Brandenburg University
of Technology at Cottbus,
Dep. of Computer Science

DEPENDABLE SOFTWARE - AN UNREALISTIC DREAM OR JUST A REALITY FAR AWAY?

MONIKA HEINER

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

MURPHY'S COMPUTER LAWS

(1) No program without faults.

(2) As soon as a program becomes useful,
it has to be changed.

(3) Each program is completed
until it is completely confused.

corollary: The program complexity grows as long as
it exceeds the programmer's skills.

(4) Team size extension of a late software project
promotes additional delays.

(5) The approaching of the project deadline comes along
with increasingly important project changes.

(6) Each important problem hides another one,
just waiting for being allowed to advent.

(7) Having sold a program six month ago,
the most important program fault will be discovered.

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 is different from other engineering disciplines*

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

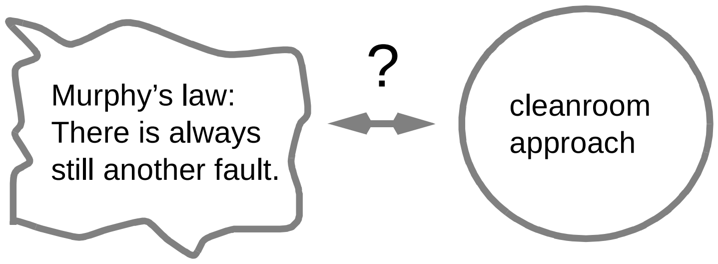
- ❑ Sw does not grow old.

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

  -> *sw cemetery*

  But, *"sw mammoths" keep us busy.*

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.*

❑ 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

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

❑ systematic testing of concurrent programs is much more complicated than of sequential ones

Bosch Workshop - Dependable Software 04/08/18

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

PN	T $\rightarrow$ TIME $\longrightarrow$	TPN
prop(pn)	$\not\Leftrightarrow$	prop(tpn)
RG (pn)	$\supseteq$	**RG** (tpn)

- result: masking of certain types of bugs:

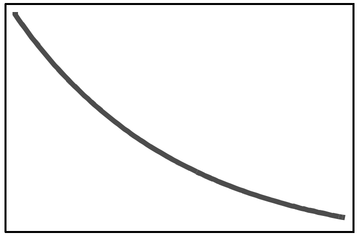
DSt (pn)	->	not DSt (tpn)
live(pn)	->	not live (tpn)
not BND (pn)	->	BND (tpn)

❑ non-deterministic behavior,
pn: time-dependent dynamic conflicts

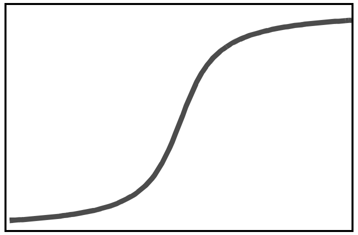
❑ dedicated testing techniques to guarantee reproducibility,
e. g. Instant Replay [LeBlanc 87]

Criteria to Finish Testing

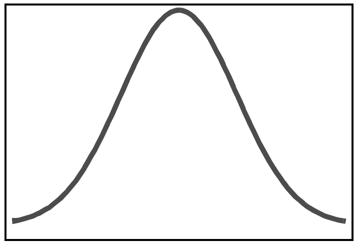
- ❑ common
  - time is over (time-to-market pressure)
  - all test cases successful
- ❑ better (?)
  - discover a given amount of bugs
  - reach a specified degree of test coverage(s)
  - reach a specified fault rate
    (number of found bugs per time)



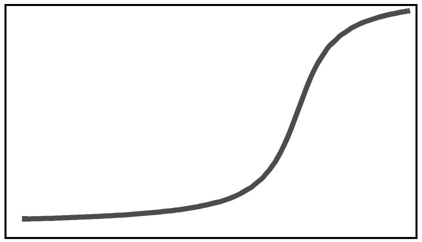
optimistic view



pessimistic view

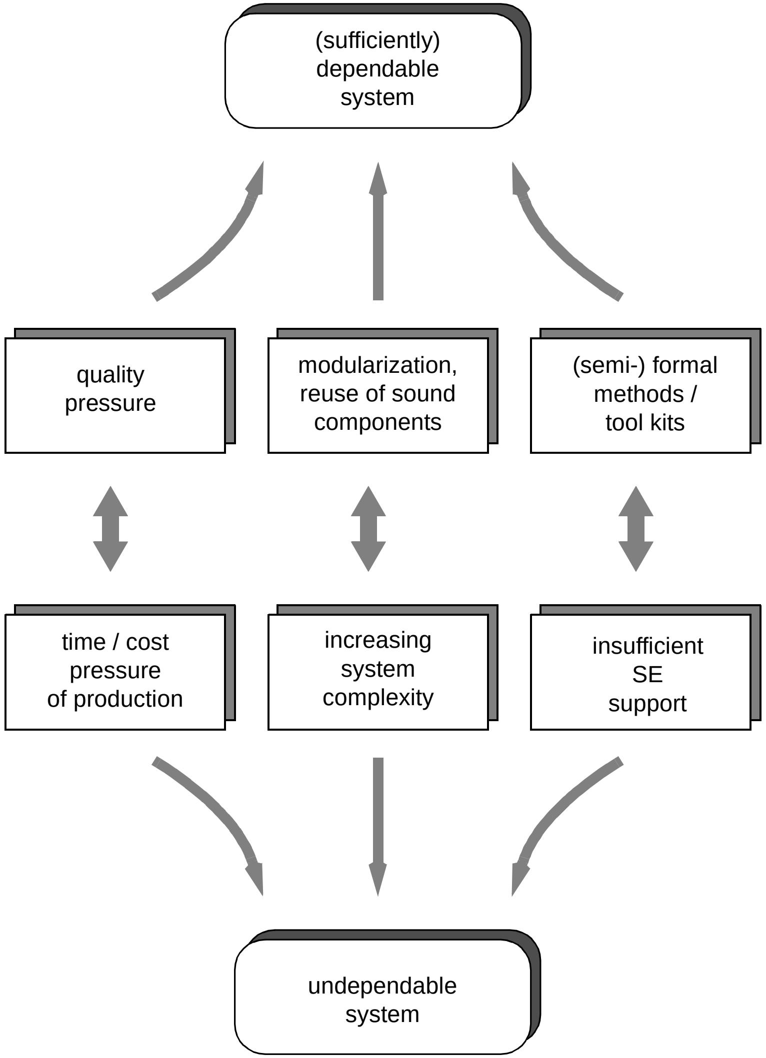


realistic view (?)



ageing model

Increasing Dependability Demands

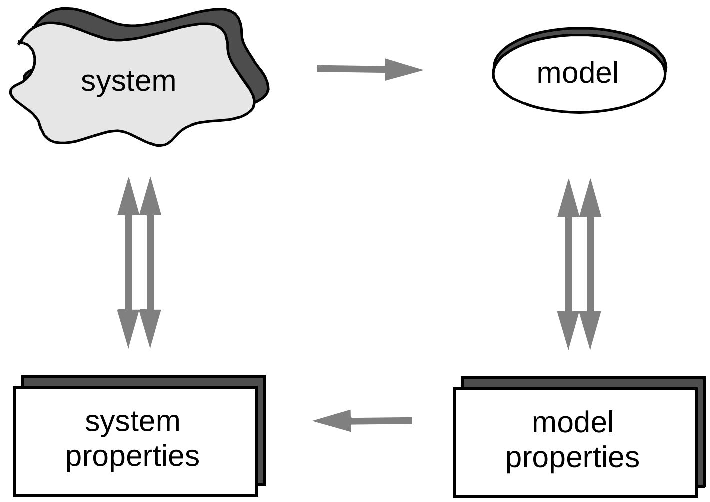


METHODS

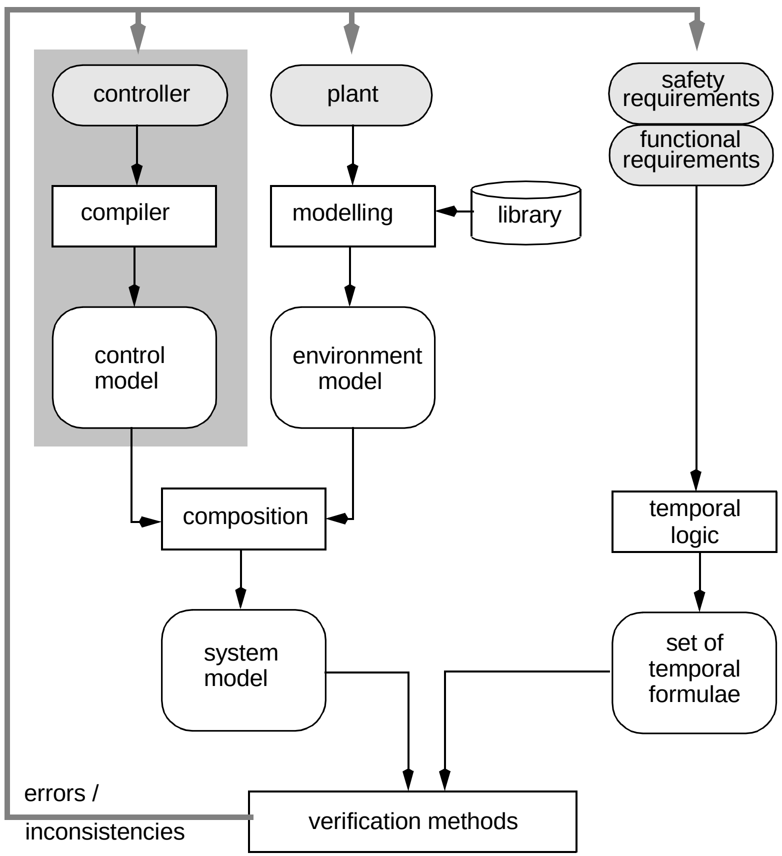
SOFTWARE DEPENDABILITY

- 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

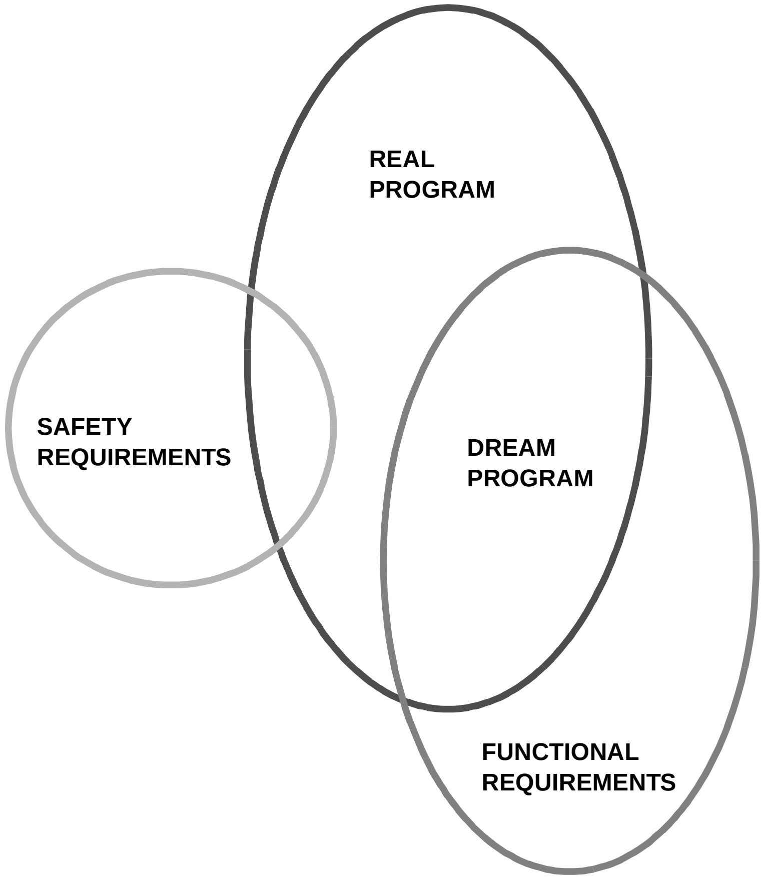
MODEL BASED SYSTEM VALIDATION, GENERAL PRINCIPLE



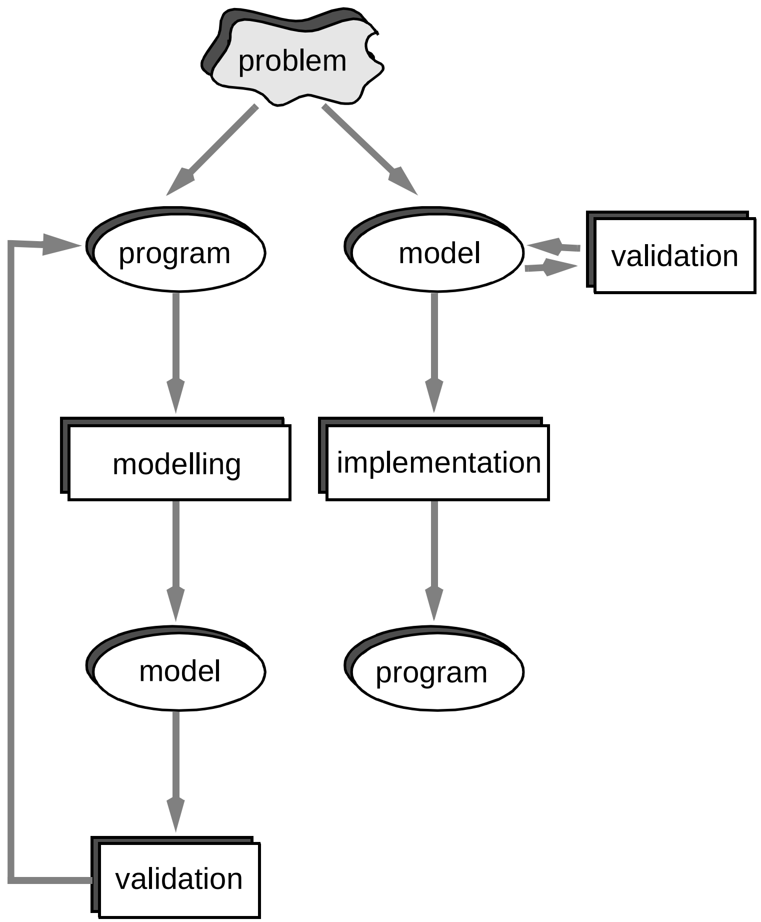
MODEL BASED SYSTEM VALIDATION, PROCESS AND TOOLS



OBJECTIVE - REUSE OF CERTIFIED COMPONENTS



SOFTWARE ENGINEERING & MODELS, TWO APPROACHES



POSSIBLE ANSWERS TO STATE EXPLOSION

'BASE CASE' TECHNIQUES

- compositional methods
  -> *simple interfaces between modules*
- abstraction by ignoring some state information
  -> *conservative approximation*

'ALTERNATIVE' METHODS (PN)

- structural analysis
- integer programming
- compressed state space representations
- lazy state space construction
  (partial order methods)
- alternative state spaces
  (partial order representations)

QUALITATIVE ANALYSIS METHODS (PN)

- NET REDUCTION
- STRUCTURAL PROPERTIES
- LINEAR PROGRAMMING
  - place / transition invariants
  - state equation
  - trap equation

} static analysis

- REACHABILITY ANALYSIS
  - (complete) reachability graph
  - compressed state spaces
    - OBDDs, ONDDS
    - Kronecker products
  - reduced state spaces
    - coverability graph
    - symmetry
    - stubborn sets
    - sleep sets
  - branching process

} dynamic analysis *(model checking)*

EXAMPLE - BDD ANALYSIS RESULT

PHIL1000:

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

Number of states: ca. 1.1 * 10e667

```
113751760865620516280672035436276768405854187694780001109285823216991 8\\
```

Time to compute P-Invariants: 45885.66 sec
Number of P-Invariants: 3000
Time to compute compact coding: 385.59 sec
Number of Clusters: 3000
Number of Variables: 4000
Iteration depth: 3

(no reordering)
Time: 3285.73 sec ca. 54.75’
DagSize: 22134
PeakDagSize: 5521866

Case Studies

ACADEMIC:

- ❑ low-level mutex algorithm
- ❑ Dijkstra’s philosophers
- ❑ Milner’s scheduler
- ❑ solitaire, . . .

MORE REALISTIC

- ❑ production cell
- ❑ concurrent pushers, . . .

TO APPEAR :

- ❑ 2-hand switch, (plc press controller)
- ❑ [Moon 92], [Probst 96], . . .

Case Studies Objectives

Modelling

- ❑ How many basic patterns (Petri net components) ?
  -> *Small set of flexible, reusable components ?*
- ❑ Adequate environment model ?
  -> *Representation of actuator & (continuous) sensor states ?*
- ❑ Suitability of the tool kit in use ?
  -> *Additional features ?*

Analysis

- ❑ Which kinds of function & safety requirements ?
  -> *Which temporal operators are really necessary ?*
- ❑ Which kinds of analysis techniques are helpful ?
  -> *Recommendable order ?*
- ❑ What about the chance to avoid state explosion ?
  -> *How strong do ’alternative’ analysis techniques work ?*

PLC WORKBENCH *SAFETY KNIGHT*, BASIC FEATURES

- dedicated technical language(s)
  - -> *functional requirements*
  - -> *safety requirements*
  - -> *performance requirements*
- combination of different analysis tools
  - -> *general Petri net framework;*
- user guidelines:
  analysis question -> analysis technique(s)
  - -> *dedicated Petri net tool kits;*
- batch processing of requirement specifications
  - -> *distributed over different tools & processors*
- libraries of reusable Petri net components
  - -> *environment model*
  - -> *cooperation patterns*

SUMMARY

- validation can only be as good as the specification
  - -> *readable <-> unambiguous*
  - -> *complete <-> limited size*
- validation is extremely time and resource consuming
  - -> *'external' quality pressure*
- sophisticated validation is not manageable
  without theory supported by tools
- 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*