DEPENDABLE SOFTWARE FOR EMBEDDED SYSTEMS

MONIKA HEINER

BTU Cottbus
Computer Science Institute
Data Structures & Software Dependability

PROLOGUE

- my new car !

MSR
ASR
ABSEBV
ESP USC

- my new software toolkit ?

BOOP ASPECT
CORE TL ADT OOP VDM++
SADT HOL JSD LOTOS
RBD MASCOT VDM
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*

  -> *SE differs from other engineering disciplines*

  ***But,** human skills in abstract reasoning are limited.*

- **Sw does not grow old.**

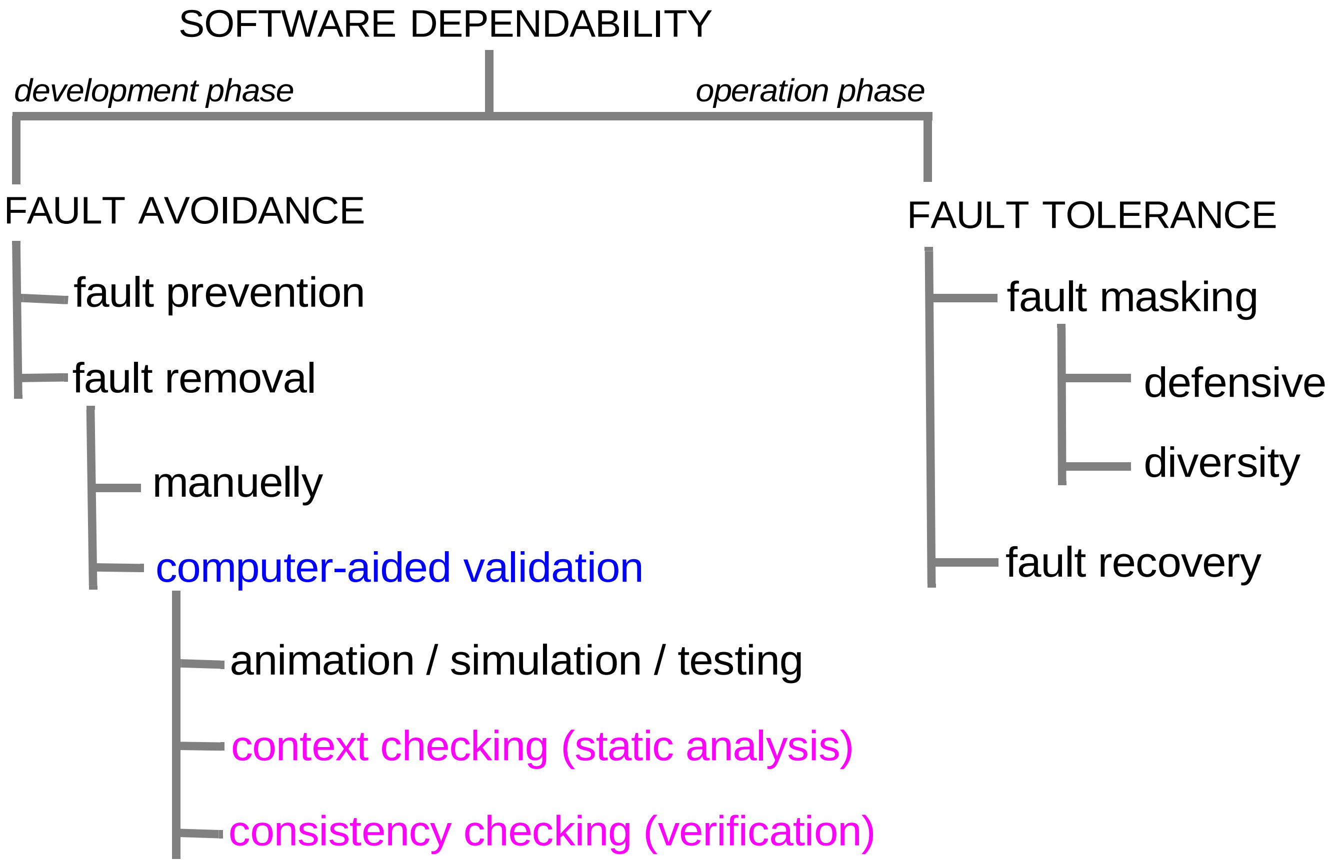
  -> *no natural die out of over-aged sw*

  -> *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
    - manuelly
    - computer-aided validation
      - animation / simulation / testing
      - context checking (static analysis)
      - consistency checking (verification)

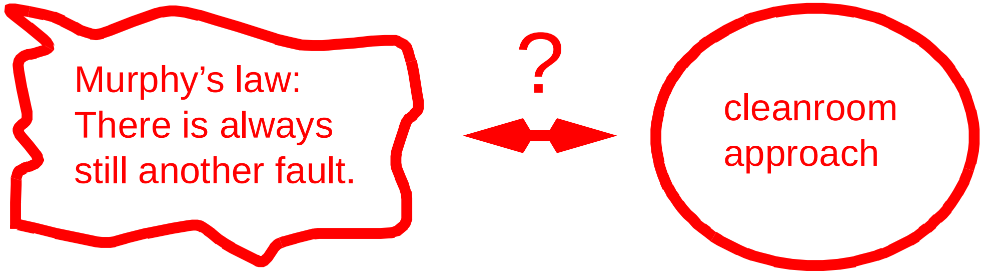
operation phase

- 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
  - . . .



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

?

cleanroom
approach

- validation = testing

- testing portion of total sw production effort

  -> *standard system: ≥ 50 %*

  -> *extreme availability demands: ≈ 80 %*

monika.heiner(at)informatik.tu-cottbus.de
data structures and software dependability

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)

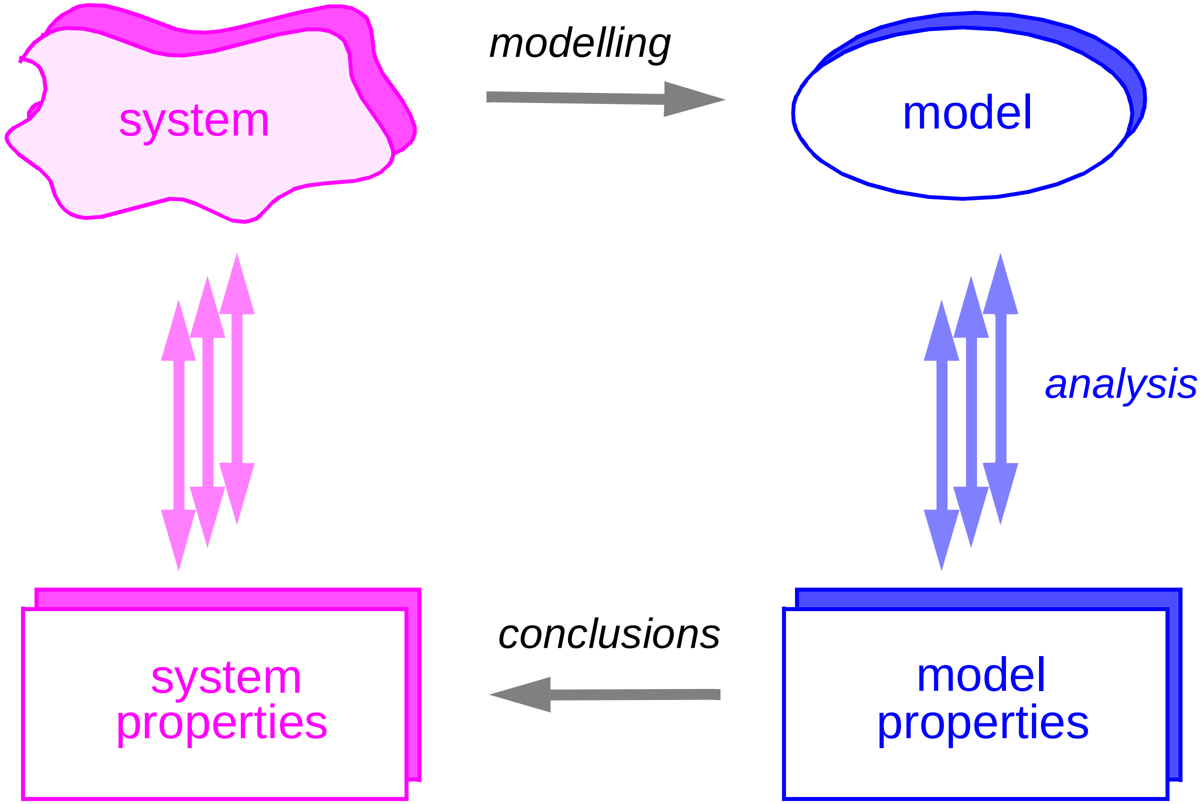
$$\begin{tikzcd} \mathrm{PN} \arrow[r, "T \to \mathrm{TIME}"] & \mathrm{TPN} \\ \mathrm{prop}(pn) \arrow[r, phantom, "\not\longleftrightarrow"] & \mathrm{prop}(tpn) \\ \mathbf{RG}\ (pn) \arrow[r, phantom, "\supseteq"] & \mathbf{RG}\ (tpn) \end{tikzcd}$$

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

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

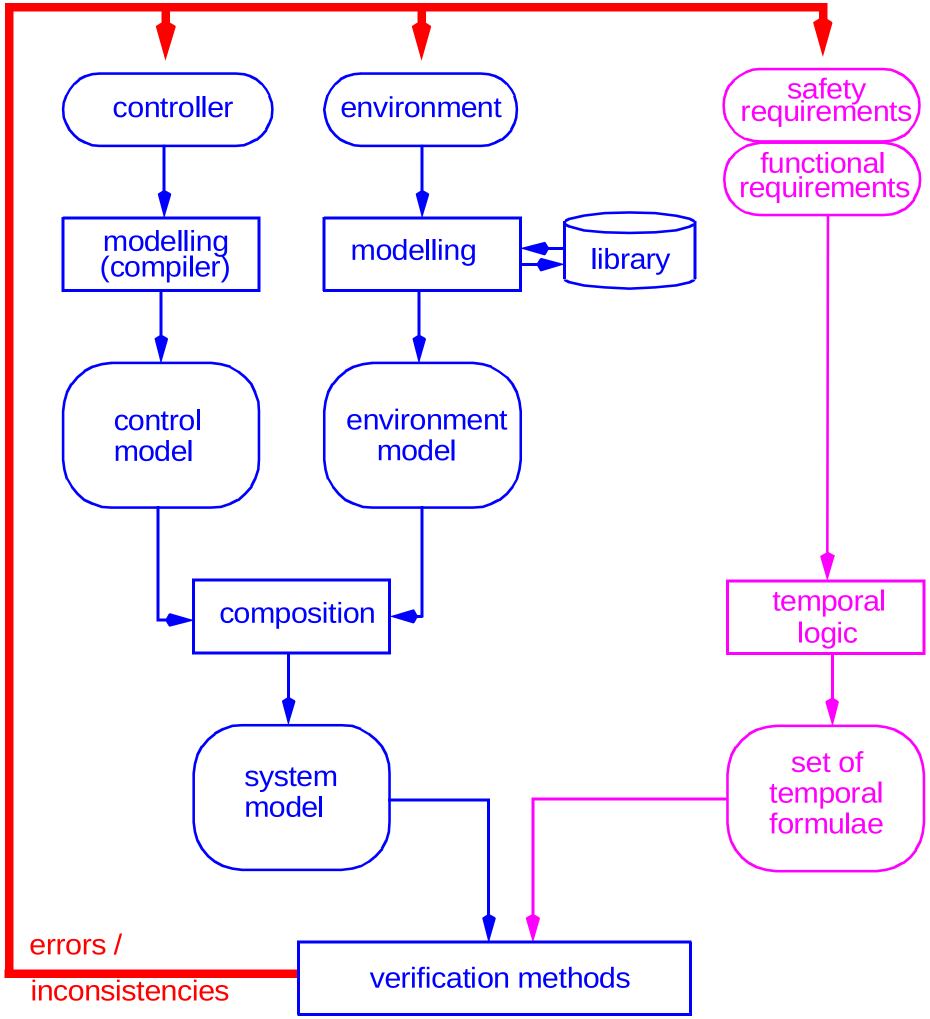
MODEL-BASED SYSTEM VALIDATION

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



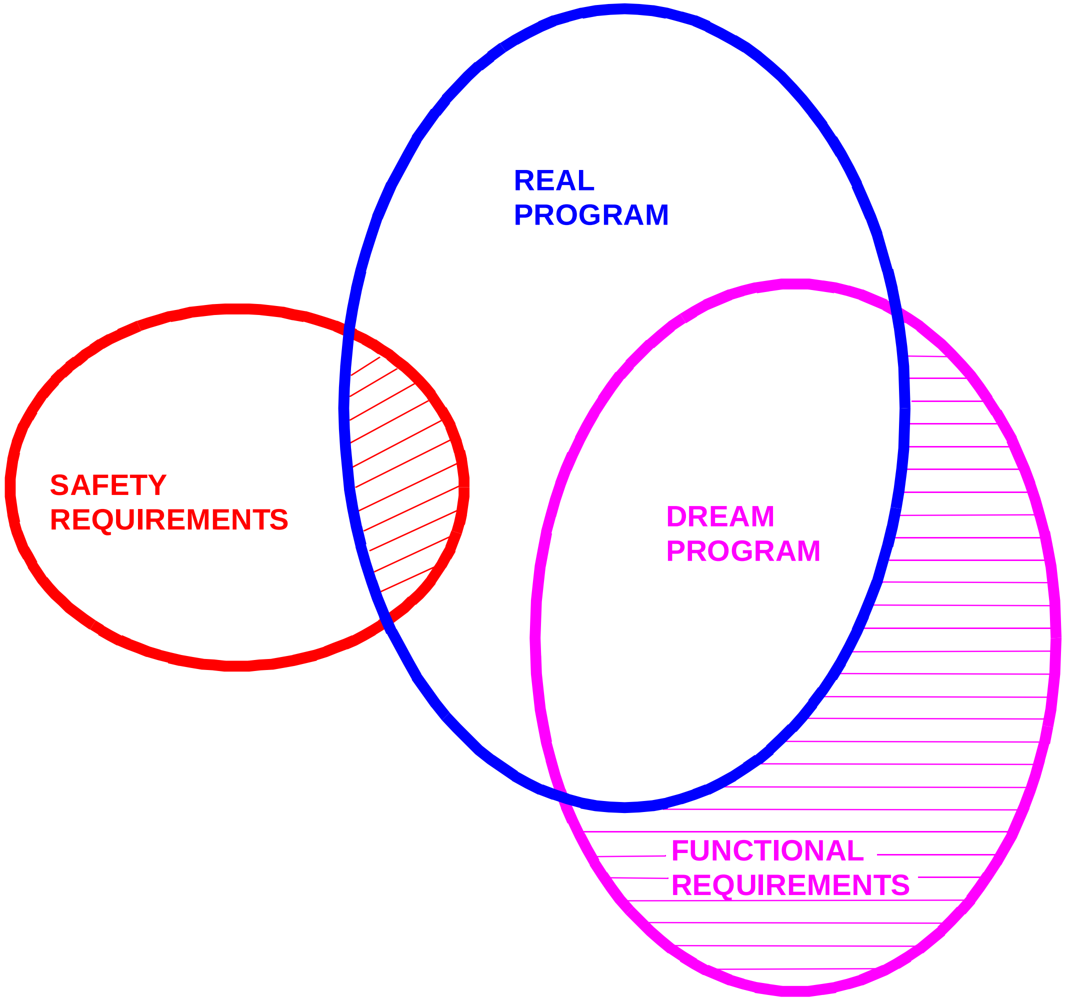
MODEL-BASED SYSTEM VALIDATION

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



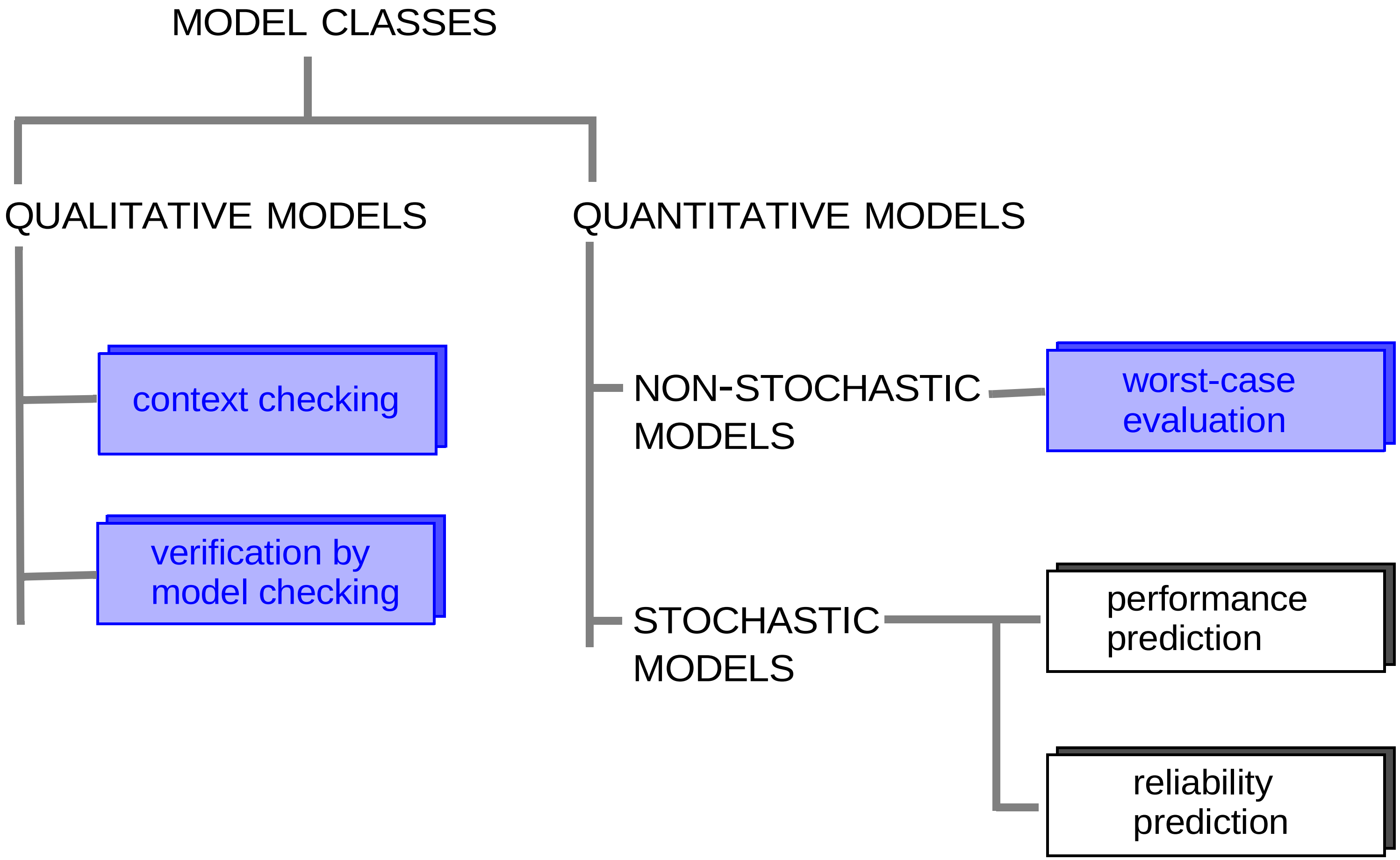
MODEL-BASED SYSTEM VALIDATION

- ☐ objective - reuse of certified components



MODEL-BASED SYSTEM VALIDATION

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



MODEL CLASSES

QUALITATIVE MODELS

context checking

verification by model checking

QUANTITATIVE MODELS

NON-STOCHASTIC MODELS

worst-case evaluation

STOCHASTIC MODELS

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

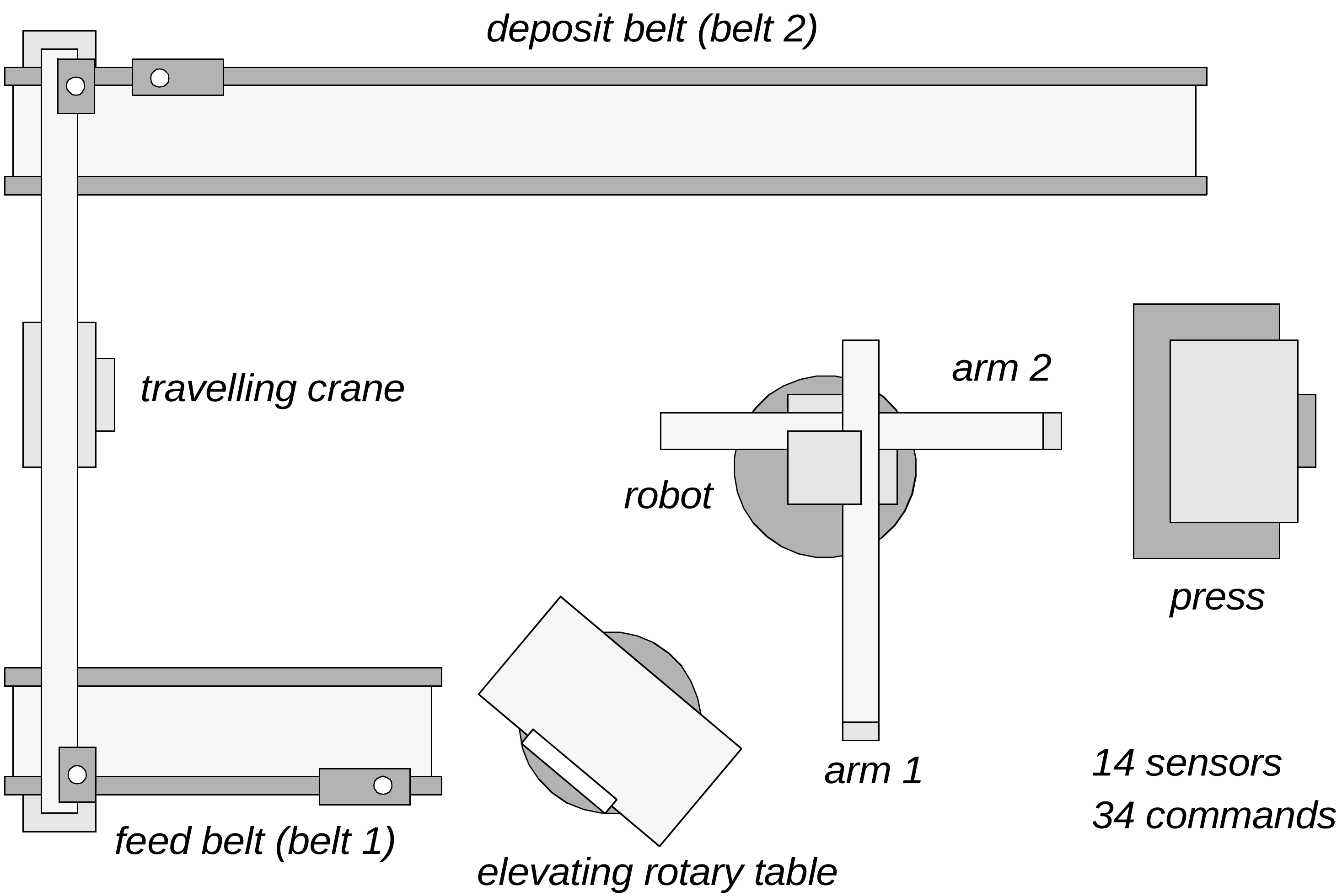
⇊

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

```
113751760865620516280672035436276768405854187694780001109285823216991 8\\
159959588122031332641120690971790713407413960379370132051412946235771 0\\
244289522738424241885324723952294300718880861927052755597203329394869 1\\
334498271287409035878953318171137286359195790723689557093738307422542 1\\
493299735055934871120872608511650262781852464476299128123872281683542 6\\
439043702222222716712699874004961590120093014497021663026892511863169 6\\
792192797756430854076755677722422066045029462353435568315492194903488 7\\
413893510872611522753508464671945735340847108696533249480549775338294 2\\
171781101168772051021154169003921176627995642292903237688541475038527 5\\
512488192401053636525511904747774118 74
```

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’

monika.heiner(at)informatik.tu-cottbus.de
data structures and software dependability

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 - DOUBTS

FAULT TOLERANCE

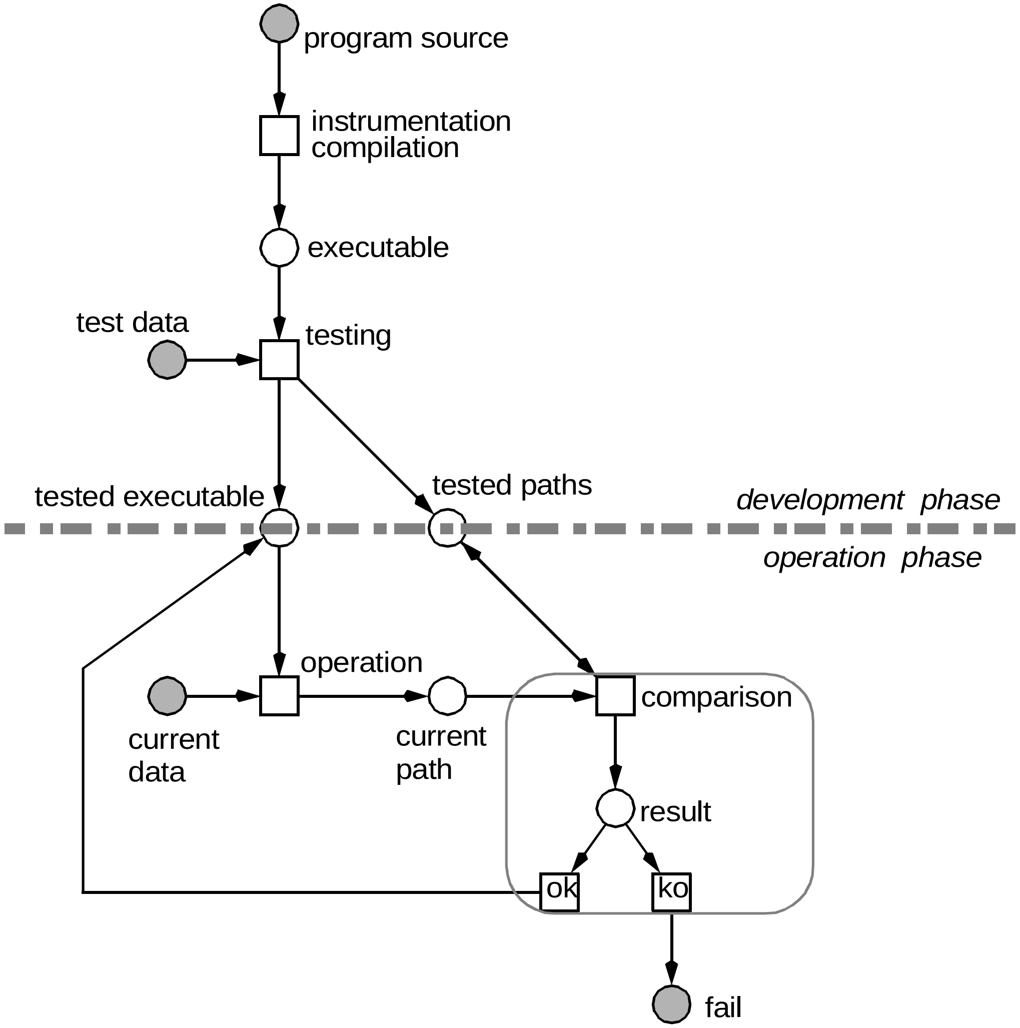
- **International Standard IEC 61508**
  Functional safety of electrical/electronic/programmable electronic safety-related systems
- **part 7**
  Overview of techniques & measures, first edition August 2002
- **Annex C**
  Overview of techniques and measures for achieving software safety integrity

- **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

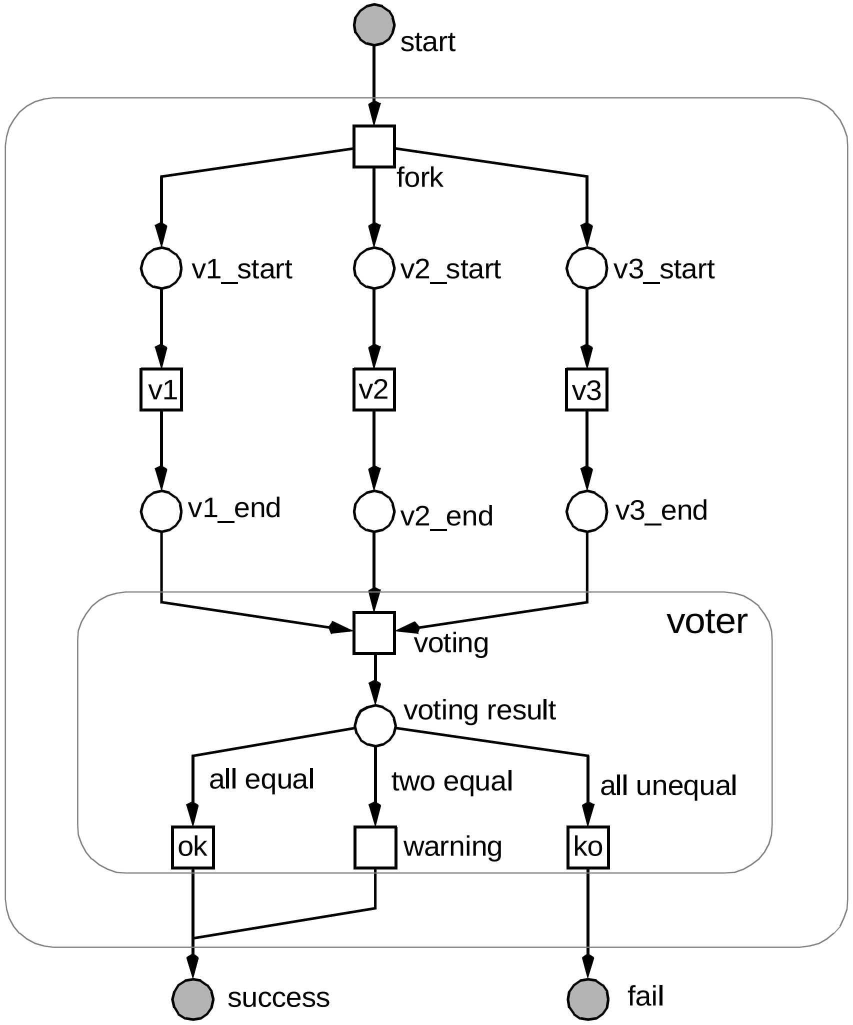
- ☐ to prevent the execution of un-known paths
- ☐ only tested paths are reliable paths
- ☐ requires excessive testing



FAULT TOLERANCE - SOFTWARE DIVERSITY

N VERSION PROGRAMMING

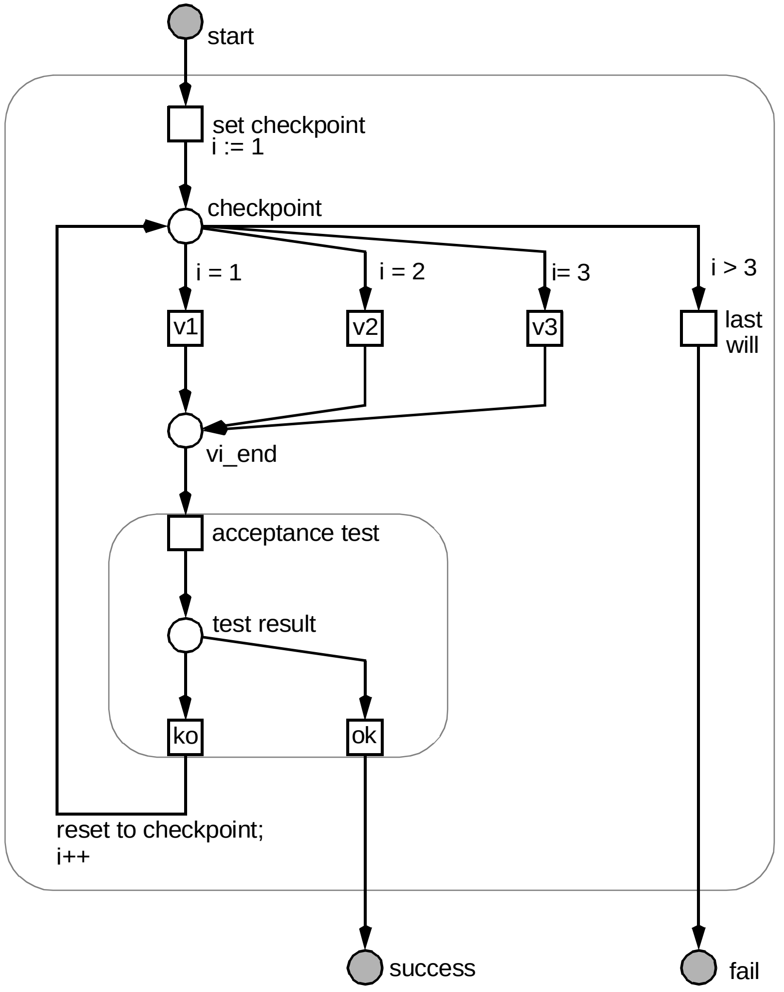
- ☐ parallel execution of n program versions
- ☐ followed by majority test
- ☐ higher abstraction level, transitions:
  - -> program versions
  - -> voting algorithm



FAULT TOLERANCE - SOFTWARE DIVERSITY

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

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 ?