
Verified Software Systems

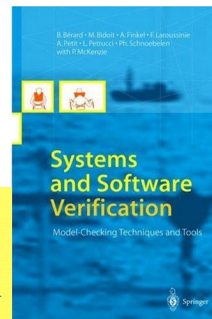
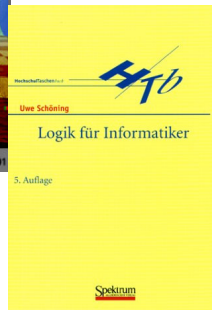
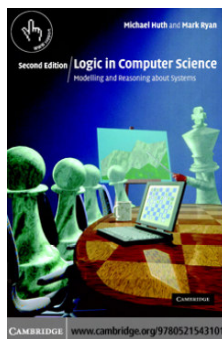
Rainer Marrone

Software Systems Institute (STS)

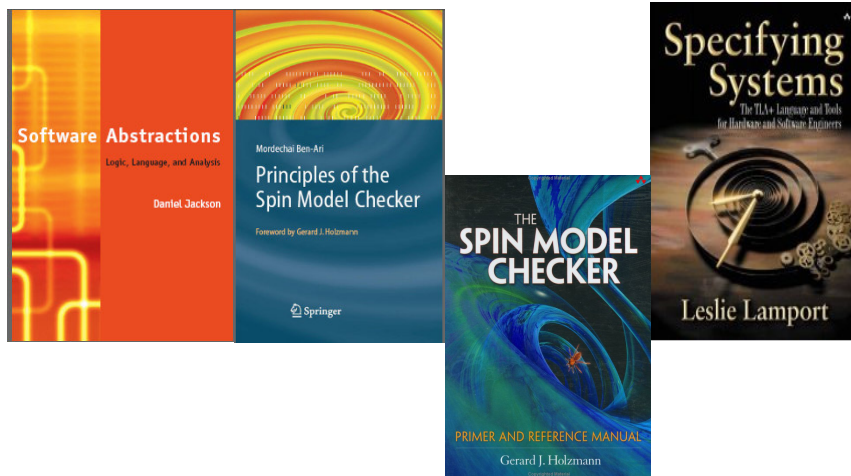
Harburger Schlosstraße 20
21073 Hamburg

<http://www.sts.tu-harburg.de/teaching/>

Literature



Literature



Literature – Introductory Papers

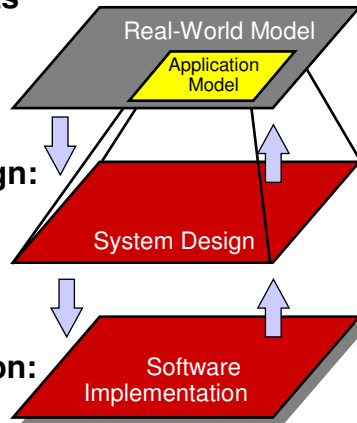
- J.M. Wing, A Specifier's Introduction to Formal Methods. IEEE, Computer, 23(9):8-24, September 1990.
- E. Clarke and J. Wing, Formal Methods: State of the Art and Future Directions, ACM Computing Surveys, 28(4):626-643, 1996.

Introduction – Software Systems

Requirements Analysis:
Why?

System Design:
What?

Software Implementation:
How?

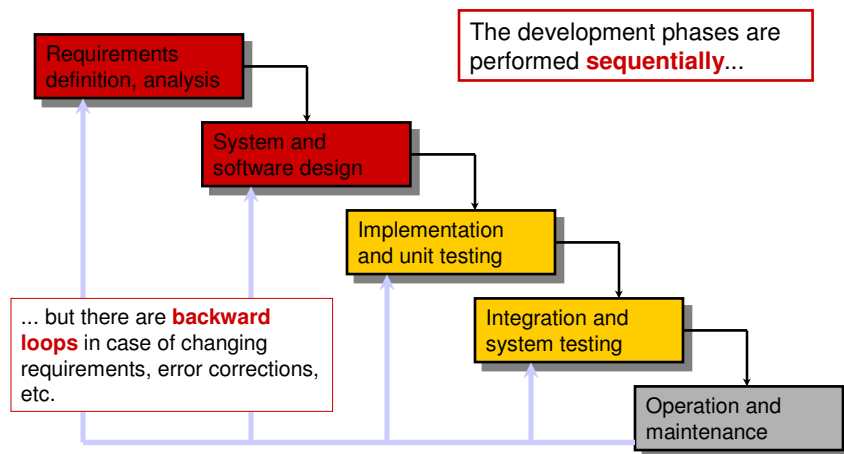


- requirements
- domain knowledge
- goals

- abstractions
- models
- structures
- architecture

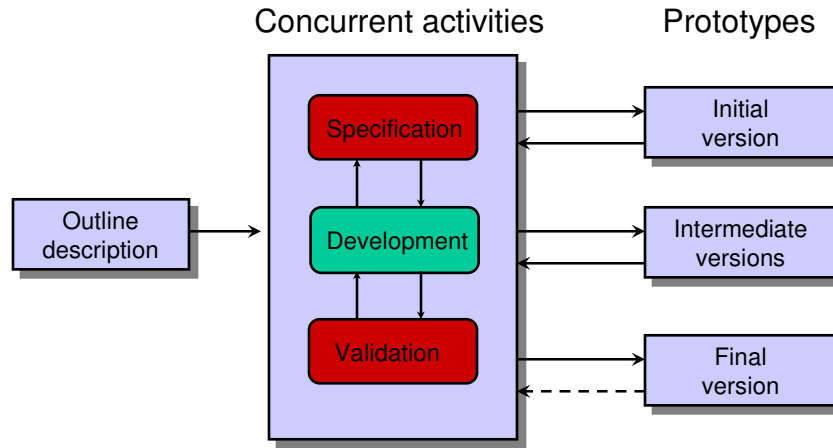
- algorithms
- data structures
- generic services
- platform-specific services

The Waterfall Process



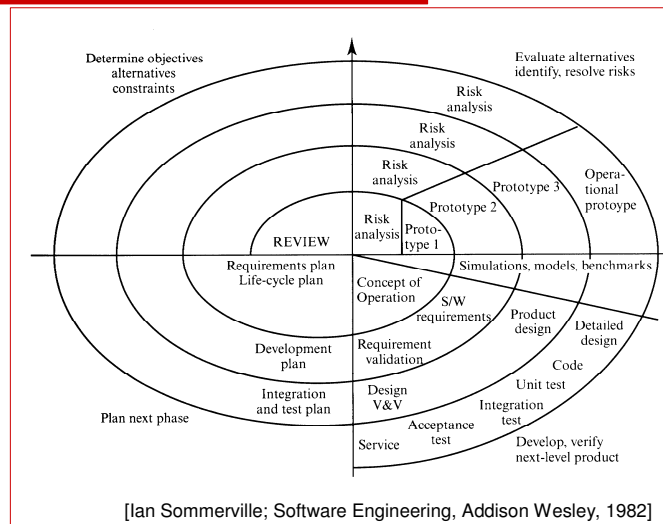
[Ian Sommerville; Software Engineering, Addison Wesley, 1982]

Evolutionary Development



[Ian Sommerville; Software Engineering, Addison Wesley, 1982]

Boehm's Spiral Model



[Ian Sommerville; Software Engineering, Addison Wesley, 1982]

Central Question

- Does the system meet the requirements?
 - **Correctness** of the system

- Validation is necessary

Validation & Verification

- **Validation** : The process of evaluating software at the end of software development to ensure compliance with intended usage

- **Verification** : The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

IV&V stands for "*independent verification and validation*"

Validation & Verification

- Validation and Verification processes go hand in hand, but visibly Validation process starts after Verification process ends (after coding of the product ends).
 - Each Verification activity (such as Requirement Specification Verification, Functional design Verification etc.) has its corresponding
 - Validation activity (such as Functional Validation/Testing, Code Validation/Testing, System/Integration Validation etc.).

Validation - Simulation

- Simulation of sequences of operations (randomly selected)
- Does not check the implementation but the system model created within the design
- Should be done in an early stage
- Tool support is available
 - Not all of the possible sequences are tested
 - Modified Dijkstra:
"Simulation can only show the presence of errors never their absence."

Example - Simulation

- Examples:
Hardware, Circuit Design
- Simulation based on the model of the chip
 - Input is given
 - Behavior is checked
 - Functionality is checked through tests

Validation – Static Analysis

- Program analysis:
syntactical analysis of the program code
- Question:
 - Unused variables
 - Assignments that are never used
- During compilation – optimization
- Checking certain properties

Validation - Test

- Within the Waterfall Process: Tests
 - Running the software
 - Costs about 30%-50% of all costs
 - The later the more changes are required and the costs will increase
 - Dijkstra:
"Testing can only show the presence of errors never their absence."

White-box and Black-box Testing

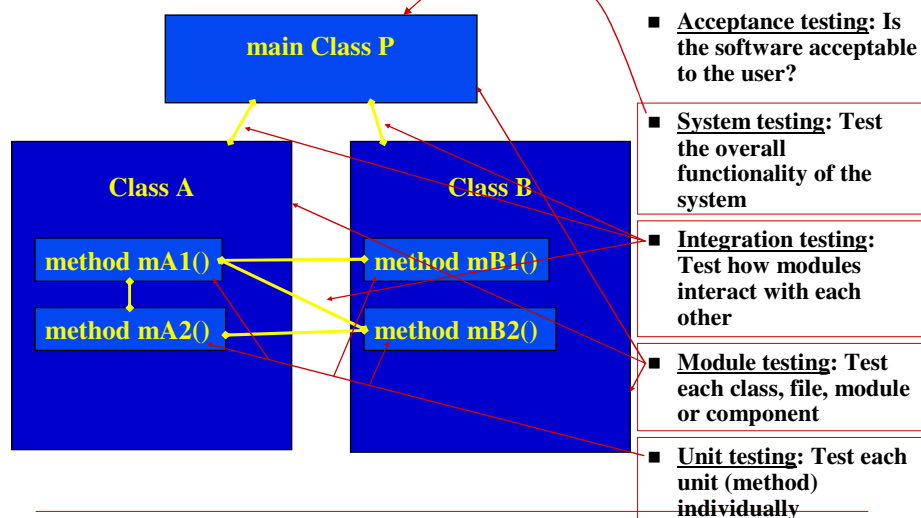
- **Black-box testing** : Deriving tests from external descriptions of the software, including specifications, requirements, and design
- **White-box testing** : Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements

This view is really out of date.
The more general question is: *from what level of abstraction do we derive tests?*

Changing Notions of Testing

- Old view of testing is of testing at specific software development phases
 - Unit, module, integration, system ...
- New view is in terms of structures and criteria
 - Graphs, logical expressions, syntax, input space

Old : Testing at Different Levels



Old : Find a Graph and Cover It

- Tailored to:
 - a particular software artifact
 - code, design, specifications
 - a particular phase of the lifecycle
 - requirements, specification, design, implementation

- This viewpoint obscures underlying similarities

- Graphs do not characterize all testing techniques well

- Four abstract models suffice

New : Test Coverage Criteria

A tester's job is **simple** : Define a model of the software, then find ways to cover it

- **Test Requirements** : Specific things that must be satisfied or covered during testing

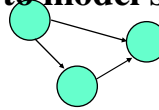
- **Test Criterion** : A collection of rules and a process that define test requirements

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures ...

New : Criteria Based on Structures

Structures : Four ways to model software

1. Graphs



2. Logical Expressions

(not X or not Y) and A and B

3. Input Domain Characterization

A: {0, 1, >1}

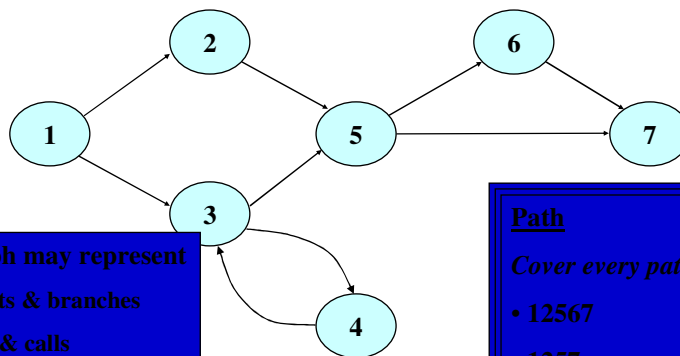
B: {600, 700, 800}

C: {swe, cs, isa, infs}

4. Syntactic Structures

```
if (x > y)
  z = x - y;
else
  z = 2 * x;
```

1. Graph Coverage – Structural



This graph may represent

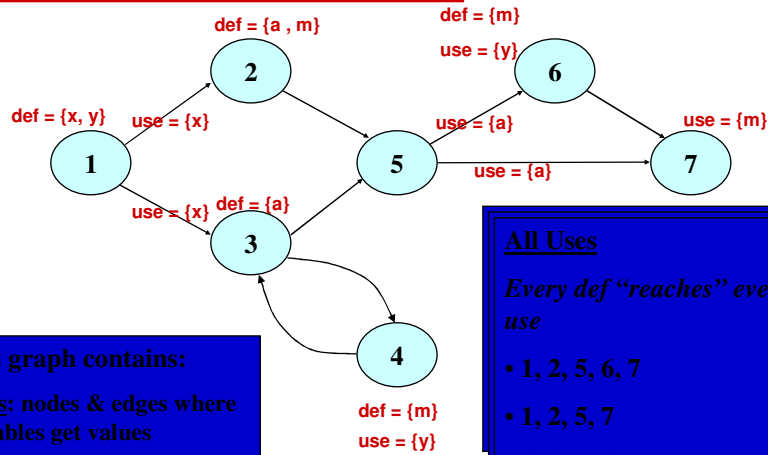
- statements & branches
- methods & calls
- components & signals
- states and transitions

Path

Cover every path

- 12567
- 1257
- 13567
- 1357
- 1343567
- 134357 ...

1. Graph Coverage – Data Flow



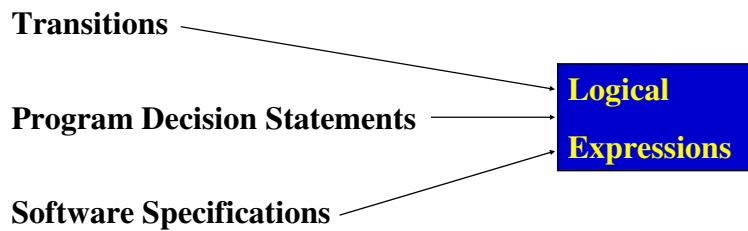
This graph contains:

- **defs:** nodes & edges where variables get values
- **uses:** nodes & edges where values are accessed

- All Uses**
- Every def “reaches” every use
- 1, 2, 5, 6, 7
 - 1, 2, 5, 7
 - 1, 3, 5, 6, 7
 - 1, 3, 5, 7
 - 1, 3, 4, 3, 5, 7

2. Logical Expressions

$$((a > b) \text{ or } G) \text{ and } (x < y)$$



2. Logical Expressions

$$((a > b) \text{ or } G) \text{ and } (x < y)$$

- **Predicate Coverage** : Each predicate must be true and false
 - $((a > b) \text{ or } G) \text{ and } (x < y) = \text{True, False}$
- **Clause Coverage** : Each clause must be true and false
 - $(a > b) = \text{True, False}$
 - $G = \text{True, False}$
 - $(x < y) = \text{True, False}$
- **Combinatorial Coverage** : Various combinations of clauses
 - *Active Clause Coverage*: Each clause must determine the predicate's result

2. Logic – Active Clause Coverage

$$((a > b) \text{ or } G) \text{ and } (x < y)$$

With these values for G and (x<y), (a>b) determines the value of the predicate

1	T	F	T
2	F	F	T
3	F	T	T
4	F	F	T
5	T	T	T
6	T	T	F

duplicate

3. Input Domain Characterization

- Describe the **input domain** of the software
 - Identify **inputs**, parameters, or other categorization
 - Partition each input into **finite sets** of representative values
 - Choose **combinations** of values
- **System level**
 - Number of students $\{ 0, 1, >1 \}$
 - Level of course $\{ 600, 700, 800 \}$
 - Main courses $\{ se, cs, mech \}$
- **Unit level**
 - Parameters $F (int X, int Y)$
 - Possible values $X: \{ <0, 0, 1, 2 \}, Y: \{ 10, 20, 30 \}$
 - Tests
 - $F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F(3, 10)$

4. Syntactic Structures

- Based on a **grammar**, or other syntactic definition
- Primary example is **mutation testing**
 1. Induce **small changes** to the program: **mutants**
 2. **Find tests** that cause the mutant programs to fail: **killing mutants**
 3. Failure is defined as **different output** from the original program

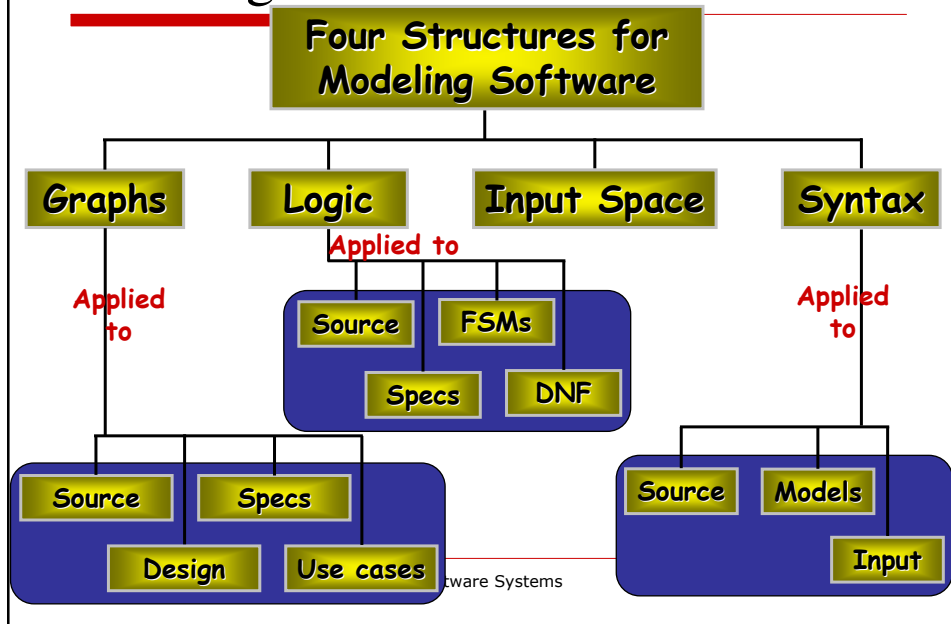
```
if (x > y)
  z = x - y;
else
  z = 2 * x;
```

```
if (x > y)
  Δif (x >= y)
    z = x - y;
  Δ z = x + y;
else
  z = 2 * x;
```

Source of Structures

- These structures can be **extracted** from lots of software artifacts
 - **Graphs** can be extracted from UML use cases, finite state machines, source code, ...
 - **Logical expressions** can be extracted from decisions in program source, guards on transitions, conditionals in use cases, ...
- **Model-based testing** derives tests from a model that describes some aspects of the system under test
 - The model usually describes part of the **behavior**
 - The **source** is usually **not** considered a model

Coverage Overview



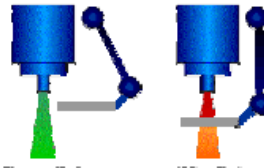
Validation - Verification

- Verification:
 - Mathematical proof of the correctness of the system model in the context of the defined requirements
 - "absence of errors" is proofed (only certain "errors")
- Precondition:
 - **Formal** description of the **model** and the **requirements**.
 - ! Higher complexity (additional costs, training of the workforce)

Why verification – Bug 1

Therac-25 Accident :

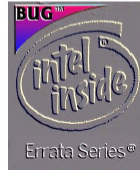
- X-ray machine with two modes
 - X-rays, generated high energy electron-beam directed on metal shield (between beam and patient)
 - Low energy electron-beam without metal target
- A software error let operator inadvertently select high energy beam without metal shield.
- Results: At least five patients die.



Why verification – Bug 2

Pentium bug (1994)

- First release of Intel Pentium chip
- Mistakes when dividing floating-point numbers that occur within a specific range
- Estimated 3 million to 5 million defective chips
- PR nightmare for Intel
- Cost : \$475 million



Comp3153 Ansgar Fehnker

Why verification – Bug 3

Ariane 5 (1996)

- Ariane 5 used software used prior in Ariane 4
- 64-bit floating-point to 16-bit integer generated conversion an overflow
- Error was caught, sub-system shut down
- Back-up systems failed for the same reason.
- Rocket veered off course.
- Control system decided to abort mission.
- Result: Rocket self-destructed
- Cost : \$400 million payload



Comp3153 Ansgar Fehnker

Why verification – Bug 4

Code Red:

- Potential buffer over-flow in Microsoft Internet Information Server
- Worm uses exploit. It sends specially crafted packets.
- Triggering a buffer overflow
- Giving worm administrative privileges to the worm
- Cost: > \$2 billion.



Comp3153 Ansgar Fehnker

Verification – When/What

- Verification of the system model at an early stage (+):
 - Requires a formal model and the formalization of the requirements.
- Verification of the software at a late stage (-):
 - Requires formal semantic of the implementation language (very hard)
 - And verification techniques (very hard)

Modeling languages

- Model: abstract description of a system
- The model is specified by using a modeling language (e.g. UML)

- Our goal:
 - Mathematical proof of the correctness of the model regarding the requirements
 - We need a precise unambiguous description of the model.
- → Modeling language with semantic
- UML does not have a strong semantic

Formal Methods ...

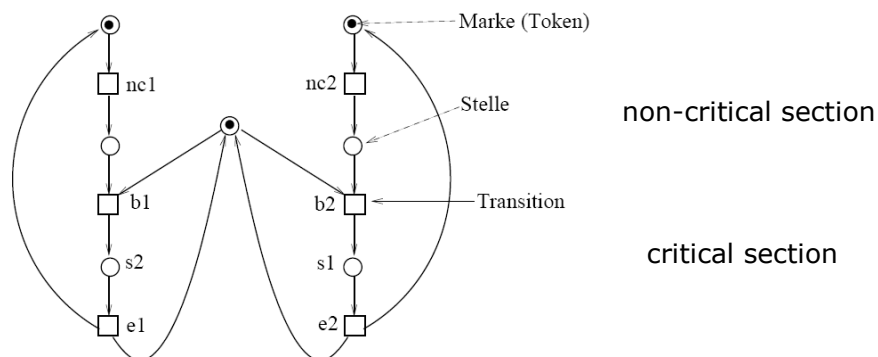
- ... are modeling languages for the system design
- ... have a formal, mathematical defined semantic
- ... are often associated with analysis techniques

Zoo of formal methods

- Z, Alloy, CCS, CSP, LOTOS, π -Calculus, Petri net, ASMs, Larch, FOL, LTL, CTL, CTL*, BDD, ...
- Active research area with lots of conferences:
 - FME: Formal Methods Europe
 - FASE: Fundamental Approaches to SE
 - CONCUR: Concurrency Theory
 - ICFEM: Formal Engineering Methods

Example Petri Net

- Two parallel processes not allowed to be both in the critical section.



Petri net

- Defined semantic of the Token Game
 - A transition can fire if all channels contain tokens.
 - the tokens are taken from the input places and are carried to the places behind the transition
- Analysis of a property
 - "could there be a mark in s1 and s2 at the same time?"
 - Example for "Model has Property"

Calculus of Communicating Systems

- CCS – Process algebra
 - Buffer of size 2 described by process equations (Model in CCS):
 - Empty = in.Half
 - Half = out.Empty + in.Full (+ is choice)
 - Full = out.Half

Sequential description

See <http://www.cs.sunysb.edu/~cwb/> - Concurrency Workbench of the New Century (CWB-NC) - for a tool for verifying CSS specifications.

Implementation of the Buffer

- Composition of 2 Buffers of size 1
 - $Puf1 = \overline{\text{in}}.\text{mid}.Puf1$
 - $Puf2 = \overline{\text{mid}}.\text{out}.Puf2$
 - $\text{Sys} = (Puf1|Puf2) \setminus \{\text{mid}\} // | \rightarrow \text{parallel}$

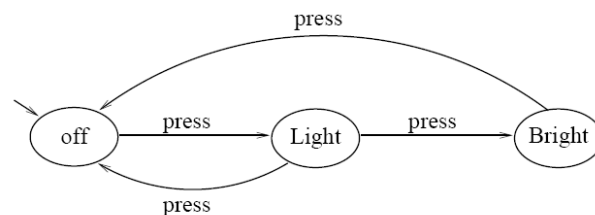
Parallel description

- Is such an implementation correct?
 - $\text{Empty} = \text{Sys}?$

Example for “is the model of the next layer correct in respect with the prior layer”

Example: Intelligent Light Switch

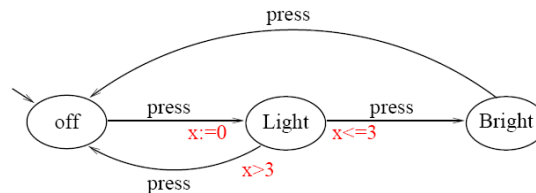
- Behavior: press – on, press – out, press two times quickly – light gets brighter
- As Finite Automata:



how can we model “press two times quickly”

As Timed Finite Automata

- Add Clock



- $x := 0$ Reset, set clock to 0,
- $x \leq 3, x > 3$ Guards, define transition conditions.
Question: can the light be made brighter?
- Example for "Model has Property"

Logic & Specification

- The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such a way that we can **reason about** them **formally**.

- **Example**

If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. *Therefore*, there were taxis at the station.

Example Alloy

- Alloy combines predicate calculus and relational calculus.
- Suppose you want to formally specify a Tree.

```
sig Node {
  children: set Node,
}
one sig Root extends Node {}

fact {
  Node in Root.*children // * means transitive closure
}
```

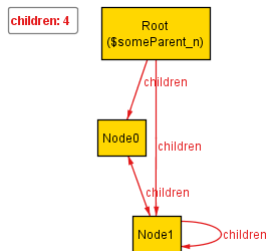
Example Alloy

- Suppose you now want to specify that it should hold, that every Node has some parent.

```
assert someParent {
  all n: Node | some children.n
}
check someParent for 3
```

- the **check** directive informs the analyzer of Alloy that it should try to find a counterexample of the assertion `someParent` with at most three elements for every signature.

Solution - counterexample



Example Alloy

- Changing the assertion solves the problem.

```
assert someParent {
  all n: Node - Root | some children.n
}
check someParent for 3
```

- Leads to no **counterexample** found!

```
Alloy Analyzer 4.1.0 (build date: 2008/08/07 16:29 EDT)
Executing "Check someParent2 for 3"
  solver:z3(1/0) | Backend: NaiveP1 | SLOemDepth=1 | Symmetry=20
  159 vars, 14 primary vars, 171 dauses, 109ms.
  No counterexample found. Assertion may be valid. Ops.
```

Limitations of Formal Methods

Possible reasons for errors

- Model/Program is not correct (does not satisfy the specification)
Formal verification proves absence of this kind of error
- Model/Program is not adequate (error in specification)
Formal specification/verification avoid/find this kind of error
- Error in operating system, compiler, hardware
Not avoided (unless compiler etc. specified/verified)

No full specification/verification

In general, it is neither useful nor feasible to fully specify and verify **large software systems**. Then, formal methods are restricted to:

- Important parts/modules
- Important properties/requirements

The Main Point of Formal Methods is Not

- To show “correctness” of entire systems
(What IS correctness? Always go for specific properties!)
- To replace testing entirely
- To replace good design practices

There is no silver bullet that lets you get away without writing crystal clear requirements and good design, in particular, Formal Methods aren't one

But

- Formal proof can replace many test cases
- Formal methods can be used in automatic test case generation
- Formal methods improve the quality of specifications

Plan for the lecture

- Recapitulation:
 - Propositional logic, predicate logic, satisfiability, entailment, Temporal logic (next lectures)
- TLA, +CAL, Alloy, Snip, NuSVM, ...
- UML, OCL
- Testing

Exercises and Lab Classes

- First exercise on 4 Nov 2008
- Location: HS20, room 215

Contact persons:

Miguel Garcia

miguel.garcia@tu-harburg.de

Exam

- Date can be found at
<http://intranet.tu-harburg.de/stud/pruefung/index.php3>
- In written form
- Closed-book