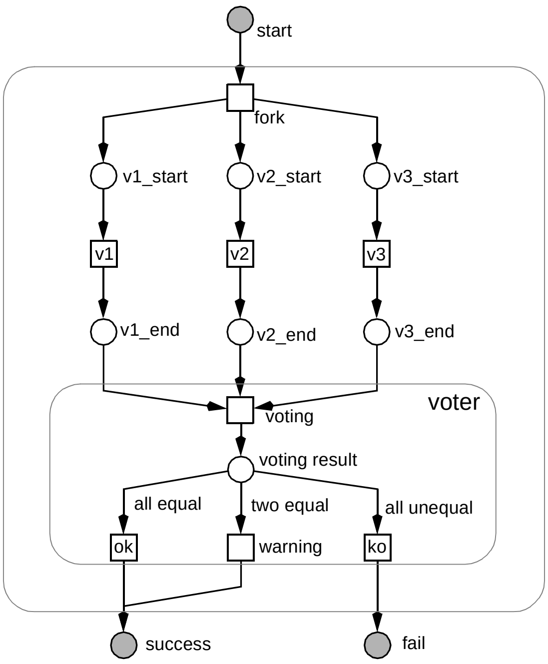
MEMORISING EXECUTED CASES

- **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

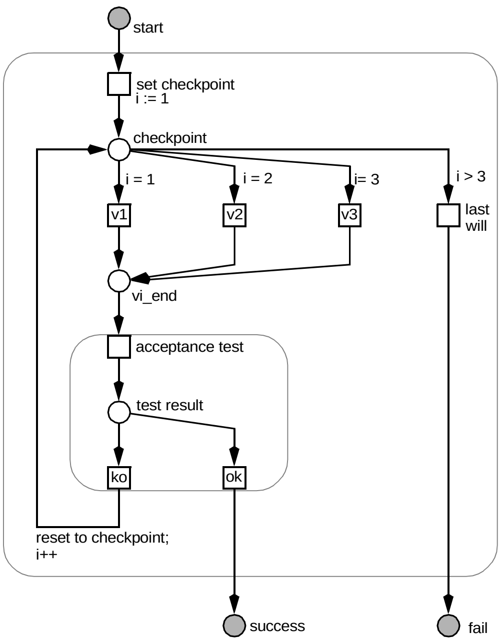


FAULT TOLERANCE - SOFTWARE DIVERSITY

dependability engineering

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

- **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 ?