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.  
  \( \rightarrow \) mastering the complexity?  
  \textbf{But,} small system's techniques cannot be scaled up easily.
- Large systems must be developed by large teams.  
  \( \rightarrow \) communication / organization overhead  
  \textbf{But,} many programmers tend to be lonely workers.
- Sw systems are abstract, i.e. have no physical form.  
  \( \rightarrow \) no constraints by manufacturing processes or by materials governed by physical laws  
  \( \rightarrow \) SE is different from other engineering disciplines  
  \textbf{But,} human skills in abstract reasoning are limited.
- Sw does not grow old.  
  \( \rightarrow \) no natural die out of over-aged and nonviable sw  
  \( \rightarrow \) sw cemetery  
  \textbf{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  
  \( \rightarrow \) standard system: \( \geq 50 \% \)  
  \( \rightarrow \) 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

**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

- 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)

**INCREASING DEPENDABILITY DEMANDS**

- (sufficiently) dependable system
- modularization, reuse of sound components
- (semi-) formal methods / tool kits
- quality pressure
- increasing system complexity
- time / cost pressure of production
- insufficient SE support
- undependable system

**Optimistic view**

**Pessimistic view**

**Realistic view (?)**

**Ageing model**
METHODS

SOFTWARE DEPENDABILITY

Fault Avoidance  \[\rightarrow\] development phase

- Fault Prevention
- Fault Removal
  - Manual
  - Computer-Aided
    - animation / simulation / testing
    - context checking (static analysis)
    - consistency checking (verification)

Fault Tolerance  \[\rightarrow\] operation phase

- Fault Masking
  - Defensive
  - Diversity
- Fault Recovery

MODEL BASED SYSTEM VALIDATION, GENERAL PRINCIPLE

System \[\rightarrow\] model

- System properties
- Model properties
MODEL BASED SYSTEM VALIDATION, PROCESS AND TOOLS

controller

plant

controller

plant

modelling

environment model

library

safety requirements

functional requirements

temporal logic

system model

set of temporal formulae

verification methods

errors / inconsistencies

OBJECTIVE - REUSE OF CERTIFIED COMPONENTS

REAL PROGRAM

SAFETY REQUIREMENTS

DREAM PROGRAM

FUNCTIONAL REQUIREMENTS
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

REACHABILITY ANALYSIS

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

EXAMPLE - BDD ANALYSIS RESULT

PHIL1000:

Number of places/marked places/transitions:

7000/2000/5000

Number of states: ca. 1.1 * 10e667

11375176086562051628067203543627676840585418769478000110928582321669189
1599968812020333264112069097179071340741396037937013320514129462357710
244289852273842424188653247239522943007188806619270527555979203329948691
334496271287490358789533181711372863591957907236895579093830704225211
493299735055934871102872608511650262781852464476299112812387228166354268
439043702222222717126998740049615901209301449702166302689251186316902
7921979777543085470675677722422066045029462353435568315492149034867
4138935108726115275350846467194573534084710866533249480549775383289241
17178101168772051021154169003921176279564229290337685414750385275
5124861924010536365255190474774116784

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